Deliverable D2.1.b

Language specification v2

1 Introduction **2**

2 SysML/KAOS requirements modeling language **2**
2.1 Overview of the goal model 2
2.2 Illustration 3

3 SysML/KAOS domain modeling language **3**
3.1 Current state on domain modeling in requirements engineering 3
3.2 Our approach 4
3.2.1 Context model 4
3.2.2 Domain model 4

4 Conclusion **5**

Appendix A **8**

Appendix B **14**

Appendix C **34**

1 Introduction

The Formose ANR project (ANR-14-CE28-0009) aims to design a formally-grounded, model-based requirements engineering (RE) method for critical complex systems, supported by an open-source environment. The project has been launched on November 17, 2014. The main partners are: ClearSy, LACL, Institut Mines-Telecom, OpenFlexo, and THALES.

One of the main issues in the domain of RE for critical complex systems is to take into account the high complexity of such systems, the need of a better integration of RE with verification and validation techniques to ensure a better quality of requirements, and more generally the need of method guidance and tool support during the process of elaborating high quality requirements models.

The aim of Work Package 2.1 (WP2.1) is to elicit a set of concepts for RE and then to define its abstract syntax as a meta-model. All this work will be inspired by the case studies from WP1.1, as well as from the academic state of the art and from the state of practice in the technical fields known to the partners. Two deliverables are planned in WP2.1:

– D2.1.a: Language specification v1

– D2.1.b: Language specification v2

This document corresponds to the second deliverable: D2.1.b.

Our aim is to define a requirements modeling language integrating basic concepts of existing languages, such as KAOS or Tropos/*i**, and adding new ones to take into account the specific characteristics of critical complex systems: their abstract architecture will be considered by allowing requirements to be defined at different abstraction layers and verifying their consistency; the language will allow to specify not only non-functional requirements related to safety and performance but also specific requirements related to the presence of different operational modes and reconfigurations in such systems. The language will be multi-views (natural language, graphical notations, formal notations) to be understandable by all the stakeholders. For verification purpose, we will adopt existing and complementary formal methods, supported by efficient tools.

The first deliverable focused on the state of the art. This deliverable describes the requirements engineering language we have adopted in the project. To specify all the informations necessary in requirements enginering, two kinds of models are used: a *product* model and a *process* model. A *product* model describes the product (system) that will be achieved. It is composed of a *product* requirements model and a domain model. The *process* model describes the process used to build the *product* model. It is composed of a specification of the *Process* and a context model. Section 2 summarizes the `SysML/KAOS` requirements modeling language and Section 3 presents the `SysML/KAOS` domain modeling language. The *process* model is described in the Deliverables D4.1.b and D4.1.c.

2 SysML/KAOS requirements modeling language

2.1 Overview of the goal model

In the conclusion of the first deliverable (Deliverable D2.1.a : Language specification v1), we state that, based on the state of the art and on the first case studies, we have chosen to use the `SysML/KAOS` approach as the requirements modeling language. The language is detailed in [GSL13], [GS10], [GS11], we just give an overview. The `SysML/KAOS` language is an extension of the SysML requirements language[OMG12] with the most relevant concepts of the `KAOS` goal model [DvLF93] and the NFR Framework [CNYM00]. Several models exist to represent goal oriented requirements such as i* [Yu97], Goal-Based Requirements Analysis Method (GBRAM) [Ant96]. The choice of KAOS is motivated by the following reasons. Firstly, it permits the expression of several models (goal, agent, object, behavioral models) and relationships between them. Secondly, KAOS provides a powerful and extensive set of concepts to specify goal models. This facilitates the design of goal hierarchies with a high level of expressiveness that can be considered at different levels of abstraction. As SysML is an extension of UML, it provides concepts to represent requirements and to relate them to model elements, allowing the definition of traceability links between requirements and system models. However the set of SysML concepts for requirements modeling is not as extensive as in goal models. The objective of the `SysML/KAOS` language is to take advantage of both models while considering functional and non functional requirements from the earlier development phase.

In `SysML/KAOS`, a functional goal prescribes intended behaviors where some target condition must sooner or later hold whenever some other condition holds in the current system state (this state is an arbitrary current one). Non functional goals express qualities of the system to be developed such as efficiency, accuracy, security and so on. A goal model is an AND/OR graph where higher-level goals can be refined into lower-level sub-goals, and then, recursively, into low-level sub-goals that lead to the satisfaction of requirements of the system-to-be. When a goal is AND-refined into sub-goals, all of them must be satisfied for the parent goal to be satisfied. When a goal is OR-refined, the satisfaction of one of them is sufficient for the satisfaction of the parent goal.

An important concept of `SysML/KAOS` is the analysis and modeling of the impact of non-functional goals on functional goals, which can be expressed in different manners. We have shown that non-functional goals may have an impact on the

choices and decisions that are taken when refining functional goals and when transforming them into target systems. In addition, analyzing non-functional goals can lead to the identification of new functional goals, which must be integrated with the existing functional goal hierarchy.

Therefore, the SysML/KAOS approach provides three main steps. In the first step, functional and non-functional requirements are specified in two separate goal models. Then, the impact of the non functional requirements on functional ones are analyzed and described. In the last step, a final integrated goal model is obtained.

Figure 1 presents the complete metamodel of the SysML/KAOS goal language.

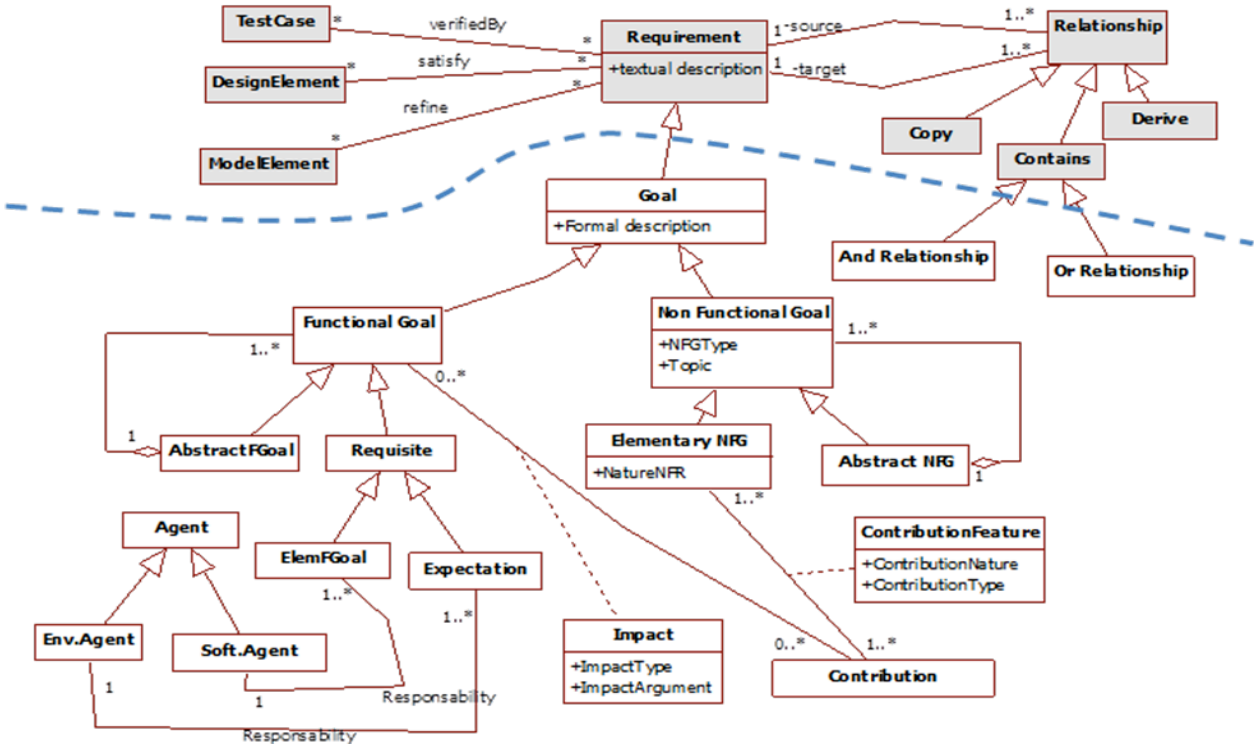

Figure 1: The SysML/KAOS goal metamodel

In the FORMOSE project we have extended the language to consider time-relative requirements since they are very important in critical systems, violation of which can put human lives at risk or damage assets and then very quickly lead to critical situations. This extension is described in Appendix A.

2.2 Illustration

To illustrate the SysML/KAOS method, we use an excerpt of the landing gear system case study which deals with the safe extension/retraction of the landing gear of an aircraft [BW14]. Figure 2 is an excerpt of the goal model, focused on the purpose of landing gear expansion (**makeLGExtended**). The **AND** refinement operator is used to specify the subgoals that must be achieved to realize a parent goal. For instance, to achieve the extension of the landing gear, the handle must be put down (**putHandleDown**) and landing gear sets must be extended (**makeLSExtended**). To extend the landing gear sets, the landing gear doors must be opened (**makeDoorsOpen**), the gears must be extended (**makeGearsExtended**) and the landing gear doors must be closed in order to stabilise the gears (**makeDoorsClosed**). Conversely, the **OR** refinement operator is used to specify distinct ways of achieving a parent goal: the realization of one of the subgoals is sufficient to achieve the parent goal. For instance, the handle can be automatically put down by a software agent (**putHandleDownAutomatically**), if the extension conditions can be clearly specified such as *the altitude below 50 ft*. However, another possibility is to let the pilot manually put down the handle (**putHandleDownManually**).

3 SysML/KAOS domain modeling language

3.1 Current state on domain modeling in requirements engineering

A domain model gathers all the informations on the domain of the studied system that must be known and understood to allow, on one hand, to specify the system requirements and, on the other hand, to implement and verify the system. It

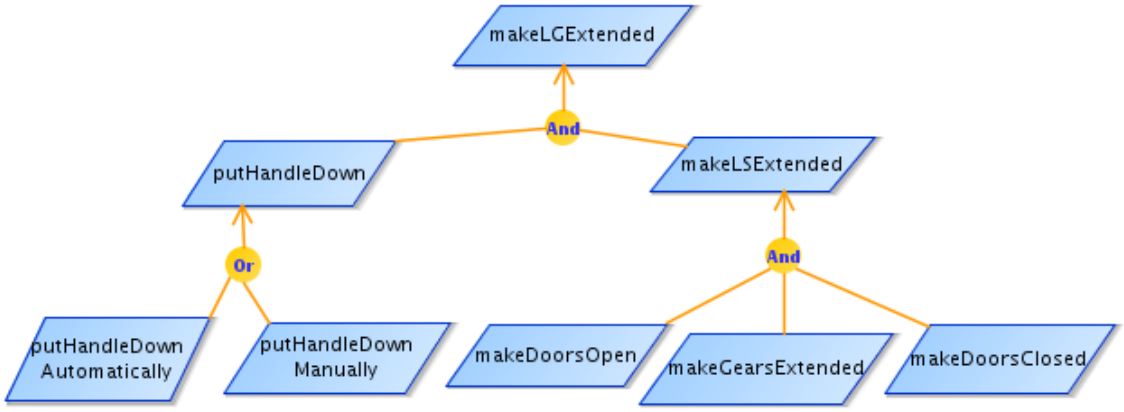

Figure 2: Excerpt from the landing gear system goal diagram

can be perceived at different levels of abstraction. Manfred Broy [Bro13] states that a domain model is composed of the following elements:

- the vocabulary, terminology that describes the concepts, the data types, the domain functions and the domain rules
- a context model that describes the different elements that interact with the system, such as the software, the physical systems and the stakeholders. It defines the boundary between the system and its environment.

In [vL09], the domain of a system is specified by an *object model* described by *UML* class diagrams. An object within this model can be an ***entity*** if it exists independently of the others and does not influence the state of any other object, an ***association*** if it links other objects on which it depends, an ***agent*** if it actively influences the system state by acting on other objects or an ***event*** if its existence is instantaneous, appearing to impulse an update of the state of the system. This approach, which is essentially graphic and little formal as argued in [MC01] is difficult to exploit in case of critical systems [NVLG14]. Moreover, it does not offer mechanisms for referencing a model within another, which limits the reusability of models.

In [ML16], in addition to *UML* class diagrams, the representation of the system domain involves *UML* object diagrams and ontologies. The case study presented reveals the use of ontologies for the representation of domain knowledge; the model obtained is named the ***domain model***. Furthermore, object and class diagrams are used in the modeling of system structure and constraints and lead to the ***structural model*** which must conform to the *domain model*. This approach, like the previous one, uses *UML* diagrams which are graphical representations, not very formal [MC01] and slightly expressive [NVLG14]. Moreover, the use of several languages for system domain modeling obliges the user to master and manipulate them all, which is an extra source of complexity.

In [NVLG14], ontologies are used not only to represent domain knowledge, but also to model and analyze requirements. The proposed methodology is called *knowledge-based requirements engineering (KBRE)* and is mainly used for detection and processing of inconsistencies, conflicts and redundancies among requirements. In spite of the fact that *KBRE* proposes to model domain knowledge through ontologies, the proposal focuses on the representation of requirements and proposes nothing regarding domain modeling.

3.2 Our approach

3.2.1 Context model

The metamodel that allows to describe context models is an extension of the SysML metamodel. It is presented in Figure 3. A *Context Definition Diagram* is composed of one *System Block* that represents the studied system and several *Context Relationship*, each one relates a system to an entity (*Context Block*) of its environment.

Figure 4 shows the context model of a landing gear system that is a part of an aircraft.

3.2.2 Domain model

In order to model the vocabulary of the domain model we use the concept of ontology. Our approach is described in the article contained in Appendix B. It is a pre-print version of the article published in a book containing works from the NII Shonan meeting called "Implicit and explicit semantics integration in proof based developments of discrete systems" in

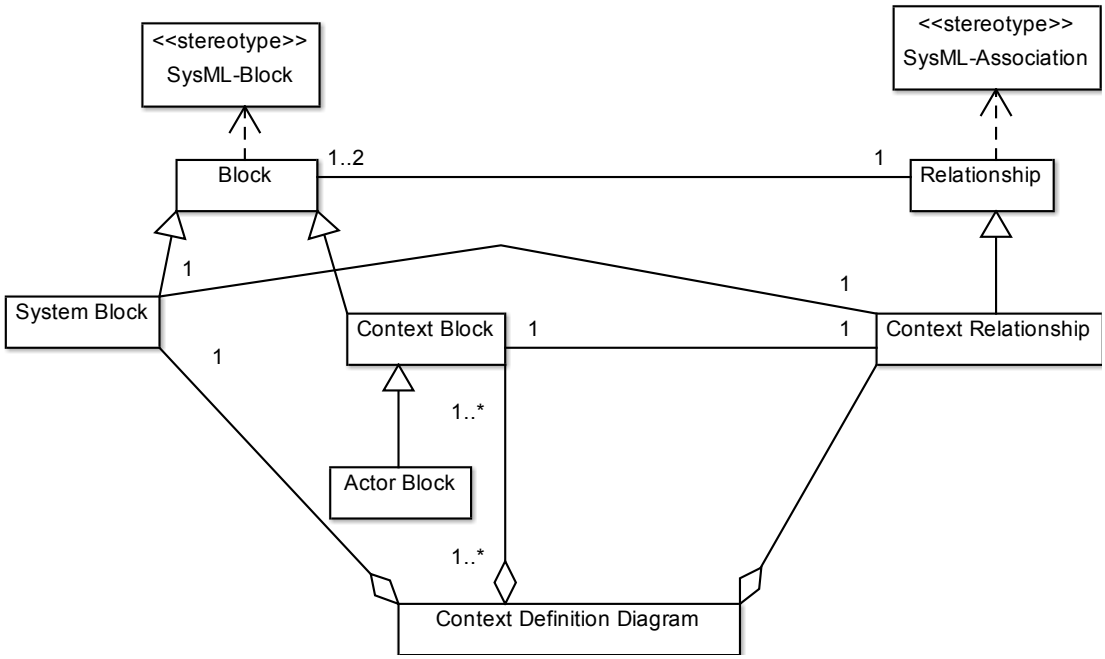

Figure 3: The product context metamodel

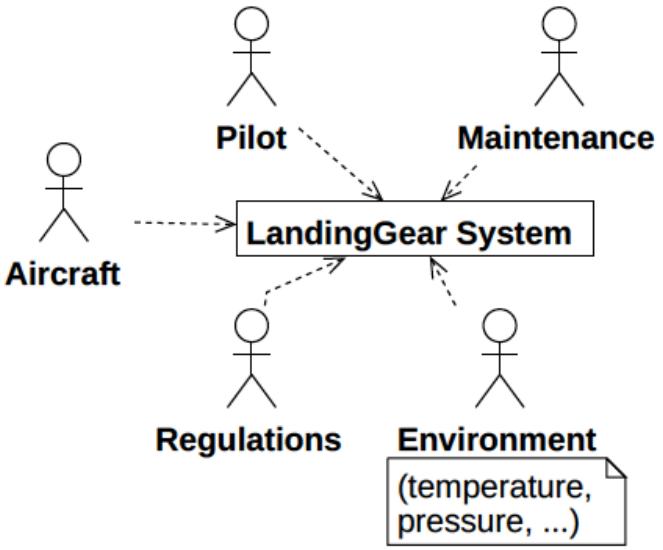

Figure 4: An example of a product context diagram

November 2016. However, the use of SysML/KAOS on the case study called SATURN, led us to extend the domain modeling language and make it more suitable for use in system modeling: the language has been adjusted to allow the definition of associations between associations and to support variable data items. This completes the definition of the domain modeling language: associations have been generalised into concepts and variability has been extended to individuals. These extensions are described in the article contained in Appendix C.

4 Conclusion

This document is devoted to the presentation of the domain modeling language. It is specified with a set of metamodels. It has been validated on several case studies :

- Specification of a transportation system for the city of Montréal ([FLF+19])
- Modeling of the Hybrid ERTMS/ETCS Level 3 Standard ([FFLM18b])

– Specification of the system controlling the level of water in a steam-boiler ([FFLM18a], [FFL+18])

– Specification of the rail communication protocol SATURN ([FLB+19])

– Specification of the airplanes localization system DVOR (THALES, confidential)

The semantics of domain models will be defined by translation rules into the B System language. This will be described in Deliverable D3.2.a.

References

[Ant96] A. I. Anton. Goal-based requirements analysis. In *Int. Conf. on Requirements Engineering*. IEEE Computer Society, 1996.

[Bro13] Manfred Broy. Domain modeling and domain engineering: Key tasks in requirements engineering. In J. Munch and K. Schmid, editors, *Perspectives on the Future of Software Engineering*, pages 15–30. Springer, 2013.

[BW14] Frédéric Boniol and Virginie Wiels. The landing gear system, ABZ Case Study. volume 433 of *Communications in Computer Information Science*. Springer, 2014.

[CNYM00] Lawrence Chung, Brian A. Nixon, Eric Yu, and John Mylopoulos. *Non-Functional Requirements in Software Engineering*. Kluwer Academic Publishers, 2000.

[DvLF93] Anne Dardenne, Axel van Lamsweerde, and Stephen Fickas. Goal-directed requirements acquisition. *Sci. Comput. Program.*, 20(1-2):3–50, 1993.

[FFL+18] Steve Jeffrey Tueno Fotso, Marc Frappier, Régine Laleau, Amel Mammar, and Michael Leuschel. Formalisation of sysml/kaos goal assignments with B system component decompositions. In Carlo A. Furia and Kirsten Winter, editors, *Integrated Formal Methods - 14th International Conference, IFM 2018, Maynooth, Ireland, September 5-7, 2018, Proceedings*, volume 11023 of *Lecture Notes in Computer Science*, pages 377–397. Springer, 2018.

[FFLM18a] Steve Jeffrey Tueno Fotso, Marc Frappier, Regine Laleau, and Amel Mammar. Back propagating B system updates on sysml/kaos domain models. In *23rd International Conference on Engineering of Complex Computer Systems, ICECCS 2018, Melbourne, Australia, December 12-14, 2018*, pages 160–169. IEEE Computer Society, 2018.

[FFLM18b] Steve Jeffrey Tueno Fotso, Marc Frappier, Régine Laleau, and Amel Mammar. Modeling the hybrid ERTMS/ETCS level 3 standard using a formal requirements engineering approach. In Michael J. Butler, Alexander Raschke, Thai Son Hoang, and Klaus Reichl, editors, *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 6th International Conference, ABZ 2018, Southampton, UK, June 5-8, 2018, Proceedings*, volume 10817 of *Lecture Notes in Computer Science*, pages 262–276. Springer, 2018.

[FLB+19] Steve Jeffrey Tueno Fotso, Régine Laleau, Héctor Ruíz Barradas, Marc Frappier, and Amel Mammar. A formal requirements modeling approach: Application to rail communication. In Marten van Sinderen and Leszek A. Maciaszek, editors, *Proceedings of the 14th International Conference on Software Technologies, ICSOFT 2019, Prague, Czech Republic, July 26-28, 2019*, pages 170–177. SciTePress, 2019.

[FLF+19] Steve Jeffrey Tueno Fotso, Régine Laleau, Marc Frappier, Amel Mammar, Francois Thibodeau, and Mama Nsangou Mouchili. Assessment of a formal requirements modeling approach on a transportation system. In Yamine Aït Ameur and Shengchao Qin, editors, *Formal Methods and Software Engineering - 21st International Conference on Formal Engineering Methods, ICFEM 2019, Shenzhen, China, November 5-9, 2019, Proceedings*, volume 11852 of *Lecture Notes in Computer Science*, pages 470–486. Springer, 2019.

[GS10] Christophe Gnaho and Farida Semmak. Une extension sysml pour l'ingénierie des exigences dirigée par les buts. In *Actes du XXVIIIème Congrès INFORSID, Marseille, France, 25-28 mai 2010*, pages 277–292, 2010.

[GS11] Christophe Gnaho and Farida Semmak. Une extension sysml pour l'ingénierie des exigences non fonctionnelles orientée but. *Ingénierie des Systèmes d'Information*, 16(1):9–32, 2011.

[GSL13] Christophe Gnaho, Farida Semmak, and Régine Laleau. Modeling the impact of non-functional requirements on functional requirements. In Jeffrey Parsons and Dickson K. W. Chiu, editors, *Advances in Conceptual Modeling - ER 2013 Workshops, LSAWM, MoBiD, RIGiM, SeCoGIS, WISM, DaSeM, SCME, and PhD Symposium, Hong Kong, China, November 11-13, 2013, Revised Selected Papers*, volume 8697 of *Lecture Notes in Computer Science*, pages 59–67. Springer, 2013.

[MC01] William E. McUmber and Betty H. C. Cheng. A general framework for formalizing UML with formal languages. In Hausi A. Müller, Mary Jean Harrold, and Wilhelm Schäfer, editors, *Proceedings of the 23rd International Conference on Software Engineering, ICSE 2001, 12-19 May 2001, Toronto, Ontario, Canada*, pages 433–442. IEEE Computer Society, 2001.

[ML16] Amel Mammar and Régine Laleau. On the use of domain and system knowledge modeling in goal-based event-b specifications. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part I*, volume 9952 of *Lecture Notes in Computer Science*, pages 325–339, 2016.

[NVLG14] Tuong Huan Nguyen, Bao Quoc Vo, Markus Lumpe, and John Grundy. KBRE: a framework for knowledge-based requirements engineering. *Software Quality Journal*, 22(1):87–119, 2014.

[OMG12] OMG. *SysML Specification. v1.3*. 2012.

[vL09] Axel van Lamsweerde. *Requirements Engineering - From System Goals to UML Models to Software Specifications*. Wiley, 2009.

[Yu97] E. Yu. Towards modeling and reasoning support for early-phase requirements engineering. In *Int. Conf. on Requirements Engineering*, pages 226–235. IEEE Computer Society, 1997.

Appendix A

Towards a goal-oriented approach for the modeling of time-relative requirements in critical systems

Christophe Gnaho[1,2]
[1] Laboratoire Algorithmique, Complexité et Logique
Université Paris Est
61 av du général de Gaulle, 94010 Créteil cedex
[2] Université Paris Descartes
45 rue des Saints-pères, 75006 Paris
christophe.gnaho@parisdescartes.fr

Farida Semmak[1], Regine Laleau[1]
[1] Laboratoire Algorithmique, Complexité et Logique
Université Paris Est
61 av du général de Gaulle, 94010 Créteil cedex
{semmak, laleau}@u-pec.fr

***Abstract*—Critical systems are used today in a variety of domains, such as transportation, health, aeronautics, etc. They are subject to complex temporal requirements, violation of which can put human lives at risk or damage assets and then very quickly lead to critical situations. Therefore, time-relative requirements and dependencies should be considered from the early requirements engineering phase because even on this phase, stakeholders are already able to decide about them. Even if the current Requirements Engineering (RE) approaches, and particularly Goal Oriented Requirements Engineering (GORE) approaches provide a high degree of flexibility, they do not seem to propose explicit constructs to deal with time-relative dimensions. We have proposed in previous works, SysML/KAOS, a GORE approach to manage functional and non-functional requirements in complex critical systems. While this approach has been used in a variety of research projects, it does not yet provide explicit representation of time and then limits possibilities for real-time systems modeling. This paper presents a part of our on-going research, which aims to contribute to this issue by introducing time semantics into SysML/KAOS Goal based concepts.**

Index Terms*—*Temporal and real time requirements, critical system, Non-Functional Requirements, Goal-oriented requirements modelling, dependencies FR-NFR

I. Introduction

Critical systems are used today in a variety of domains, such as transportation, health, aeronautics, etc. They are subject to complex temporal requirements, which violation can risk lives or damage assets and then very quickly lead to critical situations. Managing these requirements as secondary activity is not optimal; we argue that they should be considered from the early requirements engineering phase because even on this phase, stakeholders are already able to decide about temporal requirements and dependencies between them.

Two types of requirements relating to time are generally considered: temporal requirements and timed requirements. The first deals with the notion of time qualitatively while the second one allows quantitative reasonning about time. In this paper we will use the term Time-relative to encompass both types.

Different approaches [1][2][3] have been proposed for the elicitation and analysis of early requirements. Although providing a high degree of flexibility, these approaches do not seem to provide explicit constructs to deal with all the time-relative dimensions. KAOS method enables formalization of individual goals using LTL [4]. While it is essential for rigorous analysis, this practice generally requires significant investment and expertise. In addition, it is not explicitly address the mentioned time-relative issues.

We have proposed in previous works SysML/KAOS [5][6], a Goal Oriented Requirements Engieering (GORE) approach to manage functional and non-functional requirements in complex critical systems. Even if this approach has been used in a variety of research projects, it does not yet provide explicit representation of time and then limits possibilities for real-time systems modeling. Thus, we need to extend it expressiveness for specification of this kind of systems requirements. This paper represents a first step towards this objective; it presents a work in progress that aims at extended SysML/KAOS by introducing time semantics into Goal based concepts.

For the expression of time-relative concepts, we focused on user-friendly graphical notations along with constrained natural language instead of formal languages such as TCTL [7] and LTL [4]. Indeed, we argue that it would be more effective to provide notations that are intuitive enough to be used by stakeholders lacking formal languages training, while still providing a formal semantics that can be used as an input for formal verification methods. In order to reach this objective, we have chosen to base our work on the pattern-based approach proposed by Dwyer and al [8][9], which provide a catalogue of patterns for the description of formal requirements in user-friendly manner. However, The Dwyer patterns allow to reason about occurrence and order of events or states, but not explicitly about their timing. Consequently, we have adopted a subset of the patterns, which are useful for our purpose and extended them with timing concepts. The Landing Gear System Case (LGS) study [11] will be used to illustrate the presented concepts.

The remainder of the paper is structured as follows. The next section presents some background that the paper relies on.

Section 3 presents and discusses the extensions proposed to manage explicitly time-relative requirements. Section 4 discusses related work and finally Section 5 concludes the paper and gives an overview about future work.

II. Backgroud

This section briefly introduces some background, which the paper relies on. It respectively presents an overview of SysML/KAOS, summarizes the Dwyer patterns approach and gives a short description of the Landing Gear Case study that will be used to illustrate the concepts.

A. An overview of SysML/KAOS Approach

SysML/KAOS approach is based on KAOS [1] and the NFR Framework [3], two approaches largely recognized and used in requirements engineering over the past decade. It is founded on two main ideas:

- To integrates non-functional requirements much earlier, at the same level of abstraction than functional requirements; and emphasizing the impact of non-functional requirements on functional requirements.
- To take advantage of the contribution of SysML, such as easily relating requirements to specifications and easily providing tool support.

The main contribution of this approach is the analysis and modelling of the impact of non-functional goals on functional goals, which can be expressed in different manners. In the present state of our work, we have shown that non-functional goals may have an impact on the choices and decisions that are taken when refining functional goals and when transforming them into target systems. In addition, analysing non-functional goals can lead to the identification of new functional goals, which must be integrated with the existing functional goal hierarchy.

Therefore, this approach provides three main steps. In the first step, functional and non-functional requirements are specified in two separate goal models. Then, the impact of the non-functional requirements on functional ones are analyzed and described. In the last step, a final integrated goal model is obtained. Functional requirements are represented with concepts of the KAOS goal model and non-functional are represented with concepts inspired and adapted from NFR Framework. See [5][6] for more details.

B. Overview on property specification patterns

Property specification patterns were first introduced by Dwyer et al. [8][9] These patterns include a set of commonly occurring high-level specification abstractions for formalisms like LTL [4], CTL [10] or TCTL [7]. For example, the property specification ”Globally, S responds to P” is expressed as $AG(P \Rightarrow AF(S))$ in CTL or as $(P \Rightarrow \blacklozenge S)$ in LTL [4]. They enable stakeholders who are not familiar with such formalisms to read and write formal specifications. According to [9], 92% of 555 property specifications collected from different sources matched one of the patterns.

A property specification consists of a pattern, which describes what must occur and a scope, which describes when the pattern must hold.

As shown in Figure 1, Patterns are organized in hierarchy, based on their semantics, which distinguishes properties that deal with the occurrence and ordering of states/events during a system execution. We summarize below the most important patterns; a more detailed description of these patterns can be found in [8][9].

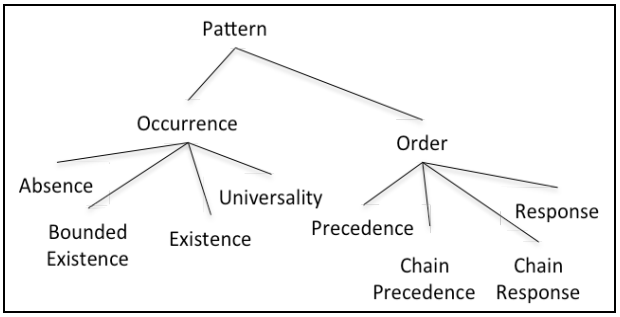

Fig. 1. Pattern hierarchy

The symbols P or Q represent given states/events.

Occurrence patterns

- *Absence*: P does never occur within a scope.
- *Universality*: P occurs throughout a scope.
- *Existence*: P must occur within a scope
- *Bounded Existence*: P must occur at least / exactly or at most k times within a scope.

Order patterns

- *Precedence*: P must always be preceded by Q within a scope
- *Response*: P must always be followed by Q within a scope
- *Chain Precedence / Chain Response*: A sequence P1, . . . „Pn must always be preceded /followed by a sequence Q1, . . . , Qm within a scope.

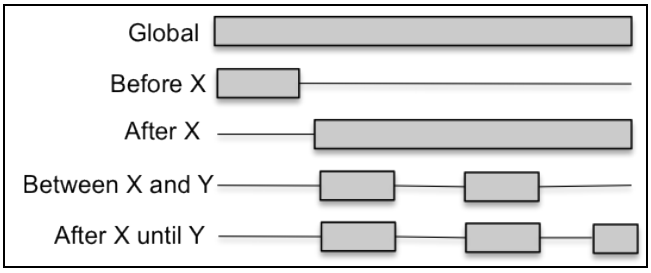

Fig. 2. Pattern scopes

Scopes (see Fig. 2) define, when the above patterns must hold:

- *Global*: the pattern must hold during the complete system execution.
- *Before*: the pattern must hold up to a given event X.
- *After*: the pattern must hold after the occurrence of a given event X.

- *Between*: the pattern must hold from the occurrence of a given X to the occurrence of a given Y
- *Until*: the same as ”between”, but the pattern must hold even if Y never occurs

With the above patterns, properties like ”An occurrence of event or state A must be followed by an occurrence of event or state B” can be expressed. However, They do not explicitly include timed properties like ”An occurrence of event A must be followed by an occurrence of event B within x time units”. In this work (see next section), we have selected the patterns that are relevant to our purpose and add some extensions in order to include timed issues, which are essential for real-time critical systems.

C. Landing Gear System (LGS) overview

The landing gear system [11] is in charge of maneuvering landing gears and associated doors. It is composed of 3 landing sets: front, left and right. Each landing set contains a door, a landing gear and associated hydraulic cylinders. This system is representative of critical embedded systems. The action to be done at each time depends on the state of all the physical devices and on their temporal behavior.

The architecture and the requirements of the system are presented in [11]. It is composed on three main parts: a mechanical part which contains all the mechanical devices and the three landing sets, a digital part including the control software, and a pilot interface.

The only human input to the system is the pilot handle: when pulled up it orders the gears to retract, and when pulled down it orders the gears to extend. The signal from the pilot handle is fed both to the replicated computer system and to the analogical switch. The purpose of the analogical switch is to protect the system against abnormal behavior of the digital part. In order to prevent inadvertent order to the electro-valves, the general electro-valve can be stimulated only if this switch is closed. A set of discrete sensors informs the digital part about the state of the equipment.

III. MODELING TIME-RELATIVE REQUIREMENTS WITH SYSML/KAOS

This section presents the main contribution of the paper. It describes the concepts introduced in SysML/KAOS meta-model to extend it expressiveness for the specification of goal models with time-relative dimension. Fig. 3 shows an extract of the extended meta-model. For space reason, we focus on the proposed extensions that are represented by the gray boxes; refer to [6] for more details on other concepts.

As shown in Fig. 3, both functional and non-functional requirements are represented as abstract goals, which are recursively refined into sub-goals, thanks to the AND/OR refinement mechanism. A goal that cannot be further refined is called elementary (functional or non-functional) goal.

We propose to specify time-relative goals as specialization of non-functional goals (see Fig. 3). This activity is mainly supported by a classification, which is described in the following section along with the associated concepts. These concepts are illustrated with examples from the LGS case study.

A. Specification of time-relative goals

As we said, the modeling of time-relative goal is mainly supported by our classification, which is built upon the property patterns introduced in Section 2. In this context, the main objective is to abstract the property patterns and apply them at goal level. We have considered both the sub-categories *order* and *occurrence,* which we have extended with real-time concepts.

More precisely, our main objective is twofold: to provide the ability to express goals regarding the temporal order in which they need to be achieved and to place time bound on the duration of the achievement of goals.

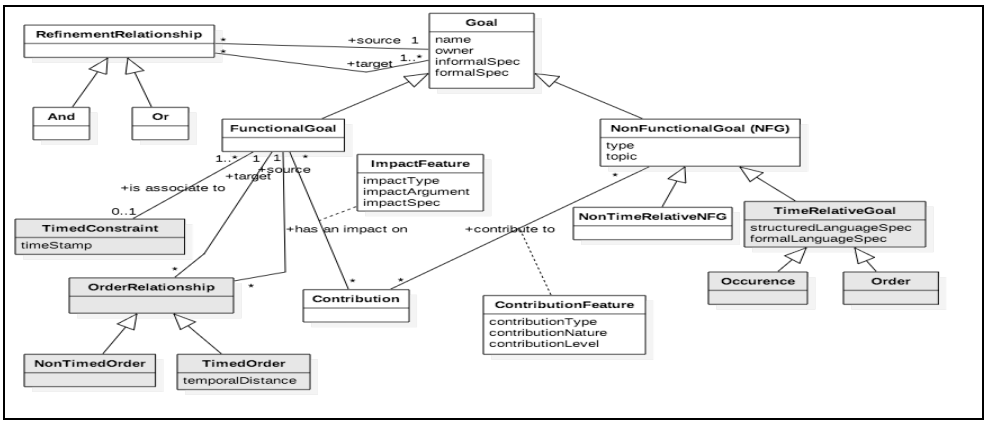

Fig. 3. An extract of the extended SysML/KAOS meta-model

Thanks to this work, time-relative goals can be easily expressed by the requirements engineer in structured natural language and can be potentially mapped into some formal languages for verification (LTL, CTL etc.). Finally, this work supports both the elicitation, refinement and operationalization steps.

1) Classification overview

As the other concepts have already been presented in Section 2 (note that *Untimed Response* and *Untimed Precedence* correspond respectively to *Response* and *Precedence in section 2*), we describe here only the proposed extensions, which are represented by the gray boxes in Fig. 4.

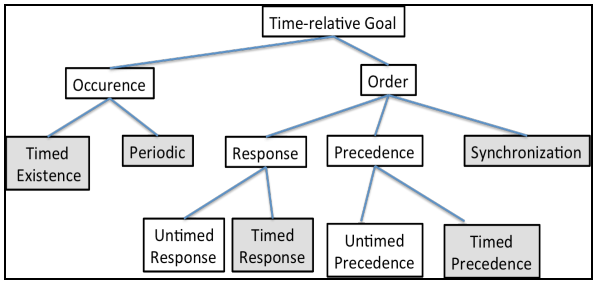

Fig. 4. Time-relative goal classification

The symbol P or Q represent a given state/event.

Occurrence sub-categories

- ***Periodic***: describes properties that address periodic occurrences. It denotes that P must occur recurrently every x time units. For example, *the computing module of the LGS must check the sensors every 20 milliseconds.*
- ***Timed existence***: denotes that starting from the current point of time, P must occur (at least/exactly/most) within x time-units. For instance, *the landing sets doors must be closed within a maximum of 10seconds.*

Order sub-categories

- *Timed response*: denotes that P must always be followed by Q but Q must occur within a given time span. For instance, *the gears will be locked down and the doors will be seen closed less than 15 seconds after the handle has been pushed*.
- *Timed precedence*: denotes that P must always be preceded by Q but after a delay. For example, *stimulations of the general electro-valve and of the maneuvering electro-valve must be separated by at least 200 milliseconds*
- *Synchronization*: specifies a relationship between two or more properties indicating that the behavior of the latter must be performed at the same time. For example *the simultaneity of the three landing gears during the outgoing sequence, and the retraction sequence.*

2) Structured natural language

Several formalisms (LTL [4], CTL [10], TCTL [7], Timed Automata [12]) have been proposed in order to specify time-relative properties. As we said, even if using these formalisms is essential for verification and validation this remains tedious and subject to error. Thus, it is necessary to provide a solution that hides the formal foundation to the requirements engineer. Thanks to the previous classification, the different categories of goals can be specified in a structured natural language and may be optionally mapped into formal languages. For instance, the Timed Response goal (Globally Q responds to P within x time units) is expressed in TCTL by $AG(P \Rightarrow AFx\ (Q\ ))$.

The structured natural language specification captures the scope (*globally*, *before*, *after*, *between*, or *after-until*) followed by the category (*timed existence, periodic, timed response, untimed response, synchronization, untimed precedence, timed precedence*). In order to help the requirements engineer, we provide a generic syntax expressed using BNF (Backus Naur Form) notation. An extract of this syntax is presented below.

```
<Time-Relative Goal> ::=<scope> ";" <specification>
<scope> ::="globally" | "before"<entity>| "after" <entity> |
"between" <entity> "and" <entity>
<specification>   ::=<entity>  <GoalCategory> <entity>
<timeInterval>
<entity> ::=state | event
```

An example from the Landing Gear System of such specification is "***Globally****; the gears* ***must*** *be locked* ***before*** *a maximum delay of 15 second* ***after*** *the handle position has been pushed down.*"

3) Time-relative goals fufillment

Thanks to the SysML/KAOS process, when all the abstract time-relative goals are refined into a set of elementary goals, we need to find and express solutions that satisfied them. For this purpose, the concept of contribution and the concept of impact defined in [6] are used. These concepts are summarized below.

The concept of *contribution* aims at describing the alternative solutions to satisfy elementary non-functional goals. It expresses the way by which an elementary non-functional goal could be achieved. A contribution is characterized by three main properties: *contributionType*, *contributionLevel* and *contributionNature.* The property *ContributionType* specifies whether the contribution is *positive* or *negative*. A positive (or negative) contribution helps (or prevents) to the satisfaction of an elementary non-functional goal. The property *ContributionLevel* allows us to associate to the type of contribution (positive or negative), a level that can range from very high to low. The property *ContributionNature* specifies whether the contribution is *explicit* or *induced*.

Finally, the contribution to the satisfaction of some time-relative goals may have an impact on functional goals. This

purpose is addressed thanks to the concept of *impact*. In the current state of our work, we have observed that the expression of this concept requires considering some ordering and timed constraints on functional goals. We argue that these constraints are relevant and should be explicitely specified in the goals models. For this purpose, we have introduced two additional concepts, which are presented in the next subsection.

B. Goal ordering and timed constraint modeling

The left-hand part of Figure 3 presents the two additional concepts: ordering of goal fulfillment and timed constraint.

1) Ordering of goal fulfillment

We consider that the parent goal is not just a set of AND/OR refined sub-goals but could be an ordered sequence. For this purpose, we introduce (see Fig. 3) the concept of *OrderRelationship* that connects goals with each other. This concept is specialized in two sub-types: *NonTimedOrder* and *TimedOrder*.

A *NonTimedOrder* relationship between two goals means that the satisfaction of the latter is not possible unless the former has been satisfied. A particular case of the *NonTimedOrder* relationship is the KAOS *Milestone refinement*, which consists of identifying the sub-goals as successive steps in time to satisfy the parent goal. A *TimedOrder* relationship between two goals means that satisfaction of the latter is possible within x time units after the former has been satisfied. Fig. 5 shows an example that illustrates these concepts. The goal *Extend landing gear* is refined thanks to the milestone refinement into four sub-goals : *Push command to Down*, *Open the doors*, *Lock down the gears* and *Close the doors*. A timed order relationship beetween the first and the fourth goal specifies that *The doors must be closed before a maximum delay of 15s after the handle command has been pushed down*. A timed order relationship is graphically represented in the model by an annotated link between the two goals.

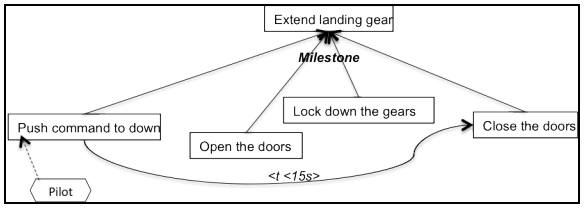

Fig. 5. An example of ordering of goal fulfillment

2) Timed Constraint modeling

Some functional goals can be associated to a timed constraint, meaning that their achievement is constrained by a quantified delay. This is captured in the meta-model of Fig. 3 by the concept *TimedConstraint*. For instance, "*the landing gears must be loeked down within a maximum delay of 5 seconds*". A *TimedConstraint* is caracterized by the property *timestamp* that represents the duration during which the constrained goal must be achieved. It can be expressed as *minumum duration, maximum duration or exactely time*. Figure 6 presents a goal model that illustrates this concept. Timed constrained goals are graphically represented by rectangles with pictograms.

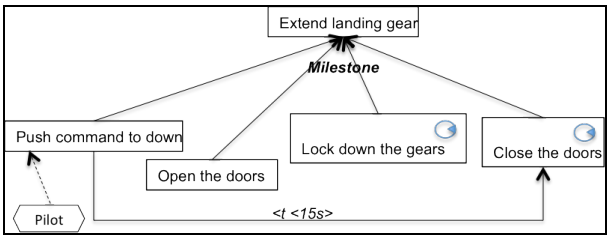

Fig. 6. An example of timed constraints

C. An illustrative example

This sub-section illustrates with an excerpt of the LGS case study, the SysML/KAOS process along with the concepts presented above.

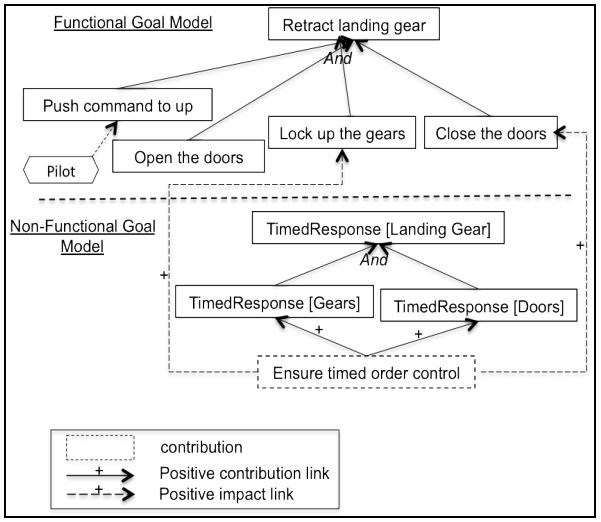

Fig. 7. Excerpt of the LDG functional and non-functional goal models

According to the first step of SysML/KAOS process, the functional and the non-functional goal models are constructed in parallel; the result is shown in Fig. 7.

The functional goal *Retract landing gear* is refined thanks to the *and refinement* into four sub-goals : *Push command to Up*, *Open the doors*, *Lock up the gears* and *Close the doors*. In the non-functional goal model, the time-relative goal *TimedResponse[Landing Gear]* is *and refined* into the following two sub-goals.

- *TimedResponse [Gears]* : "*Globally* the gears *must occcur* (locked up) *before a maximum delay of 5 seconds after* the handle position has been pushed up".
- *TimedResponse [Doors]* : "*Globally*, the doors *must occur* (close) *before a maximum delay of 10 seconds after* the gears locked up".

As shown in Fig. 7, the contribution *Ensure timed order control* represents an alternative way to contribute to *the* satisfaction of the two sub-goals. In addition, this contribution has an impact on the functional goals *Lock up the gears and Close the doors.* This impact should be reflected in the functional goal model that thereby needs some changes. The result is a new functional goal model that we call *integrated functional goal model*, presented in Fig. 8. The two goals *Lock up the gears* and *Close the doors* are associated to *timed constraints* that are graphically represented by rectangles with pictograms. The fulfillment of three goals (*Push command to up, lock up the gears and close the doors*) are ordered, which is graphically represented by the annotated links. Finally the *And refinement* is replaced by a *milestone refinement*.

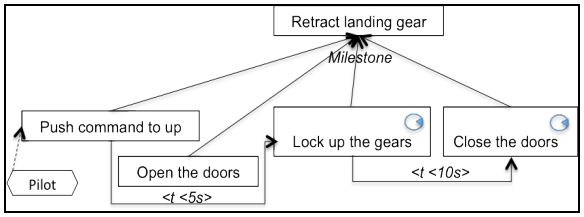

Fig. 8. Excerpt of the integrated LDG goal model

IV. Related work

The existing goal-oriented requirements engineering approaches have addressed interesting issues such as elicitation, dependency and prioritization [1][2][3]. A major benefit of them is their modeling languages for the specification and refinements of goal models. Even though these approaches offer a variety of concepts for modeling dependencies among goals and tasks, they have paid little attention to time-relative issues. Other works [13] investigate possibilities for extending UML meta-model for the specification of real-time systems. However, UML is more suitable to the late requirements phase than to the early requirements phase.

More recent works [14][15] propose some extensions to KAOS and I* to support the specification of temporal dependencies between goals. However, these works cannot be used to specify real-time requirements, since they focus exclusively on the qualitative aspect of time, they do not allow quantitative reasoning about time. The novelty of our work in comparison to other approaches is that we consider both temporal and timed requirements.

V. Conclusion and perspectives

In this paper, we have presented the concepts introduced in SysML/KAOS meta-model to extend it expressiveness for the specification of goal models with time-relative dimension. This work therefore represents a first step towards this objective. While it introduces and demonstrates the concepts, a number of works are ongoing. First, we are improving and completing the current result. Second, tool support is being developed.

References

[1] A. Dardenne, A. van Lamsweerde, S. Fickas, Goal-oriented Requirements Acquisition, Science of Computer Programming 20(1-2), 3–50 (1993)

[2] A. Fuxman, L. Liu, J. Mylopoulos, M. Pistore, M. Roveri, and P. Traverso, Specifying and analyzing early requirements in tropos. Requirements Engineering, 9(2): 132–150, 2004.

[3] L. Chung, B.A. Nixon, E. Yu, J. Mylopoulos, Non-functional Requirements in Software Engineering. Kluwer, Academic Publishers (2000)

[4] S. Chang, Z. Manna, and A. Pnueli, Characterization of Temporal Property Classes, in W.Kuich, editor, Proceedings of the 19th International Colloquium on Automata, Languages, and Programming, LNCS 623, Springer-Verlag, 1992, pp. 474-486.

[5] C. Gnaho, F. Semmak, R. Laleau. An overview of a SysML extension for goal-oriented NFR modelling: Poster paper**.** *Research Challenges in Information Science (RCIS)*, pages 1-2, 2013. IEEE.

[6] C. Gnaho, F. Semmak, R. Laleau, Modeling the Impact of Non-Functional Requirements on Functional Requirements, in volume 8697 of the Lecture Notes in Computer Science series, 2014

[7] T. A. Henzinger, X. Nicollin, J. Sifakis and S. Yovine, Symbolic Model Checking for Real-Time Systems, in: 7th. Symposium of Logics in Computer Science (1992), pp. 394–406.

[8] M. B Dwyer, G. S. Avrunin and J. C. Corbett, Property specification patterns for finite-state verification, in: FMSP '98: Proceedings of the second workshop on Formal methods in software practice (1998), pp. 7–15.

[9] M. B. Dwyer, G. S. Avrunin and J. C. Corbett, Patterns in property specifications for finite-state verification, in: Proc. of the 21st international conference on Software engineering (1999), pp. 411–420.

[10] F. Laroussinie, and P. Schnoebelen, Specification in CTL+Past for Verification in CTL, *Information and Computation*, 2000, pp. 236-263.

[11] F. Boniol, V. Wiels, The Landing Gear System Case Study, Volume 433 of the series Communications in Computer and Information Science pp 1-18

[12] R. Alur, D. L. Dill. 1994 A Theory of Timed Automata. In Theoretical Computer Science, vol. 126, 183-235

[13] L. Kabous. And W. Nebel, Modeling Hard Real Time Systems using UML: The OOHARTS Approach., UML'99, LNCS 1723(1999),339–355

[14] S. Liaskos and J. Mylopoulos. On Temporally Annotating Goal Models. In Proceedings of the 4th International i* Workshop (iStar'10). Hammamet, Tunisia. pp 62-66. 2010

[15] B. Nagel, C. Gerth, J. Post, G. Engels, Kaos4SOA - Extending KAOS Models with Temporal and Logical Dependencies. CAiSE Forum 2013: 9-16

Appendix B

Integrating Domain Knowledge in Formal Requirements Engineering

Steve TUENO, Régine LALEAU, Amel MAMMAR, Marc FRAPPIER

Abstract A means of building safe critical systems consists of formally modeling the requirements formulated by stakeholders and ensuring their consistency with respect to domain properties. This paper proposes a metamodel for a domain modeling language, using ontologies, based on *OWL* and *PLIB*. The language is part of the *SysML/KAOS* requirements engineering method, which also includes a goal modeling language. The formal semantics of SysML/KAOS models is represented, verified and validated using the *Event-B* method. Goal models provide machines and events of Event-B specifications, while domain models provide the corresponding structure. Our proposal is illustrated through a case study dealing with an excerpt of a *Cycab* localization component specification.

1 Introduction

Computer science is a relatively young science, but it does not prevent it from tackling extreme problems such as the implementation of critical and complex software systems. Such systems require a careful design and manipulation to ensure that they do not cause disasters. The literature on the subject is full of examples of disasters

Steve TUENO
Université Paris-Est Créteil, 94010, CRÉTEIL, France,
Université de Sherbrooke, Sherbrooke, QC J1K 2R1, Canada,
e-mail: steve.tuenofotso@univ-paris-est.fr

Régine LALEAU
Université Paris-Est Créteil, 94010, CRÉTEIL, France, e-mail: laleau@u-pec.fr

Amel MAMMAR
Télécom SudParis, 91000, Evry, France, e-mail: amel.mammar@telecom-sudparis.eu

Marc FRAPPIER
Université de Sherbrooke, Sherbrooke, QC J1K 2R1, Canada, e-mail: Marc.Frappier@usherbrooke.ca

that have occurred as a result of the neglect of this principle [17]. The purpose of the *ANR FORMOSE* project [4] is to design a formally-grounded, model-based requirements engineering method, for critical and complex systems, supported by an open-source environment. Modeling a system with this method requires the representation of its requirements as well as the properties of its application domain. This representation implicitly implies a semantics that must be defined explicitly through a formal method in order to be verified and validated and thus to prevent potential failures. The *SysML/KAOS* goal modeling language [12] focuses on the modeling of functional and non-functional requirements through a goal hierarchy. Furthermore, the study reported in [19] is interested in the explicit representation, of the semantics of SysML/KAOS goal models, with *Event-B* [1]. This paper complements the aforementioned studies with the definition of a domain modeling language. We first synthesize the body of knowledge related to the concrete representation of the semantics of SysML/KAOS goal models. Then, we analyse existing domain modeling approaches in requirements engineering and we describe and illustrate our domain modeling language. The illustration is performed on *TACOS* [3], a case study dealing with the specification of a localization software component that uses *GPS*, *Wi-Fi* and sensor technologies for the realtime localization of the *Cycab* vehicle [25], an autonomous ground transportation system designed to be robust and completely independent.

The remainder of this paper is structured as follows: Section 2 briefly describes Event-B and SysML/KAOS. Section 3 summarises existing work [19, 18] on the explicit representation of semantics of SysML/KAOS models. Follows a presentation, in Section 4, of the relevant state of the art on domain modeling in requirements engineering. Section 4 also defines our expectations regarding the domain modeling language to be developed and an evaluation of the main ontology modeling languages. In Section 5, we describe and illustrate our approach to model the domain of a system specified using the SysML/KAOS method. Finally, Section 6 reports our conclusions and discusses future work.

2 Background

In this section, we provide a brief overview of the Event-B formal method and of the SysML/KAOS requirements engineering method.

2.1 Event-B

Event-B [1] is a formal method created by *J. R. Abrial* for *system modeling*. It is used to incrementally build a specification of a system that preserves a set of properties expressed through invariants. Event-B is mostly used to model closed systems: the modeling of the system is accompanied by that of its environment and of

all interactions likely to occur between them. An Event-B model includes static parts called *contexts* and dynamic parts called *machines*. Contexts contain declarations of abstract and enumerated sets, constants, axioms and theorems. Machines contain variables, invariants and events. Moreover, a machine can access the definitions of a context. Each event has a *guard* and an *action*. The guard is a condition that must be satisfied for the event to be triggered and the action describes updates of state variables. The system specification can be constructed using stepwise refinement, by refining machines. Proof obligations are defined to prove invariant preservation by events (invariant has to be true at any system state), event feasibility, convergence and machine refinement [1]. We use *B System* [8], a variant of Event-B proposed by *ClearSy*, an industrial partner in the *FORMOSE* project, in its integrated development environment *Atelier B* [5]. *B System* and *Event-B* share the same semantics but are syntactically different [28].

2.2 SysML/KAOS

SysML/KAOS [13] is a requirements engineering method based on *SysML* [14] and KAOS [17]. *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. The *KAOS* method includes five models of which the two main ones are: i) the **goal model** for the representation of requirements to be satisfied by the system and of expectations with regards to the environment through a hierarchy of goals, and ii) the **object model** which uses the *UML* class diagram for the representation of domain vocabulary.

The goal hierarchy is built through a succession of goal refinements using different operators: *AND*, *OR* and *MILESTONE*. An *AND refinement* decomposes a goal into subgoals, and all of them must be achieved to realise the parent goal. 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. A *MILESTONE refinement* is a variant of the AND refinement which allows the definition of an achievement order between goals.

Despite of its goal expressiveness, KAOS offers no mechanism to maintain a traceability between requirements and design deliverables, making it difficult to validate them against the needs formulated. In addition, the expression of domain properties and constraints is limited by the expressiveness of UML class diagrams, which is considered insufficient by our industrial partners [4], regarding the complexity and the criticality of the systems of interest. Therefore, for goal modeling, SysML/KAOS combines the traceability features provided by SysML with goal expressiveness provided by KAOS.

A functional goal, under SysML/KAOS, describes the *expected behaviour* of the system once a certain condition holds [18] : *[if* ***CurrentCondition*** *then] sooner-or-later* ***TargetCondition***. SysML/KAOS allows the definition of a functional goal

without specifying a *CurrentCondition*. In this case, the expected behaviour can be observed from any system state.

Figure 1 represents a goal diagram, of the *Cycab System* localization component, focused on the purpose of vehicle localization.

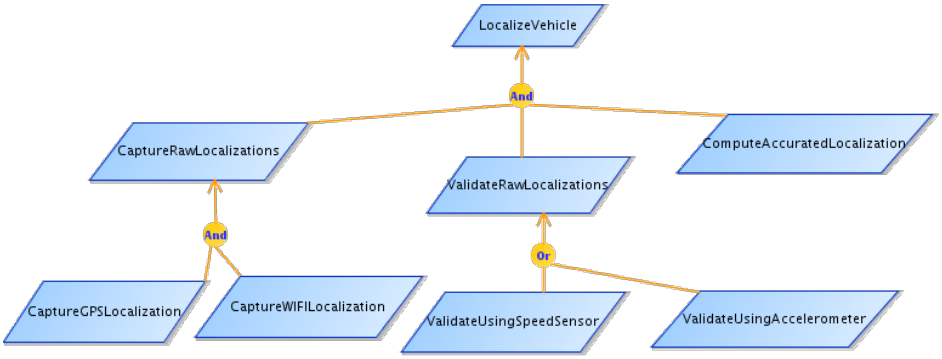

Fig. 1 Excerpt from the localization component goal diagram

To achieve the root goal, which is the localization of the vehicle (**LocalizeVehicle**), raw localizations must be captured from vehicle sub components (**CaptureRawLocalizations**) which can be GPS (**CaptureGPSlocalization**) or Wi-Fi (**CaptureWIFILocalization**), be validated using a vehicle sensor (**ValidateRawlocalizations**) which has to be either a speed sensor (**ValidateUsingSpeedSensor**) or an accelerometer (**ValidateUsingAccelerometer**) and used to compute the vehicle's accurate localization (**ComputeAccuratedlocalization**).

3 Expression of the Semantics of SysML/KAOS Models in Event-B

3.1 Semantics of Goal Models

The formalisation of SysML/KAOS goal models is detailed in [19]. Each refinement level of a goal diagram gives an Event-B machine. Each goal gives an event. The semantics of refinements links between goals is explicited using proof obligations that complement classic proof obligations for **invariant preservation** and for **event actions feasibility** defined in [1]. The other classic Event-B proof obligations are not relevant for our purpose [19]. Regarding the added proof obligations, they depend on

the refinement pattern used. For an abstract goal G and two concrete goals G_1 and G_2 : [1]

- For an *AND* refinement, the proof obligations are
  - $G_1\_Guard \Rightarrow G\_Guard$
  - $G_2\_Guard \Rightarrow G\_Guard$
  - $(G_1\_Post \wedge G_2\_Post) \Rightarrow G\_Post$
- For an *OR* refinement, they are
  - $G_1\_Guard \Rightarrow G\_Guard$
  - $G_2\_Guard \Rightarrow G\_Guard$
  - $G_1\_Post \Rightarrow G\_Post$
  - $G_2\_Post \Rightarrow G\_Post$
  - $G_1\_Post \Rightarrow \neg G_2\_Guard$
  - $G_2\_Post \Rightarrow \neg G_1\_Guard$
- For a *MILESTONE* refinement, they are
  - $G_1\_Guard \Rightarrow G\_Guard$
  - $G_2\_Post \Rightarrow G\_Post$
  - $\Box(G1\_Post \Rightarrow \Diamond G2\_Guard)$ (each system state, corresponding to the post condition of G_1, must be followed, at least once in the future, by a system state enabling G_2)

Figure 2 and 3 represent the *B System* specifications obtained respectively from the root level of the goal diagram of Fig. 1 and from its first refinement level. It appears that the structure of the system and the body of events must be manually completed. The structure of a system is represented within a *B System* specification by constants constrained by properties and variables constrained by an invariant. The objective of our study is to automatically derive this structure from a rigorous modeling of the domain of the system.

Proof obligations related to the *AND* refinement link between the root and the first refinement levels are:

$$CaptureRawlocalizations\_Guard \Rightarrow LocalizeVehicle\_Guard \quad (1)$$

$$ValidateRawlocalizations\_Guard \Rightarrow LocalizeVehicle\_Guard \quad (2)$$

$$ComputeAccuratedlocalization\_Guard \Rightarrow LocalizeVehicle\_Guard \quad (3)$$

$$CaptureRawlocalizations\_PostCondition \wedge ValidateRawlocalizations\_PostCondition \wedge ComputeAccuratedlocalization\_PostCondition \Rightarrow LocalizeVehicle\_PostCondition \quad (4)$$

[1] For an event `G`, `G_Guard` represents the guards of G and `G_Post` represents the post condition of its actions.

SYSTEM
localizationComponent
SETS
CONSTANTS
PROPERTIES
VARIABLES
INVARIANT
INITIALISATION
EVENTS
LocalizeVehicle=
BEGIN
// localization of the vehicle
END
END

Fig. 2 Formalisation of the root level of the goal diagram of Fig. 1

REFINEMENT
localizationComponentRef1
REFINES
localizationComponent
SETS
CONSTANTS
PROPERTIES
VARIABLES
INVARIANT
INITIALISATION
EVENTS
CaptureRawlocalizations=
BEGIN
// capture raw localizations
END;
ValidateRawlocalizations=
BEGIN
// validate raw localizations
END;
ComputeAccuratelocalization =
BEGIN
// compute vehicle accurate localization
END
END

Fig. 3 Formalisation of the first refinement level of the goal diagram of Fig. 1

3.2 Towards an Event-B Expression of the Semantics of Domain Models

A domain model is a conceptual model capturing the topics related to a specific problem domain [7]. The main difference between requirements and domain models is that domain models are independent of stakeholders. They must conform to the operational context of the system. In [6], *a domain description primarily specifies*

semantic entities of the domain intrinsics, semantic entities of support technologies already "in" the domain, semantic entities of management and organisation domain entities, syntactic and semantic of domain rules and regulations, syntactic and semantic of domain scripts and semantic aspects of human domain behaviour. In [24], Pierra asserts that a domain model can be defined as a set of categories represented as classes, their properties and their logical relationships. Modeling the domain of a system consists in giving a representation of the set of concepts that the system will be called upon to manipulate and the set of properties and constraints associated with them.

A first attempt at modeling domains within SysML/KAOS is achieved in [18]. Domain modeling involves *UML* class diagrams, *UML* object diagrams and ontologies. The case study presented reveals the use of ontologies for the representation of domain knowledge; the model obtained is the *domain model*. Furthermore, UML object and class diagrams are used to represent the system structure and constraints in a model known as the *structural model* which must conforms to the domain model. A set of rules is proposed for the translation of some domain model elements into Event-B specifications. However, the approach uses *UML* diagrams which are graphical representations that lack a formal semantics [20] and expressivity [21]. Moreover, the use of several languages is an extra source of complexity.

4 State of the Art on Domain Modeling in Requirements Engineering

4.1 Existing Domain Modeling Approaches

In *KAOS* [17], the domain of a system is specified with an *object model* using *UML* class diagrams. An object within this model can be (1) an *entity* if it exists independently of the others and does not influence the state of any other object, (2) an *association* if it links other objects on which it depends, (3) an *agent* if it actively influences the system state by acting on other objects or (4) an *event* if its existence is instantaneous, appearing to impulse an update of the system state. This approach, which is essentially graphic and semi-formal, as argued in [20], is difficult to exploit in case of critical systems [21].

In [10], Devedzic proposes to model the knowledge of the domain through either formulae of first-order logic or ontologies. He considers an ontology as a more structured and extensible representation of domain knowledge.

In [16], domain models are built around *concepts* and *relationships*: each definition of a domain model consists of an assertion linking two instances of *Concept* through an instance of *Relationship*. A categorisation is proposed for concepts and relationships: a concept can be a function, an object, a constraint, an actor, a platform, a quality or an ambiguity, while a relationship can be a performative or a symmetry, reflexivity or transitivity relation. However, the proposed metamodel is missing some

relevant domain entities such as datasets, predicates to express domain constraints and relation cardinalities. Moreover, it does not propose modularisation mechanisms between domain models.

In [21], ontologies are used not only to represent domain knowledge, but also to model and analyze requirements. The proposed methodology is called *knowledge-based requirements engineering (KBRE)* and is mainly used for detection and processing of inconsistencies, conflicts and redundancies among requirements. In spite of the fact that *KBRE* proposes to model domain knowledge with ontologies, the proposal focuses on the representation of requirements. A similar approach called *GOORE* is proposed in [27].

In [9], Dermeval *et al.* are interested in a systematic review of the literature related to usages of ontologies in requirements engineering. They end up describing ontologies as a standard form of formal representation of concepts within a domain, as well as of relationships between those concepts.

These approaches suggest that ontologies are relevant for modeling the domains of systems.

4.2 A Study of Ontology Modeling Languages

An ontology can be defined as a formal model representing concepts that can be grouped into categories through generalisation/specialisation relations, their instances, constraints and properties as well as relations existing between them. Ontology modeling languages can be grouped into two categories: *Closed World Assumption (CWA)* for those considering that any fact that cannot be deduced from what is declared in the ontology is false and *Open World Assumption (OWA)* for those considering that any fact can be true unless its falsity can be deduced from what is declared in the ontology. As [2], we consider that accurate modeling of the knowledge of engineering domains, to which we are interested, must be done under the *CWA* assumption. Indeed, this assumption improves the formal validation of the consistency of system's specifications with respect to domain properties. Moreover, systems of interest to us are so critical that no assertion should be assumed to be true until consensus is reached on its veracity. Similarly, we also advocate *strong typing* [2] because our domain models must be translatable to Event-B specifications.

Several ontology modeling languages exist. The main ones are ***OWL (Ontology Web Language)*** [26], ***PLIB (Part LIBrary)*** [23] and ***F-Logic (Frame Logic)*** [15]. A summary of the similarities and differences between these languages is presented through Table 1:

- *PLIB*, *OWL* and *F-Logic* implement modularisation mechanisms between ontologies. *PLIB* supports partial import: a class of an ontology *A* can extend a class of an ontology *B* and explicitly specify the properties it wishes to inherit. Moreover, if nothing is specified, no property will be imported. On the other hand, *OWL* and

Table 1 Comparative table of the three main ontology modeling languages

Characteristics	**OWL**	**PLIB**	**F-Logic**
Modularity	total	partial	total
CWA vs OWA	OWA	**CWA**	**CWA**
Inheritance	multiple	**simple**	multiple
Typing	weak	**strong** (any element belongs to one and only one type)	weak
Expressivity	**strong**	weak	weak
Contextualization of a property (parameterized attributes)	-	**+**	**+**
Different views for an element	-	**+**	-
Graphic representation	**+**	-	-
Domain Knowledge (static vs dynamic)	static	static	static

F-Logic use the total import: when an ontology *A* refers to an ontology *B*, all the elements of *B* are accessible within *A*.

- *PLIB* and *F-Logic* use the *CWA* assumption for constraint verification, *OWL* uses the *OWA* assumption.
- *OWL* and *F-Logic* implement the multiple inheritance and instantiation. *PLIB* implements the simple inheritance and instantiation. On the other hand, with the ***is_case_of*** relation, a *PLIB* class can be *a case of* several other classes, each class bringing some specific properties.
- *PLIB* and *F-Logic* allow the definition of parameterized attributes using context parameters, which is not possible with *OWL*.
- *PLIB* allows the association of several representations or view points with a concept, which is not possible with neither *OWL* nor *F-Logic*.
- The knowledge modeled using *OWL*, *PLIB* and *F-Logic* is always considered static because there is no distinguishing mechanism. It is for example impossible to specify that the localization of a vehicle can change dynamically while its brand cannot.

As stated in [30], *all the studied languages emphasize more on modeling static domain knowledge*. None of these languages allows to specify that a knowledge described must remain unchanged or that it is likely to be updated. Moreover, none of the languages fully meet our requirements. For instance, *OWL* assumes the *OWA* assumption and *PLIB* is weakly expressive. The most aligned are *OWL* and *PLIB*.

5 Our Approach for Domain Modeling

We have chosen to represent domain knowledge using ontologies since they are semantically richer and therefore allow a more explicit representation of domain characteristics. Thus, in this Section, we propose a metamodel, based on that of *OWL*

and *PLIB* while filling their shortcomings, for the representation of the domain of a system whose requirements are captured using the SysML/KAOS method. Our language makes the *Unique Name Assumption (UNA)* [2] : the name of an element is sufficient to uniquely identify it among all the others within a domain model. Furthermore, our metamodel is designed to allow the specification of knowledges that are likely to evolve over time.

5.1 Presentation

We present through Figures 4, 5, 6 and 7 the main part of the metamodel associated with our domain modeling approach, knowing that yellow elements are those having an equivalent in the *OWL* metamodel and that red ones are those that we have either inserted or customized. Furthermore, some constraints and associations, such as the *parentConcept* association, have been extracted from the *PLIB* metamodel. Due to space consideration, we will not highlight all the elements and constraints of the metamodel.

5.1.1 Concepts and Individuals, Data Sets and Data Values

Domain models are built around the notion of Concept which represents a group of individuals sharing common characteristics (Fig. 4). A *concept* can be *variable* (*isVariable=true*) when the set of its individuals is likely to be updated through addition or deletion of individuals. Otherwise, it is *constant* (*isVariable=false*). A concept can be associated with another one, known as its parent concept, through the *parentConcept* association, from which it inherits properties. As a result, any individual of the child concept is also an individual of the parent concept.

An instance of DataSet is used to group instances of DataValue having the same type (Fig. 5). Default datasets are `INTEGER`, `NATURAL` for positive integers, `FLOAT`, `STRING` or `BOOL` for booleans. 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.

5.1.2 Relations and Attributes

Class Relation is used to capture links between concepts (Fig. 6) and Attribute links between concepts and data sets (Fig. 7). A relation (Fig. 6) or an attribute (Fig. 7) 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*. The association between a relation and a concept is characterized by a *cardinality*: DomainCardinality and RangeCardinality (Fig. 6). Each instance of DomainCardinality (respectively RangeCardinality) makes it possible to define, for an instance of

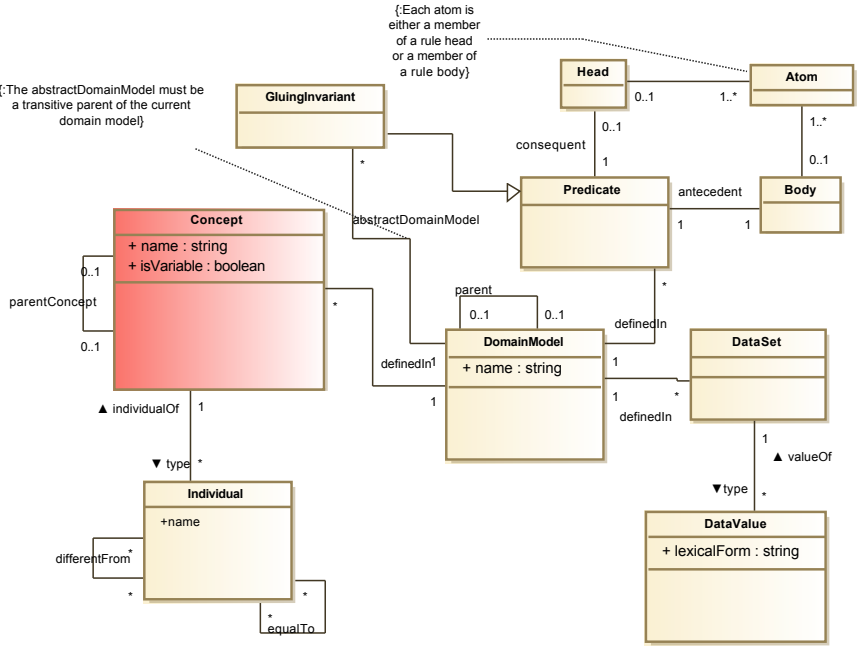

Fig. 4 First part of the metamodel associated with the domain modeling language

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 : $(minCardinality \geq 0) \wedge (maxCardinality = \infty \vee maxCardinality \geq minCardinality)$, knowing that if $maxCardinality = \infty$, then there is no maximum limit. 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 (Fig. 6) : *transitive* (*isTransitive*, default *false*), *symmetrical* (*isSymmetric*, default *false*), *asymmetrical* (*isASymmetric*, default *false*), *reflexive* (*isReflexive*, default *false*) or *irreflexive* (*isIrreflexive*, default *false*). It is said to be *transitive* (*isTransitive=true*) when the relation of an individual `x` with an individual `y` which is in turn in relation to `z` results in the relation of `x` and `z`. It is said to be *symmetric* when the relation between an individual `x` and an individual `y` results in the relation of `y` to `x`. It is said to be *asymmetric* when the relation of an individual `x` with an individual `y` has the consequence of preventing a possible relation between `y` and `x`, with the assumption that $x \neq y$. It is said to be *reflexive* when every individual of the domain is in relation with itself. It is finally

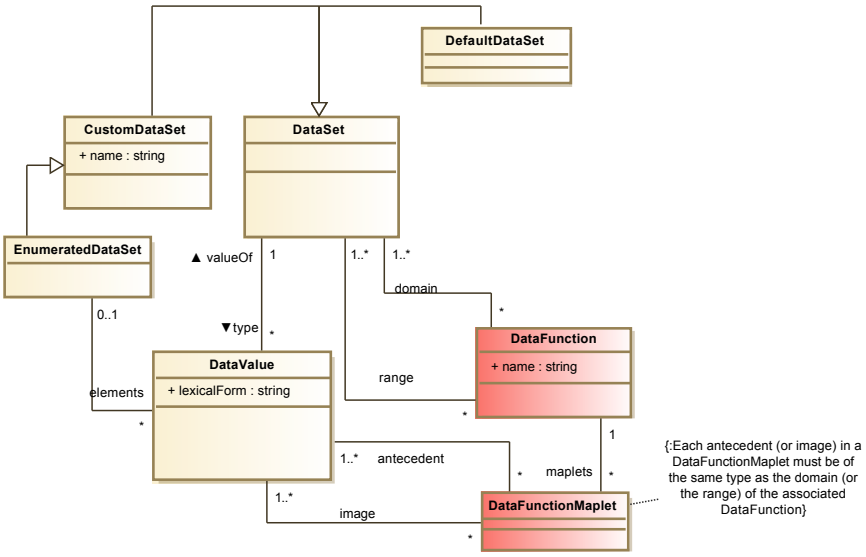

Fig. 5 Fourth part of the metamodel associated with the domain modeling language

said to be *irreflexive* when it does not authorize any connection of an individual of the domain with itself. Moreover, an attribute can be *functional* (*isFunctional*, default *true*) if it associates to each individual of the domain one and only one data value of the range.

5.1.3 Functions and Predicates

Class DataFunction (Fig. 5) makes it possible to define operations which allow to determine data values at the output of a set of processes on some input data values. At each tuple of data values of the domain, the `data function` assigns a tuple of data values of the range, and this assignement cannot be changed dynamically. ***Example***: We can define an instance of DataFunction named `multiply` to produce, given two integers `x` and `y`, the individual of `INTEGER` representing $x * y$. On the other side, class Predicate (Fig. 4) is used to represent constraints between different elements of the domain model as *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 *typing atom* is used to define the type of a term : ConceptAtom for individuals and DataSetAtom for data values (Fig. 8). An *association atom* is used to define associations between terms : RelationshipAtom for the connection of two terms through an instance of Relation, AttributeAtom for the connection of two terms through an instance of Attribute and DataFunctionAtom for the connection of terms through an instance of DataFunction (Fig. 8). For each

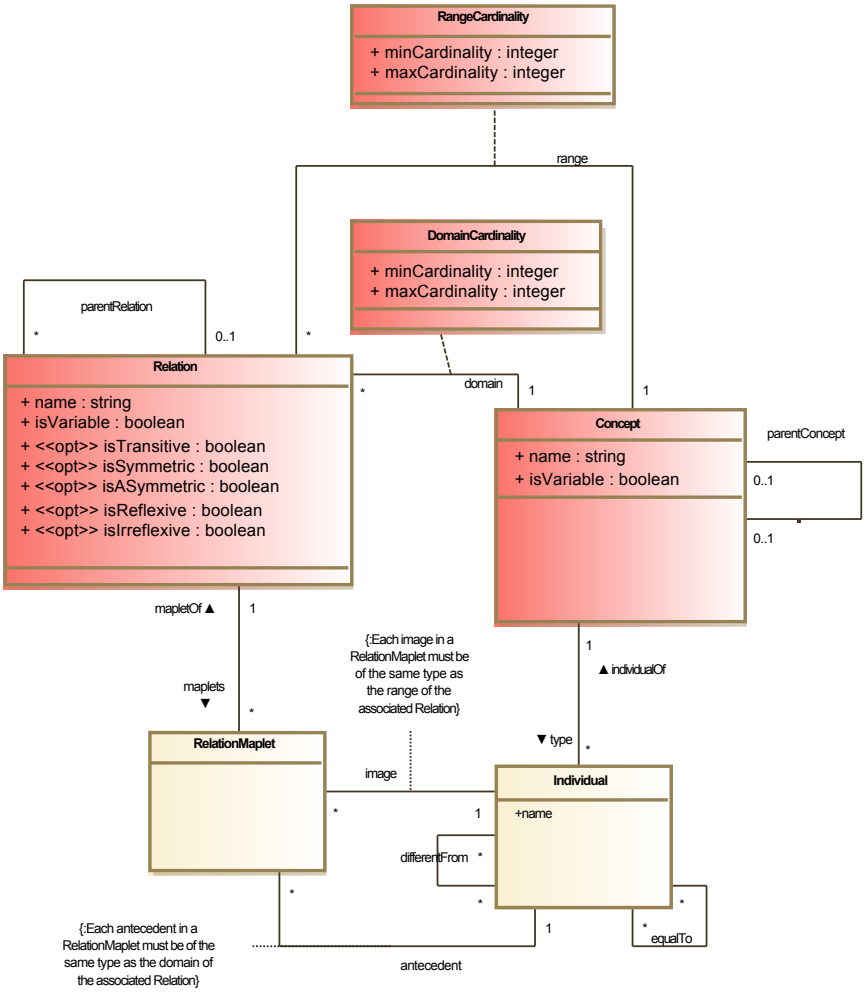

Fig. 6 Second part of the metamodel associated with the domain modeling language

case, the types of terms must correspond to the domains/ranges of the considered link. A *comparison atom* is used to define comparison relationships between terms : EqualityAtom for equality and InequalityAtom for difference (Fig. 8). 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 (Fig. 8). For example, an arithmetic constraint between several integers.

Predicates can be used to *parameterize* relations or attributes in order to define dependent associations. For example, knowing that the resistance of a material

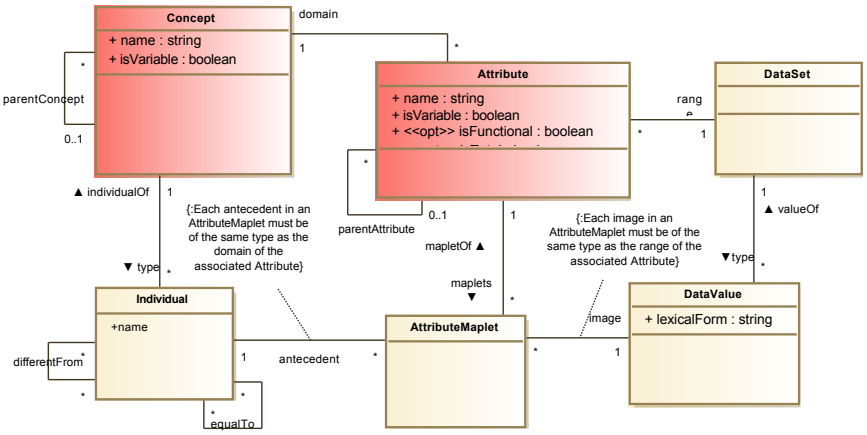

Fig. 7 Third part of the metamodel associated with the domain modeling language

depends on the temperature of the medium, resistance and temperature attributes are dependent.

5.1.4 Domain Model and 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 (Fig. 4). This allows the child domain model to access and extend some elements defined in the parent domain model. It should be noted that the parent domain model must be associated with the refinement level of the SysML/KAOS goal diagram directly above the refinement level to which the child domain model is associated.

In order to be able to be used in the setting up of large complex systems, SysML/KAOS allows the refinement of a leaf of a goal diagram in another diagram having this goal as root. For example, in Figure 9, the goal *G3*, which is a leaf of the first goal diagram, is the root of the second one. When this happens, we associate to the most abstract level of the new goal diagram the domain model associated with the most concrete level of the previous goal diagram as represented in Figure 9: *Domain Model 2*, which is the domain model associated to the most concrete level of the first diagram, is also the domain model associated to the root of the second one.

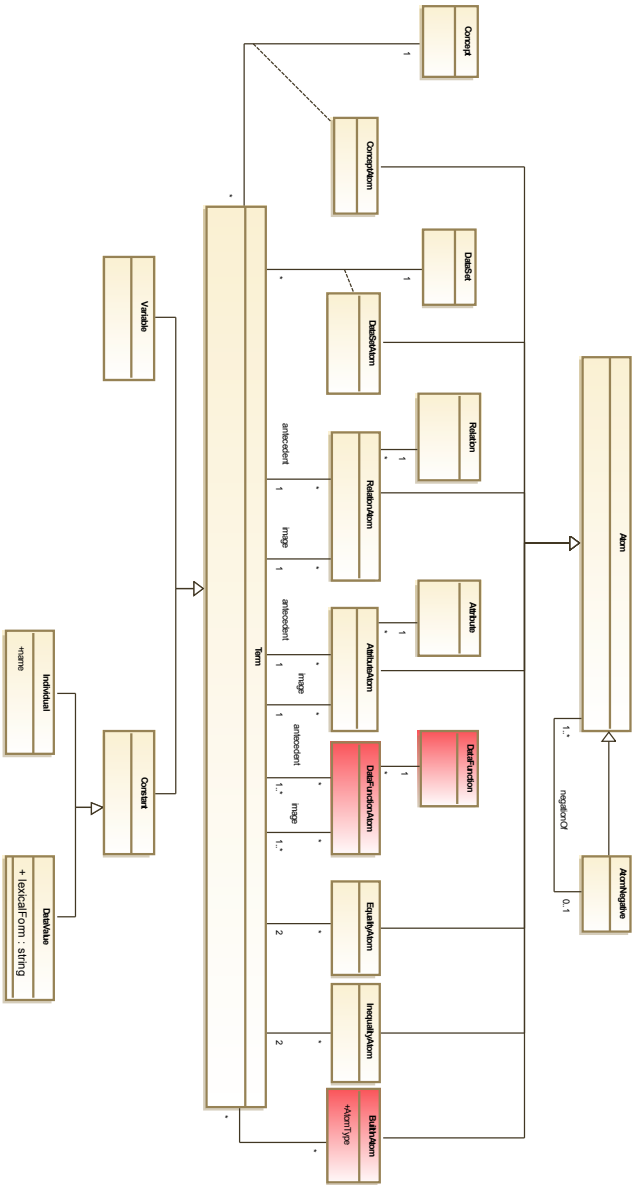

Fig. 8 Fifth part of the metamodel associated with the domain modeling language

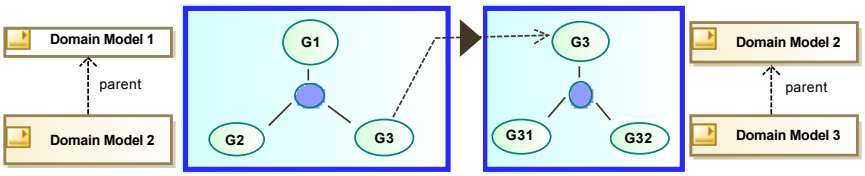

Fig. 9 Management of the partitioning of a SysML/KAOS goal model

5.2 Illustration

We have identified two graphical syntaxes for the representation of ontologies : the syntax proposed by *OntoGraph* [11] and the syntax proposed by *OWLGred* [29]. The *OntoGraph* syntax is the one used in [18]. Unfortunately, it does not allow the representation of some domain model elements such as attributes or cardinalities. For our illustration, we have thus decided to use the *OWLGred* syntax. For readability purposes, we have decided to represent the *isVariable* property only when it is set to *true* and to remove optional characteristics representation.

Figures 10, 11 and 12 represent respectively the domain model associated with the root level of the goal diagram of Fig. 1 (*localization_component_0*), that associated with the first refinement level (*localization_component_1*) and that associated with the second one (*localization_component_2*).

5.2.1 Ontology Associated with the Root Level

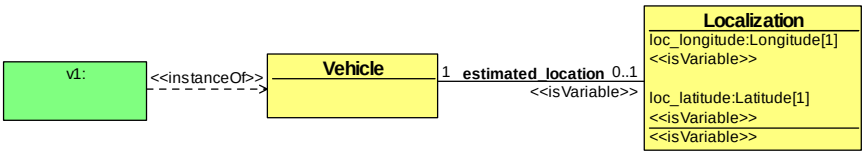

Fig. 10 ***localization_component_0***: ontology associated with the root level of the goal diagram of Fig. 1

In ontology *localization_component_0* (Fig. 10), a vehicle is modeled as an instance of Concept named `Vehicle` and its localization is represented through an instance of Concept named `Localization`. Since it is possible to dynamically add or remove vehicle localizations, the property *isVariable* of `Localization` is set to *true*, which is represented by the stereotype *«isVariable»*. Since the system is designed to control a single vehicle, it is not possible to dynamically add new ones. The involved vehicle is thus modeled as an instance of Individual named `v1` having `Vehicle` as *type*. `Localization` is the *domain* of two attributes :

the latitude modeled as an instance of Attribute named `loc_latitude` and the longitude modeled as an instance of Attribute named `loc_longitude`. Attribute `loc_latitude` has, as range, an instance of CustomDataSet named `Latitude` and `loc_longitude` an instance of CustomDataSet named `Longitude`. Since it is possible to dynamically change the localization of a vehicle, the property *isVariable* of `loc_latitude` and that of `loc_longitude` are set to *true*, which is represented by the stereotype *«isVariable»*. The association between an individual of `Vehicle` and an individual of `localization` is represented through an instance of Relation named `estimated_location`. Its associated instance of DomainCardinality has *1* as *minCardinality* and *maxCardinality*, and its associated instance of RangeCardinality has *0* as *minCardinality* and *1* as *maxCardinality*.

5.2.2 Ontology Associated with the First Refinement Level

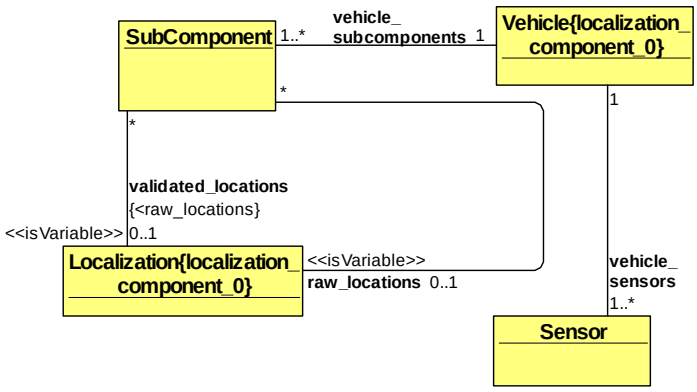

Fig. 11 ***localization_component_1***: ontology associated with the first refinement level of the goal diagram of Fig. 1

Ontology *localization_component_1* (Fig. 11) has ontology *localization_component_0* (Fig. 10) as parent and defines new concepts and relations. Each reused element is annotated with *localization_component_0*, the parent domain model name. `SubComponent`, which is an instance of Concept, is introduced to represent sub components of a vehicle. Each instance of Individual of *type* `SubComponent` associates the vehicle with a *raw location*. `Sensor`, which is also an instance of Concept is introduced to represent vehicle sensors used to validate the raw locations. Raw locations which are validated through sensors are called validated locations and are used to compute the vehicle estimated location. Each vehicle has at least one sub component and one sensor.

5.2.3 Ontology Associated with the Second Refinement Level

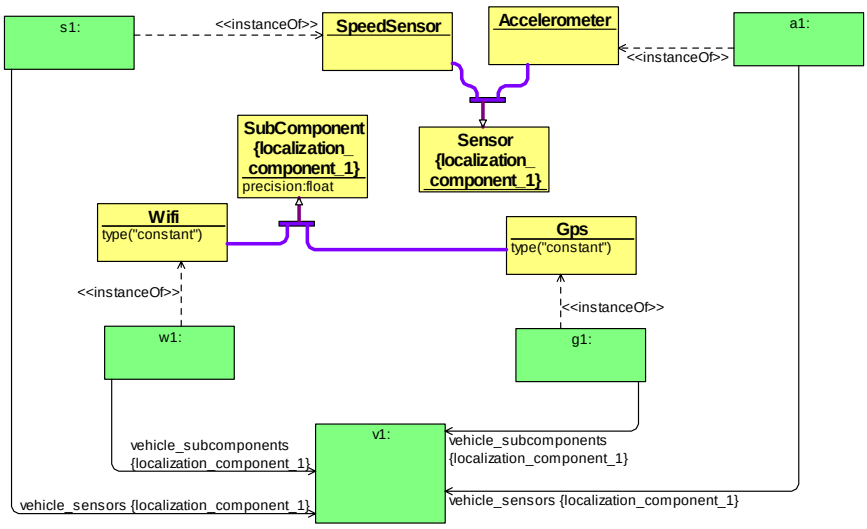

Fig. 12 ***localization_component_2***: ontology associated with the second refinement level of the goal diagram of Fig. 1

Ontology *localization_component_2* (Fig. 12) has ontology *localization_component_1* (Fig. 11) as parent.This third abstraction level represents child concepts of `SubComponent` and `Sensor`. A subcomponent is either a GPS, represented through an instance of Concept named `Gps`, or a Wi-Fi, represented through an instance of Concept named `Wifi`. A sensor is either an accelerometer, represented through an instance of Concept named `Accelerometer`, or a speed sensor, represented through an instance of Concept named `SpeedSensor`. Finally, `v1` is associated to an instance of Individual of *type* `Gps` named `g1` and to an instance of Individual of *type* `Wifi` named `w1` through `vehicle_subcomponents`, an instance of Relation introduced in *localization_component_1*. It is also associated to a speed sensor called `s1` and to an accelerometer called `a1`.

The constraint *"a GPS is more precise than a Wi-Fi"* is translated into an instance of Predicate represented through formula 5 : If an instance of Term, named `x`, having `Wifi` as its *type*, has `px` as its *precision* and an instance of Term, named `y`, having `Gps` as its *type*, has `py` as its *precision*, then $py > px$.

$$greaterThan(?py, ?px) \leftarrow Wifi(?x) \wedge precision(?x, ?px) \wedge Gps(?y) \wedge precision(?y, ?py) \quad (5)$$

6 Conclusion

In this paper, we have firstly presented the explicitness of SysML/KAOS goal models semantics in Event-B. Then, we have drawn up the state of the art related to domain modeling in requirements engineering. After positioning ourselves as to the existing, we have presented our domain modeling approach consisting in representing domain characteristics using an ontology modeling language for which a metamodel has been defined. Our approach has been illustrated through a case study dealing with the specification of the localization component of a *Cycab* vehicle.

Work in progress is aimed at developing mechanisms for the explicitness of the semantics of domain models, constructed using our domain modeling language, in Event-B. We are also working on integrating the language within the open-source platform *Openflexo* [22] which federates the various contributions of *FORMOSE* project partners [4].

Acknowledgements 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. Ameur, Y.A., Baron, M., Bellatreche, L., Jean, S., Sardet, E.: Ontologies in engineering: the OntoDB/OntoQL platform. Soft Comput. **21**(2), 369–389 (2017)
3. ANR-06-SETIN-017: TACOS ANR project (2017). URL `http://tacos.loria.fr/`
4. ANR-14-CE28-0009: Formose ANR project (2017). URL `http://formose.lacl.fr/`
5. Atelier, B.: the industrial tool to efficiently deploy the b method. URL: http://www. atelierb. eu/index-en. php (access date 22.03. 2015) (2008)
6. Bjørner, D., Eir, A.: Compositionality: Ontology and mereology of domains. In: Concurrency, Compositionality, and Correctness, Essays in Honor of Willem-Paul de Roever, *Lecture Notes in Computer Science*, vol. 5930, pp. 22–59. Springer (2010)
7. Broy, M.: Domain Modeling and Domain Engineering: Key Tasks in Requirements Engineering, pp. 15–30. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
8. ClearSy: Atelier B: B System (2014). URL `http://ajhurst.org/~ajh/teaching/ClearSy-Industrial_Use_of_B.pdf`
9. Dermeval, D., Vilela, J., Bittencourt, I.I., Castro, J., Isotani, S., da S. Brito, P.H., Silva, A.: Applications of ontologies in requirements engineering: a systematic review of the literature. Requir. Eng. **21**(4), 405–437 (2016)
10. Devedzic, V.: Knowledge modeling - state of the art. Integrated Computer-Aided Engineering **8**(3), 257–281 (2001)
11. Falconer, S.: Protégé - ontograph (2010). URL `http://protegewiki.stanford.edu/wiki/OntoGraf`
12. 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)
13. Gnaho, C., Semmak, F., Laleau, R.: Modeling the impact of non-functional requirements on functional requirements. pp. 59–67. Springer (2014)
14. Hause, M., et al.: The SysML modelling language. In: Fifteenth European Systems Engineering Conference, vol. 9. Citeseer (2006)

15. Kifer, M., Lausen, G.: F-logic: A higher-order language for reasoning about objects, inheritance, and scheme. In: Proceedings of the 1989 ACM SIGMOD, pp. 134–146. ACM Press (1989)
16. Kitamura, M., Hasegawa, R., Kaiya, H., Saeki, M.: An integrated tool for supporting ontology driven requirements elicitation. In: J. Filipe, B. Shishkov, M. Helfert (eds.) ICSOFT 2007, Volume SE, pp. 73–80. INSTICC Press (2007)
17. van Lamsweerde, A.: Requirements Engineering - From System Goals to UML Models to Software Specifications. Wiley (2009)
18. Mammar, A., Laleau, R.: On the use of domain and system knowledge modeling in goal-based Event-B specifications. In: ISoLA 2016, *Lecture Notes in Computer Science*, vol. 9952, pp. 325–339 (2016)
19. 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)
20. McUmber, W.E., Cheng, B.H.C.: A general framework for formalizing UML with formal languages. In: ICSE 2001, pp. 433–442. IEEE Computer Society (2001)
21. Nguyen, T.H., Vo, B.Q., Lumpe, M., Grundy, J.: KBRE: a framework for knowledge-based requirements engineering. Software Quality Journal **22**(1), 87–119 (2014)
22. Openflexo: Openflexo project (2015). URL `http://www.openflexo.org`
23. Pierra, G.: The PLIB ontology-based approach to data integration. In: IFIP 18th World Computer Congress, *IFIP*, vol. 156, pp. 13–18. Kluwer/Springer (2004)
24. Pierra, G.: Context representation in domain ontologies and its use for semantic integration of data. J. Data Semantics **10**, 174–211 (2008)
25. Sekhavat, S., Valadez, J.H.: The Cycab robot: a differentially flat system. In: IROS 2000, pp. 312–317. IEEE (2000)
26. Sengupta, K., Hitzler, P.: Web ontology language (OWL). In: Encyclopedia of Social Network Analysis and Mining, pp. 2374–2378 (2014)
27. Shibaoka, M., Kaiya, H., Saeki, M.: GOORE : Goal-oriented and ontology driven requirements elicitation method. In: ER 2007 Workshops, *Lecture Notes in Computer Science*, vol. 4802, pp. 225–234. Springer (2007)
28. Tueno, S., Mammar, A., Laleau, R., Frappier, M.: Event-B Expression and Validation of Translation Rules Between SysML/KAOS Domain Models and B System Specifications. Springer proceedings of 6th International ABZ Conference, 2018
29. UL, I.: Owlgred home (2017). URL `http://owlgred.lumii.lv/`
30. Zong-yong, L., Zhi-xu, W., Ai-hui, Z., Yong, X.: The domain ontology and domain rules based requirements model checking. International Journal of Software Engineering and Its Applications **1**(1), 89–100 (2007)

Appendix C

The Generic SysML/KAOS Domain Metamodel

Steve Jeffrey Tueno Fotso and Marc Frappier

GRIL, Université de Sherbrooke, Sherbrooke, QC J1K 2R1, Canada

Régine Laleau

LACL, Université Paris-Est Créteil, 94010, CRÉTEIL, France

Amel Mammar

SAMOVAR-CNRS, Télécom SudParis, 91000, Evry, France

Hector Ruiz Barradas

ClearSy System Engineering, 13100, Aix-en-Provence, France

Abstract

This paper is related to the generalised/generic version of the SysML/KAOS domain metamodel and on translation rules between the new domain models and *B System* specifications.

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

1. Background

1.1. Event-B and B System

Event-B [1] is an industrial-strength formal method for *system modeling*. It is used to incrementally construct a system specification, using refinement, and to prove useful properties. *B System* is an *Event-B* syntactic variant proposed by *ClearSy*, an industrial partner in the *FORMOSE* project [2], and supported by *Atelier B* [3]. *Event-B* and *B System* have the same semantics defined by proof obligations [1].

Figure 1 is a metamodel of the *B System* language restricted to concepts that are relevant to us. A *B System* specification consists of components (instances of Component). Each component can be either a system or a refinement and it may define static or dynamic elements. A refinement is a component which refines another one in order to access the elements defined in it and to reuse them for new constructions. Constants, abstract and enumerated sets, and their properties, constitute the static part. The dynamic part includes the representation of the system state using variables constrained through invariants and initialised through initialisation actions. Properties and invariants can be categorised as instances of LogicFormula. Variables can be involved only in invariants. In our case, it is sufficient to consider that logic formulas are successions of operands in relation through operators. Thus, an instance of LogicFormula references its operators (instances of Operator) and its operands that may be instances of Variable, Constant, Set or SetItem.

1.2. SysML/KAOS Goal Modeling

1.2.1. Presentation

SysML/KAOS [4, 5] is a requirements engineering method which combines the traceability provided by *SysML* [6] with goal expressiveness provided by KAOS [7]. It allows the representation of requirements to be satisfied by a system and of expectations with regards to the environment through a hierarchy of goals. The goal 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. An ***OR refinement*** decomposes a goal into subgoals such that the achievement of only one of them is sufficient for the achievement of the parent goal.

Email addresses: Steve.Jeffrey.Tueno.Fotso@USherbrooke.ca, Marc.Frappier@USherbrooke.ca (Steve Jeffrey Tueno Fotso and Marc Frappier), laleau@u-pec.fr (Régine Laleau), amel.mammar@telecom-sudparis.eu (Amel Mammar), hector.ruiz-barradas@clearsy.com (Hector Ruiz Barradas)

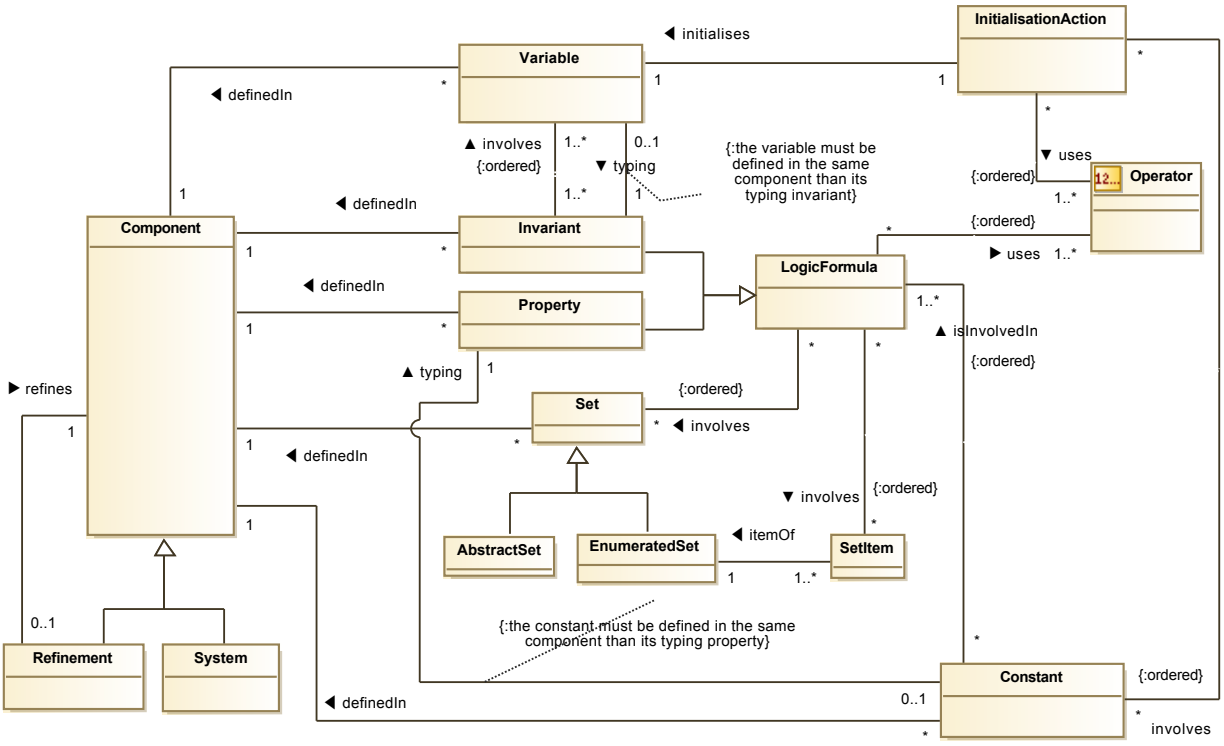

Figure 1: Metamodel of the *B System* specification language

For this work, the case study focuses on a communication protocol called *SATURN* proposed by *ClearSy*. SATURN relies on exchanges of communication frames between different agents connected through a bus. This case study is restricted to input/output agents. Input agents provide boolean data. Each input data undergoes a boolean transformation and the result is made available to output agents.

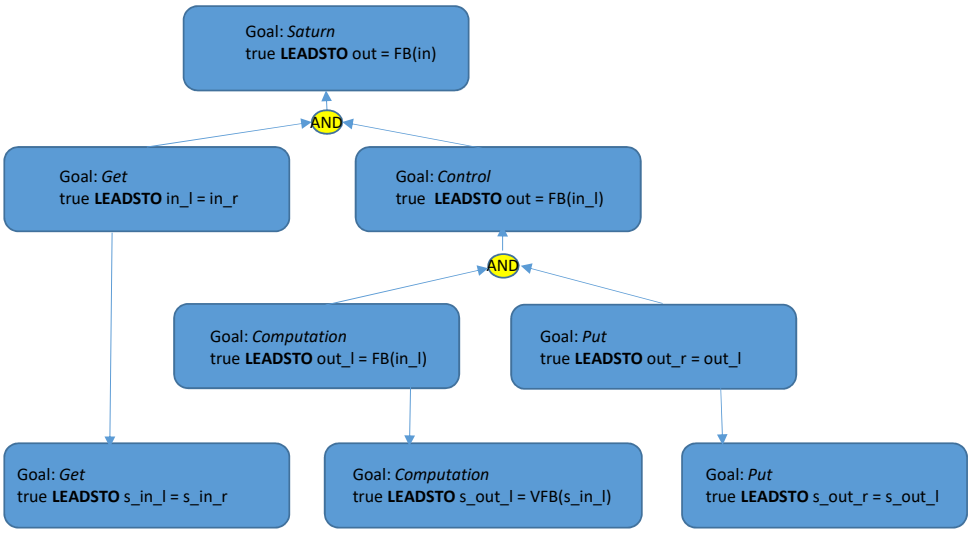

Figure 2: Excerpt from SATURN system goal diagram

Figure 2 is an excerpt from the SysML/KAOS goal diagram representing the functional goals of SATURN. The main purpose of the system is to transform data provided by input agents (`in`) and make the result (`out=FB(in)`) available to

output agents. The purpose gives the root goal *Saturn* of the goal diagram of Fig 2. However, goal *Saturn* disregards input reads and result writes. The *AND* operator is used just after to introduce, at the first refinement level, a goal *Get* for input data acquisition from input agents. Term `in_r` designates the data available within input agents and term `in_l` designates the input data used to compute the output data. Similarly, the second refinement level introduces a goal *Put* to make the result `out_l` available to output agents (`out_r` represents the data received by output agents). The third refinement level refines goals defined within the second refinement level to take into account multiplicities of input and output agents. Thus, input data acquisition generates a boolean array `s_in_l` instead of `in_l`, computation becomes a transformation between arrays `s_out_l = VFB(s_in_l)` and result delivery transfers the content of array `s_out_l` to output agents.

In addition, *SysML/KAOS* includes a domain modeling language which combines the expressiveness of *OWL* [8] and the constraints of *PLIB* [9].

1.3. SysML/KAOS Domain Modeling

1.3.1. Presentation

Domain models in SysML/KAOS are represented using ontologies. These ontologies are expressed using the SysML/KAOS domain modeling language [10, 11], built based on *OWL* [8] and *PLIB* [9], two well-known and complementary ontology modeling formalisms.

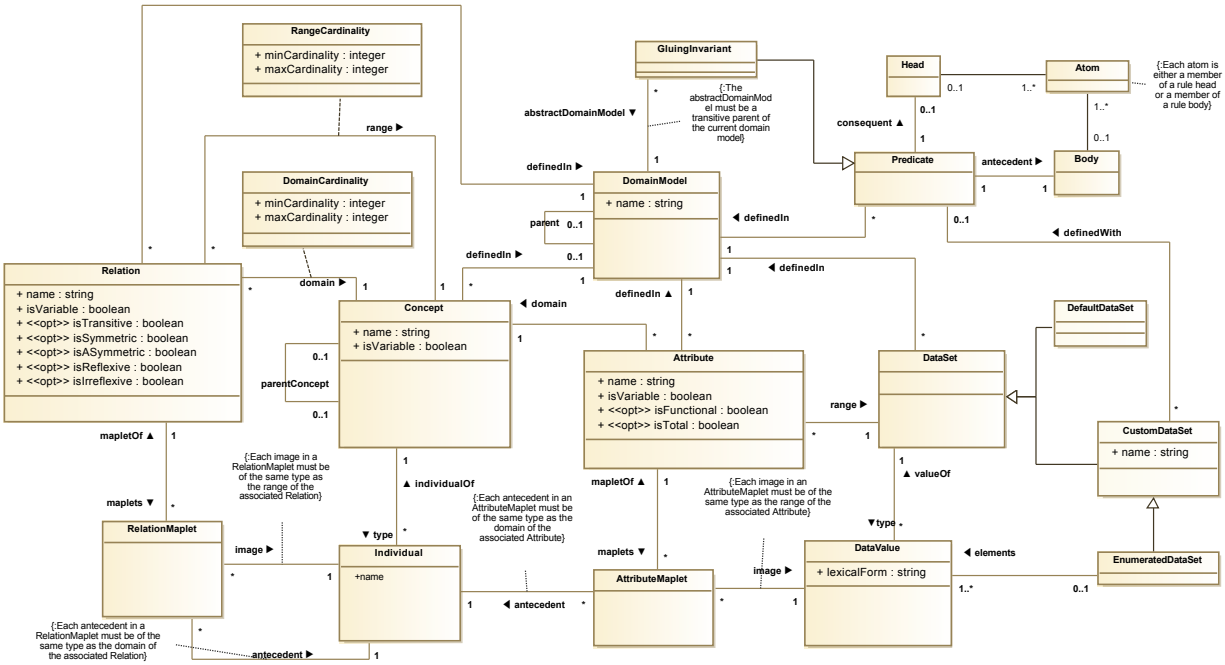

Figure 3: Excerpt from the metamodel associated with the SysML/KAOS domain modeling language

Figure 3 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. *Concepts* (instances of Concept) designate collections of *individuals* (instances of Individual) 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*).

Relations (instances of Relation) are used to capture links between concepts, and *attributes* (instances of Attribute) capture links between concepts and *data sets* (instances of DataSet). *Relation maplets* (instances of RelationMaplet) capture associations between individuals through relations and *attribute maplets* (instances of AttributeMaplet) play the same role for attributes.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*. The variability of an association (relation, attribute) is related to the ability to add or remove maplets. Each *domain cardinality* (instance of DomainCardinality) makes it

possible to define, for a relation `re`, the minimum and maximum limits of the number of individuals of the domain of `re` that can be put in relation with one individual of the range of `re`. In addition, the *range cardinality* (instance of RangeCardinality)) of `re` is used to define similar bounds for the number of individuals of the range of `re`.

Predicates (instances of Predicate) are 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 a *custom data set* (instance of CustomDataSet), and defined with a predicate. *Gluing invariants* (instances of GluingInvariant), specialisations of predicates, are used to represent links between data defined within a domain model and those appearing in more abstract domain models, transitively linked to it through the *parent* association. They capture relationships between abstract and concrete data during refinement and are used to discharge proof obligations.

1.3.2. Illustration and Shortcomings

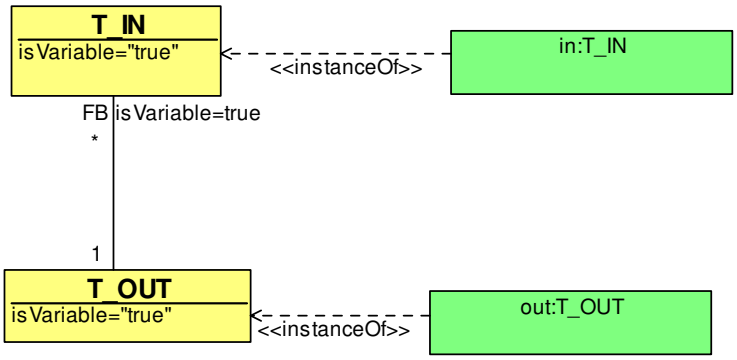

Figure 4: ***Saturn_1***: ontology associated with the root level of the goal diagram of Fig. 2

Figure 4 is an attempt to represent the domain model associated with the root level of the goal diagram of Fig. 2 using the SysML/KAOS domain modeling language previously described. It is illustrated using the syntax proposed by *OWLGred* [12] and, for readability purposes, we have decided to hide the representation of optional characteristics. It should be noted that the *individualOf* association is illustrated, through *OWLGred*, as a stereotyped link with the tag *«instanceOf»*.

The type of input data is modeled as a concept `T_IN` defining an individual `in` which represents the input data. Similarly, the type of output data is modeled as a concept `T_OUT` defining an individual `out` which represents the output data. The computation function `FB` is modeled as a functional relation from `T_IN` to `T_OUT`.

The first difficulty we encountered is related to the changeability of domain entities. In fact, the states of input and output data change dynamically. In domain model of Fig. 4, a workaround consisted in considering that concepts `T_IN` and `T_OUT` and relation `FB` are variables. Thus, going from a system state where `out1 = FB(in1)` to a system state where `out2 = FB(in2)` is feasible and goes through: (1) withdrawal of maplet `in1` $\mapsto$ `out1` from `FB`; (2) withdrawal of individual `in1` from `T_IN`; (3) withdrawal of individual `out1` from `T_OUT`; (4) addition of individual `in2` in `T_IN`; (5) addition of individual `out2` in `T_OUT`; and (6) addition of maplet `in2` $\mapsto$ `out2` in `FB`. However, this representation does not conform to SATURN's design. Indeed, from a conceptual point of view: (1) the input data type must be constant (corresponds to the set of n-tuples of Booleans[1]); (2) the output data type must be constant (corresponds to the set of m-tuples of Booleans[2]); (3) the computation function `FB` is hard-coded and is therefore constant. What should change are individuals representing the input and output data. It is thus necessary to be able to model variable individuals: individual which can dynamically take any value in a given concept. A similar need appears for relations with relation maplets, attributes with attribute maplets and data sets with data values.

Another difficulty has been encountered related to multiplicities of input and output agents (domain model associated with the third refinement level of the goal diagram of Fig. 2). Indeed, the array that represents input data needs to be modeled by a relation, ditto for the array that represents output data. Thus, the computation function needs to be

[1] When considering n input agents
[2] When considering m output agents

modeled by a relation for which the domain and the range are relations, which is impossible with the current definition of the SysML/KAOS domain modeling language.

The *SATURN* case study also revealed the need to be able to:

- define domain and range cardinalities for attributes;
- define a named maplet (instance of RelationMaplet or AttributeMaplet) with or without antecedent and image;
- define an initial value for a variable individual, maplet or data value;
- define associations between data sets and maplets between data values;
- refine a concept with an association or a data set[3];
- refine an individual with a maplet or a data value.

We have therefore identified the need to build a generalisation of the metamodel of Fig. 3 which enriches the expressiveness of the SysML/KAOS domain modeling language while preserving the fundamental constraints identified in [10, 11]. A major contribution of this new metamodel is that it federates notions of concept, data set, attribute and relation as well as notions of individual, maplet and data value that have always been considered distinct by ontology modeling languages. Additional constraints are defined to preserve the formal semantics of the language and to ensure unambiguous transformation of any domain model to a *B System* specification.

2. The New SysML/KAOS Domain Modeling Language

2.1. Presentation

Figure 5 is an excerpt from the updated SysML/KAOS domain metamodel.

2.1.1. Description

Domain models are also associated with levels of refinement of the SysML/KAOS goal model. *Concepts* (instances of Concept) designate collections of *individuals* (instances of Individual) 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*). In addition, a concept can be an enumeration (*isEnumeration=TRUE*) if all its individuals are defined within the domain model. It should be noted that an individual can be *variable* (*isVariable=TRUE*) if it is introduced to represent a system state variable: it can represent different individuals at different system states. Otherwise, it is *constant* (*isVariable=FALSE*).

Associations (instances of Association) are concepts used to capture links between concepts. *Maplet individuals* (instances of MapletIndividual) capture associations between individuals through associations. Each named maplet individual can reference an antecedent and an image. When the maplet individual is unnamed, the antecedent and the image must be specified. The variability of an association is related to the ability to add or remove maplets. Each *domain cardinality* (instance of DomainCardinality) makes it possible to define, for an association `re`, the minimum and maximum limits of the number of individuals of the domain of `re` that can be put in relation with one individual of the range of `re`. In addition, the *range cardinality* (instance of RangeCardinality)) of `re` is used to define similar bounds for the number of individuals of the range of `re`.

Class LogicalFormula replaces class Predicate of the metamodel of Fig. 3 to represent constraints between domain model elements.

2.1.2. Additional Constraints

This section defines the constraints that are required to preserve the formal semantics of the domain modeling language and to ensure an unambiguous transformation of any domain model to a *B System* specification. The constraints are defined using the *B* syntax [1].

- $x \in \mathsf{Concept} \setminus \mathsf{Association}$
  $\Rightarrow \mathsf{Individual\_individualOf\_Concept}^{-1}[\{x\}] \cap \mathsf{MapletIndividual} = \emptyset$: if concept `x` is not an association, then no individual of `x` can be a maplet individual.

[3] An entity `ec`, defined in a concrete domain model, refines the entity `ea`, defined in an abstract domain model, if it can be deduced that *ec* = *ea* from domain model definitions

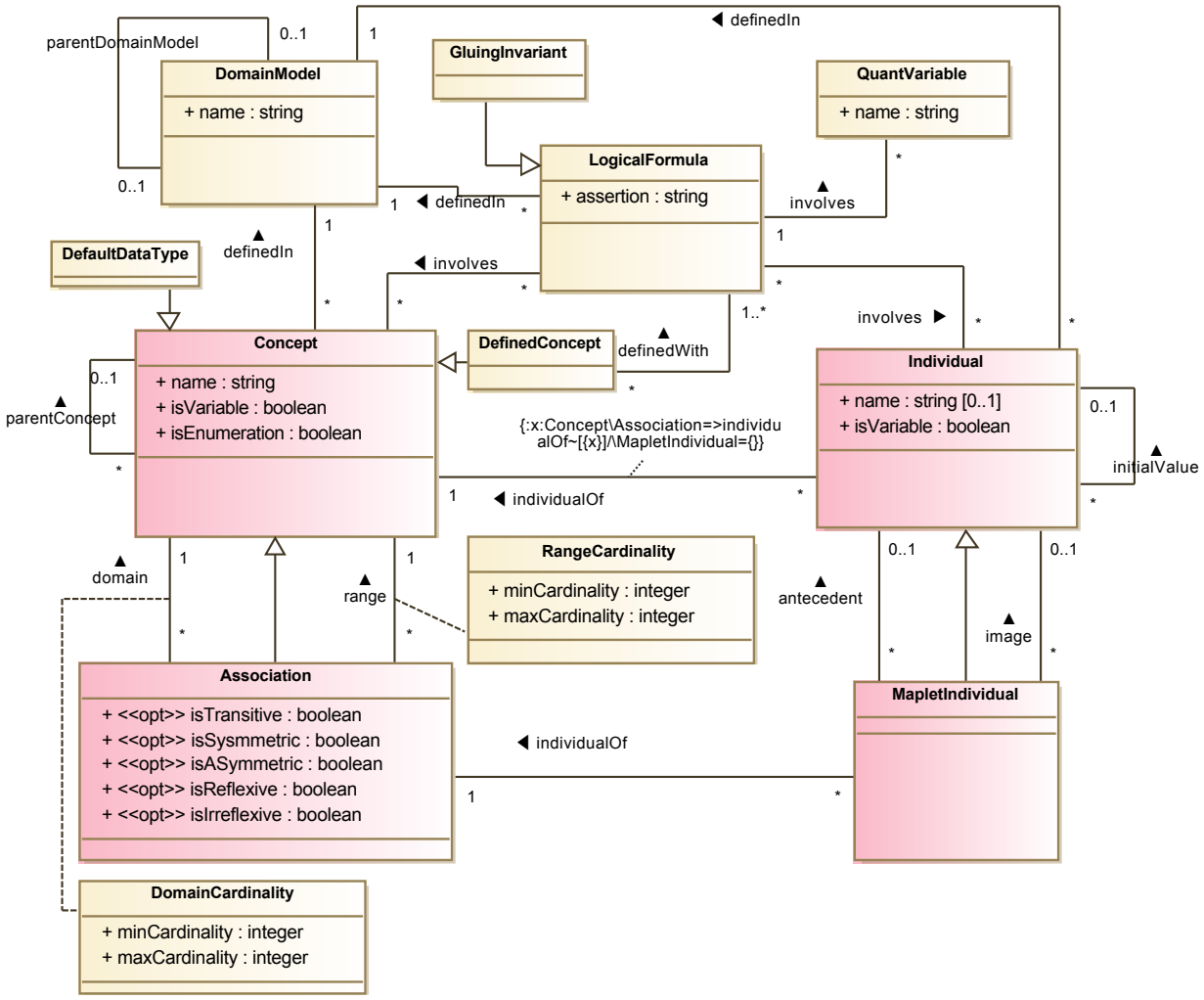

Figure 5: Excerpt from the updated SysML/KAOS domain metamodel

- $x \in$ MapletIndividual $\cap\, dom$(MapletIndividual_antecedent_Individual)
  $\Rightarrow$ MapletIndividual_antecedent_Individual(x) $\in$ Association_domain_Concept(Individual_individualOf_Concept(x)):
  if maplet individual x has an antecedent, then the antecedent is an individual of the domain of its association.
- $x \in$ MapletIndividual $\cap\, dom$(MapletIndividual_image_Individual)
  $\Rightarrow$ MapletIndividual_image_Individual(x) $\in$ Association_range_Concept(Individual_individualOf_Concept(x)):
  if maplet individual x has an image, then the image is an individual of the range of its association.
- $ind \in$ Individual $\setminus$ MapletIndividual $\Rightarrow ind \in dom$(Individual_name): every individual which is not a maplet individual must be named.
- $ind \in$ Individual $\setminus\, dom$(Individual_name) $\Rightarrow$ Individual_isVariable(ind) $= FALSE$: every unnamed individual must be constant.
- $ind \in$ MapletIndividual$\cap dom$(MapletIndividual_antecedent_Individual)$\cap dom$(MapletIndividual_image_Individual)
  $\Rightarrow$ (MapletIndividual_antecedent_Individual(ind) $\in dom$(Individual_name)$\wedge$MapletIndividual_image_Individual(ind) $\in$ dom(Individual_name)): antecedents and images of maplet individuals must be named.
- $ind \in$ MapletIndividual $\setminus\, dom$(Individual_name)
  $\Rightarrow\ ind \in dom$(MapletIndividual_antecedent_Individual) $\cap\, dom$(MapletIndividual_image_Individual): every unnamed maplet individual must have an antecedent and an image.

- $x \in$ Concept $\setminus$ (Association $\cup$ DefinedConcept $\cup$ dom(Concept_parent_Concept))
  $\Rightarrow$ Concept_isVariable(x) = $FALSE$: every abstract concept (that has no parent concept) that is not an association must be constant.
- $x \in$ Concept $\wedge$ Concept_isEnumeration(x) = $TRUE$ $\Rightarrow$ Concept_isVariable(x) = $FALSE$: every concept that is an enumeration must be constant.
- ($ind \in$ MapletIndividual$\cap dom$(MapletIndividual_antecedent_Individual)$\cap dom$(MapletIndividual_image_Individual)
  $\wedge$ Individual_isVariable(ind) = $FALSE$)
  $\Rightarrow$ (Individual_isVariable(MapletIndividual_antecedent_Individual(ind)) = $FALSE$
  $\wedge$ Individual_isVariable(MapletIndividual_image_Individual(ind)) = $FALSE$): antecedents and images of constant maplet individuals must be constant.
- ($x \in$ Association $\wedge$ Concept_isVariable(x) = $FALSE$)
  $\Rightarrow$ (Concept_isVariable(Association_domain_Concept(x)) = $FALSE$
  $\wedge$ Concept_isVariable(Association_range_Concept(x)) = $FALSE$): domains and ranges of constant associations must be constant.

2.2. *Illustration*

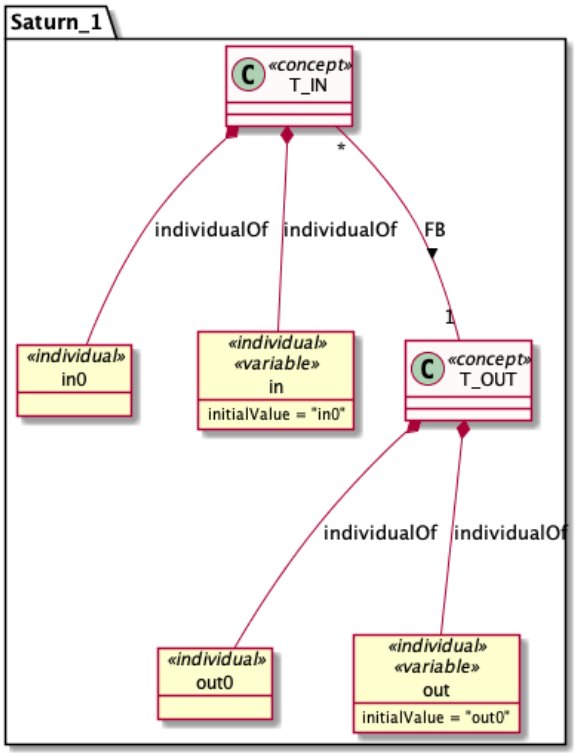

Figure 6: ***Saturn_1***: ontology associated with the root level of the goal diagram of Fig. 2

Figures 6, 7, 8, 9 represent domain models associated with refinement levels 0 (root level) .. 3 of the goal diagram of Fig. 2 using the updated SysML/KAOS domain modeling language. They are illustrated using the syntax proposed by the *SysML/KAOS Domain Modeling tool* [13][4] and, for readability purposes, we have decided to hide the representation of optional characteristics.

[4]The tool has been implemented on top of *Jetbrains MPS* [14] and PlantUML [15] to provide a proof of concept of the SysML/KAOS Domain Modeling Language.

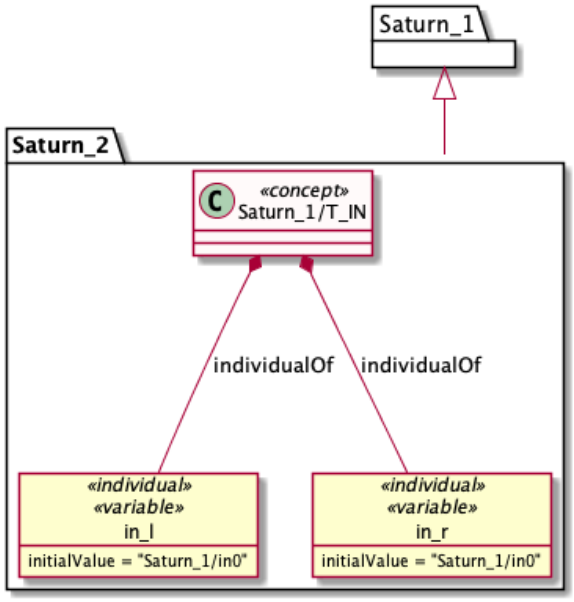

Figure 7: ***Saturn_2***: ontology associated with the first refinement level of the goal diagram of Fig. 2

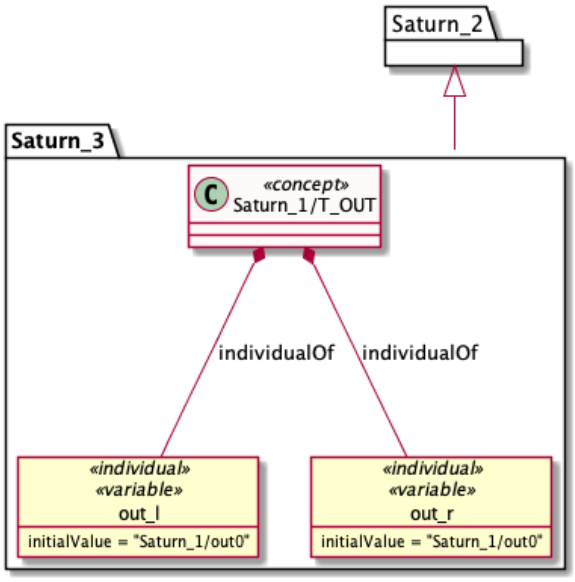

Figure 8: ***Saturn_3***: ontology associated with the second refinement level of the goal diagram of Fig. 2

In domain model `Saturn_1` (Fig. 6), the type of input data is modeled as a constant concept `T_IN` (instance of class Concept of Fig. 5) defining a variable individual `in` (instance of class Individual of Fig. 5) which represents the input data. Similarly, the type of output data is modeled as a constant concept `T_OUT` defining a variable individual `out` which represents the output data. Finally, the computation function `FB` is modeled as a functional association (instance of class Association of Fig. 5) from `T_IN` to `T_OUT`. Constant individuals `in0` and `out0` represent respectively the initial value of `in` and that of `out`.

In domain model `Saturn_2` (Fig. 7) which refines `Saturn_1`, individual `in` is refined by an individual named `in_l` (*in_l* = *in*) and a new variable individual named `in_r` is defined to represent the acquired input data. Similarly, in domain model `Saturn_3` (domain model associated with refinement level 2 of the goal diagram of Fig. 2), `out` is

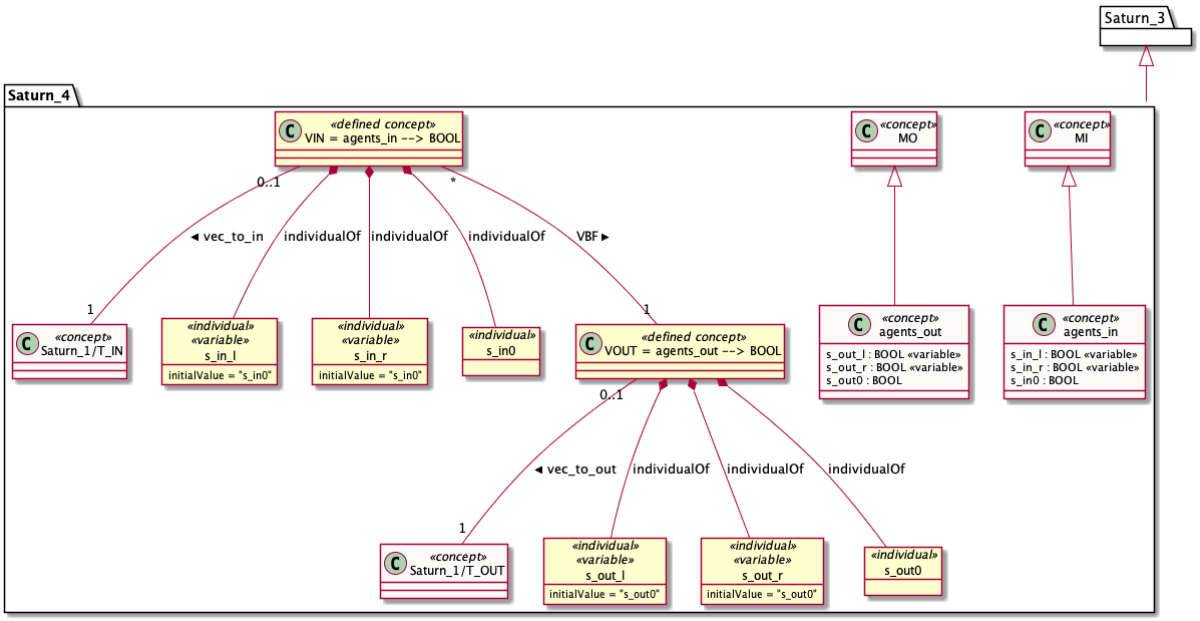

Figure 9: ***Saturn_4***: ontology associated with the third refinement level of the goal diagram of Fig. 2

refined by `out_l` (*out_l* = *out*) and individual `out_r` is added.

In domain model `Saturn_4` (Fig. 9) which refines `Saturn_3`, two concepts are defined: `MI` which represents the set of input agents and `MO` which represents the set of output agents. Concept `agents_in` (respectively `agents_out`) is a subconcept of `MI` (respectively `MO`) which represents the set of input (respectively output) agents that are active. Concept `VIN`, defined as the set of total functions from `agents_in` to `BOOL` ($VIN = agents\_in \longrightarrow BOOL$ where $BOOL = \{TRUE, FALSE\}$), represents the type of input data which are now arrays. Similarly, concept `VOUT` ($VOUT = agents\_out \longrightarrow BOOL$) represents the type of output data. Individuals `in_l`, `in_r`, `out_l` and `out_r` are refined respectively by individuals `s_in_l`, `s_in_r`, `s_out_l` and `s_out_r` using total injective associations `vec_to_in` from `VIN` to `T_IN` and `vec_to_out` from `VOUT` to `T_OUT`: $in\_l = vec\_to\_in(s\_in\_l), in\_r = vec\_to\_in(s\_in\_r), out\_l = vec\_to\_out(s\_out\_l), out\_r = vec\_to\_out(s\_out\_r)$. Finally, the computation function is modeled as a functional association named `VBF` from `VIN` to `VOUT`: $VBF = vec\_to\_in; FB; vec\_to\_out^{-1}$ (operator ; is the association composition operator used in logical formula assertions).

3. Updates in Translation Rules from Domain Models to B System Specifications

In the following, we describe a set of rules that allow to obtain a *B System* specification from domain models that conform to the updated SysML/KAOS domain modeling language.

Table 1 gives the translation rules. It should be noted that *o_x* designates the result of the translation of *x*. In addition, when used, qualifier *abstract* denotes "without parent". The rules have been implemented within the *SysML/KAOS Domain Modeling tool* [13] built on top of *Jetbrains MPS* [14] and PlantUML [15] to provide a proof of concept of the SysML/KAOS Domain Modeling Language. Rules *3, 4, 6..8*, and *12..16* have undergone significant updates to the previously defined translation rules [16].

Table 1: The translation rules

		Domain Model		B System	
	Translation Of	**Element**	**Constraint**	**Element**	**Constraint**
1	**Abstract domain model**	DM	$DM \in$ DomainModel $DM \notin dom$(DomainModel_parent_DomainModel)	o_DM	$o\_DM \in$ System
2	**Domain model with parent**	DM PDM	$\{DM, PDM\} \subseteq$ DomainModel $PDM =$ DomainModel_parent_DomainModel(DM) $o\_PDM \in$ Component	o_DM	$o\_DM \in$ Refinement Refinement_refines_Component(o_DM) = o_PDM

3	**Abstract concept that is not an enumeration**	CO	$CO \in$ Concept \ (Association ∪ DefinedConcept ∪ DefaultDataType) $CO \notin dom$(Concept_parent_Concept) Concept_isEnumeration$(CO) = FALSE$	o_CO	$o\_CO \in$ AbstractSet
4	**Abstract concept that is an enumeration**	CO $(I_j)_{j\in 1..n}$	$CO \in$ Concept \ (Association ∪ DefinedConcept ∪ DefaultDataType) $CO \notin dom$(Concept_parent_Concept) Concept_isEnumeration$(CO) = TRUE$ $\forall j \in 1..n, I_j \in$ Individual ∧ Individual_individualOf_Concept$(I_j) = CO$ ∧ Individual_isVariable$(I_j) = FALSE$	o_CO $(o\_I_j)_{j\in 1..n}$	$o\_CO \in$ EnumeratedSet $\forall j \in 1..n, o\_I_j \in$ SetItem ∧ SetItem_itemOf_EnumeratedSet$(o_I_j) = o_CO$
5	**Concept with constant parent**	CO PCO	$\{CO, PCO\} \subseteq$ Concept Concept_parent_Concept$(CO) = PCO$ $o\_PCO \in$ Set ∪ Constant	o_CO	**IF** $Concept\_isVariable(CO) = FALSE$ **THEN** $o\_CO \in$ Constant **ELSE** $o\_CO \in$ Variable **LogicFormula:** $o\_CO \subseteq o\_PCO$
6	**Constant concept with variable parent**	CO PCO PPCO	$\{CO, PCO, PPCO\} \subseteq$ Concept Concept_isVariable$(CO) = FALSE$ Concept_parent_Concept$(CO) = PCO$ $o\_PCO \in$ Variable $PPCO \in (closure1$(Concept_parent_Concept))[$\{PCO\}$][5] $o\_PPCO \in$ Set ∪ Constant	o_CO	$o\_CO \in$ Constant **Property:** $o\_CO \subseteq o\_PPCO$ **Invariant:** $o\_CO \subseteq o\_PCO$
7	**Variable concept with variable parent**	CO PCO	$\{CO, PCO\} \subseteq$ Concept Concept_isVariable$(CO) = TRUE$ Concept_parent_Concept$(CO) = PCO$ $o\_PCO \in$ Variable	o_CO	$o\_CO \in$ Variable **Invariant:** $o\_CO \subseteq o\_PCO$
8	**Enumerated concept with parent**	CO $(I_j)_{j\in 1..n}$	$CO \in dom$(Concept_parent_Concept) Concept_isEnumeration$(CO) = TRUE$ $\forall j \in 1..n, I_j \in$ Individual ∧ Individual_individualOf_Concept$(I_j) = CO$ ∧ Individual_isVariable$(I_j) = FALSE$ $o\_CO \in$ Constant[6] $\forall j \in 1..n, o\_I_j \in o\_CO$		**Property:** $o\_CO = (o\_I_j)_{j\in 1..n}$
(9)	**Association or defined concept without parent**	CO	$CO \in$ (DefinedConcept ∪ Association) $CO \notin dom$(Concept_parent_Concept)[7] To ensure that each variable or constant is typed, this rule has to be combined with either rule 10, for associations, or with a translation of the defining logical formula (contained in definedWith), for defined concepts.	o_CO	**IF** Concept_isVariable$(CO) = FALSE$ **THEN** $o\_CO \in$ Constant **ELSE** $o\_CO \in$ Variable
10	**Association**	AS CO1 CO2 da di ra ri	$\{CO1, CO2\} \subseteq$ Concept $AS \in$ Association $CO1$ = Association_domain_Concept(AS) $CO2$ = Association_range_Concept(AS) Association_DomainCardinality_maxCardinality$(AS) = da$ Association_DomainCardinality_minCardinality$(AS) = di$ Association_RangeCardinality_maxCardinality$(AS) = ra$ Association_RangeCardinality_minCardinality$(AS) = ri$ $o\_AS \in$ Constant ∪ Variable $\{o\_CO1, o\_CO2\} \subseteq$ (Set ∪ Constant ∪ Variable)	T_o_AS	**IF** Concept_isVariable$(CO1) = FALSE$ ∧ Concept_isVariable$(CO2) = FALSE$ **THEN** $T\_o\_AS \in$ Constant **ELSE** $T\_o\_AS \in$ Variable **IF** $\{ra, ri, da, di\} = \{1\}$ **THEN LogicFormula:** $T\_o\_AS = o\_CO1 \rightarrowtail\!\!\!\!\!\twoheadrightarrow o\_CO2$ **ELSE IF** $\{ra, ri, da\} = \{1\}$ **THEN LogicFormula:** $T\_o\_AS = o\_CO1 \rightarrowtail o\_CO2$ **ELSE IF** $\{ra, ri, di\} = \{1\}$ **THEN LogicFormula:** $T\_o\_AS = o\_CO1 \twoheadrightarrow o\_CO2$ **ELSE IF** $\{ra, di\} = \{1\}$ **THEN LogicFormula:** $T\_o\_AS = o\_CO1 \nrightarrow o\_CO2$ **ELSE IF** $\{ra, da\} = \{1\}$ **THEN LogicFormula:** $T\_o\_AS = o\_CO1 \rightarrowtail\!\!\!\!\!\nrightarrow o\_CO2$ **ELSE IF** $\{ra, ri\} = \{1\}$ **THEN LogicFormula:** $T\_o\_AS = o\_CO1 \longrightarrow o\_CO2$ **ELSE IF** $ra = 1$ **THEN LogicFormula:** $T\_o\_AS = o\_CO1 \nrightarrow o\_CO2$ **ELSE** **LogicFormula:** $T\_o\_AS = o\_CO1 \leftrightarrow o\_CO2$ $\wedge\forall x.(x \in CO2 \Rightarrow card(o\_RE^{-1}[\{x\}]) \in di..da)$ $\wedge\forall x.(x \in CO1 \Rightarrow card(o\_RE[\{x\}]) \in ri..ra)$ **LogicFormula:** $o\_AS \in T\_o\_AS$

[5] *closure1(Concept_parent_Concept)* designates the transitive closure of relation Concept_parent_Concept

[6] Every concrete enumeration is a constant

[7] If CO has a parent concept, o_CO must be introduced by rule 5. It is therefore necessary to ensure that this is not the case.

11	**Individual of a constant concept that is not an abstract enumeration**	Ind CO	$Ind \in$ Individual \ MapletIndividual CO = Individual_individualOf_Concept(Ind) $o\_CO \in$ AbstractSet ∪ Constant	o_Ind	**IF** Individual_isVariable(Ind) = $TRUE$ **THEN** $o\_Ind \in$ Variable **ELSE** $o\_Ind \in$ Constant **LogicFormula:** $o\_Ind \in o\_CO$
12	**Constant individual of a variable concept**	Ind CO PPCO	$Ind \in$ Individual \ MapletIndividual Individual_isVariable(Ind) = $FALSE$ CO = Individual_individualOf_Concept(Ind) $o\_CO \in$ Variable $PPCO \in$ Concept $PPCO \in (closure1$(Concept_parent_Concept))[{CO}] $o\_PPCO \in$ Set ∪ Constant	o_Ind	$o\_Ind \in$ Constant **Property:** $o\_Ind \in o\_PPCO$ **Invariant:** $o\_Ind \in o\_CO$
13	**Variable individual of a variable concept**	Ind CO	$Ind \in$ Individual \ MapletIndividual Individual_isVariable(Ind) = $TRUE$ CO = Individual_individualOf_Concept(Ind) $o\_CO \in$ Variable	o_Ind	$o\_Ind \in$ Variable **Invariant:** $o\_Ind \in o\_CO$
14	**Variable individual of a concept that is an abstract enumeration**	Ind CO	$Ind \in$ Individual \ MapletIndividual Individual_isVariable(Ind) = $TRUE$ CO = Individual_individualOf_Concept(Ind) Concept_isEnumeration(CO) = $TRUE$ $CO \notin dom$(Concept_parent_Concept) $o\_CO \in$ EnumeratedSet	o_Ind	$o\_Ind \in$ Variable **Invariant:** $o\_Ind \in o\_CO$
15	**Maplet individual**	Ind AS Ant Im PPCO1 PPCO2	$Ind \in$ MapletIndividual AS = Individual_individualOf_Concept(Ind)[8] $o\_AS \in$ Constant ∪ Variable $Ind \in dom$(MapletIndividual_antecedent_Individual) ⇒ Ant = MapletIndividual_antecedent_Individual(Ind) $o\_Ant \in$ Constant ∪ Variable $Ind \in dom$(MapletIndividual_image_Individual) ⇒ Im = MapletIndividual_image_Individual(Ind) $o\_Im \in$ Constant ∪ Variable {$PPCO1, PPCO2$} ⊆ Concept $PPCO1 \in (closure1$(Concept_parent_Concept))[{Association_domain_Concept(AS)}] $PPCO2 \in (closure1$(Concept_parent_Concept))[{MapletIndividual_range_Individual(AS)}] {o_PPCO1, o_PPCO2} ⊆ Set ∪ Constant	o_Ind	**IF** $Ind \in dom$(Individual_name) **THEN** **IF** Individual_isVariable(Ind) = $TRUE$ **THEN** $o\_Ind \in$ Variable **Invariant:** $o\_Ind \in o\_AS$ **IF** $Ind \in dom$(MapletIndividual_antecedent_Individual) ∩ dom(MapletIndividual_image_Individual) **THEN Invariant:** $o\_Ind = o\_Ant \mapsto o\_Im$ **ELSE** $o\_Ind \in$ Constant **IF** $o\_AS \in$ Constant **THEN Property:** $o\_Ind \in o\_AS$ **ELSE** **Property:** $o\_Ind \in o\_PPCO1 \leftrightarrow o\_PPCO2$ **Invariant:** $o\_Ind \in o\_AS$ **IF** $Ind \in dom$(MapletIndividual_antecedent_Individual) ∩ dom(MapletIndividual_image_Individual) **THEN Property:** $o\_Ind = o\_Ant \mapsto o\_Im$ **ELSE LogicFormula:** $o\_Ant \mapsto o\_Im \in o\_AS$[9]
16	**Variable individual initialisation**	Ind Init CO Init_ant Init_im	$Ind \in$ Individual ∩ $dom(Individual\_name)$ $Individual\_isVariable(Ind) = TRUE$ $o\_Ind \in$ Variable CO = Individual_individualOf_Concept(Ind) $o\_CO \in$ Set ∪ Constant ∪ Variable $Ind \notin dom$(Individual_initialValue_individual) ∨ (Individual_initialValue_individual(Ind) = $Init$ ∧ (($Init \notin dom$(Individual_name) ∧ $Init\_ant$ = MapletIndividual_antecedent_Individual($Init$) ∧ $Init\_im$ = MapletIndividual_image_Individual($Init$) ∧ {$Init\_ant, Init\_im$} ⊆ Constant ∪ Variable) ∨ $o\_Init \in$ Constant ∪ Variable))		**IF** $Ind \notin dom$(Individual_initialValue_individual) **THEN** $o\_Ind :: o\_CO$ **ELSE** **IF** $Init \notin dom$(Individual_name) **THEN Initialisation:** $o\_Ind := o\_Ant \mapsto o\_Im$ **ELSE Initialisation:** $o\_Ind := o\_Init$
17	**Variable concept initialisation**	CO $(I_j)_{j \in 1..n}$	$CO \in dom$(Concept) Concept_isVariable(CO) = $TRUE$ $\forall j \in 1..n, I_j \in$ Individual ∧ Individual_individualOf_Concept(I_j) = CO ∧ Individual_isVariable(I_j) = $FALSE$ $o\_CO \in$ Variable $\forall j \in 1..n, o\_I_j \in o\_CO$		**Initialisation:** $o\_CO := (o\_I_j)_{j \in 1..n}$[10]

[8] AS must be an association

[9] Following the variability status of o_AS, this predicate can be a property or an invariant

[10] If $\exists j \in 1..n . I_j \notin dom$(Individual_name) then o_I_j must be replaced by $o\_I_j\_Ant \mapsto o\_I_j\_Im$ as in the previous rule

18	**Association transitivity**	AS	$AS \in$ Association Association_isTransitive(AS) = $TRUE$ $o\_AS \in$ Constant ∪ Variable		**LogicFormula:** $(o\_AS \ ; \ o\_AS) \subseteq o\_AS$
19	**Association symmetry**	AS	$AS \in$ Association Association_isSymmetric(AS) = $TRUE$ $o\_AS \in$ Constant ∪ Variable		**LogicFormula:** $o\_AS^{-1} = o\_AS$
20	**Association asymmetry**	AS CO	$AS \in$ Association Association_isSymmetric(AS) = $TRUE$ $o\_AS \in$ Constant ∪ Variable Association_domain_Concept(AS) = CO $o\_CO \in$ Set ∪ Constant ∪ Variable		**LogicFormula:** $(o\_AS^{-1} \cap o\_AS) \subseteq id(o\_CO)$
21	**Association reflexivity**	AS CO	$AS \in$ Association Association_isReflexive(AS) = $TRUE$ $o\_AS \in$ Constant ∪ Variable Association_domain_Concept(AS) = CO $o\_CO \in$ Set ∪ Constant ∪ Variable		**LogicFormula:** $id(o\_CO) \subseteq o\_AS$
22	**Association irreflexivity**	AS CO	$AS \in$ Association Association_isIrreflexive(AS) = $TRUE$ $o\_AS \in$ Constant ∪ Variable Association_domain_Concept(AS) = CO $o\_CO \in$ Set ∪ Constant ∪ Variable		**LogicFormula:** $id(o\_CO) \cap o\_AS = \emptyset$

Each logical formula is translated with the definition of a *B System* logic formula corresponding to its assertion. Since both languages use first-order logic notations, the translation is limited to a syntactic rewriting.

4. Updates in Back Propagation Rules from B System Specifications to Domain Models

We choose to support only the most repetitive additions that can be performed within the formal specification, the domain model remaining the one to be updated in case of any major changes such as the addition or the deletion of a refinement level. Table 2 summarises the most relevant back propagation rules. Each rule defines its inputs (elements added to the *B System* specification) and constraints that each input must fulfill. It also defines its outputs (elements introduced within domain models as a result of the application of the rule) and their respective constraints. It should be noted that for an element *b_x* of the *B System* specification, *o_x* designates the domain model element corresponding to *b_x*. In addition, when used, qualifier *abstract* denotes "without parent".

Table 2: back propagation rules in case of addition of an element in the *B System* specification

		B System		**Domain Model**	
	Addition Of	**Input**	**Constraint**	**Output**	**Constraint**
1	**Abstract set**	b_CO	$b\_CO \in$ AbstractSet	o_CO	$o\_CO \in$ Concept
2	**Abstract enumeration**	b_CO $(b\_I_j)_{j\in 1..n}$	$b\_CO \in$ EnumeratedSet $\forall j \in 1..n, b\_I_j \in$ SetItem $\wedge$ SetItem_itemOf_EnumeratedSet(b_I_j) = b_CO	o_CO $(o\_I_j)_{j\in 1..n}$	$o\_CO \in$ Concept Concept_isEnumeration(o_CO) = $TRUE$ $\forall j \in 1..n, o\_I_j \in$ Individual $\wedge$ Individual_individualOf_Concept(o_I_j) = o_CO
3	**Set item**	b_elt b_ES	$b\_elt \in$ SetItem b_ES = SetItem_itemOf_EnumeratedSet(b_elt) $o\_ES \in$ Concept	o_elt	$o\_elt \in$ Individual Individual_individualOf_Concept(o_elt) = o_ES
4	**Constant typed as subset of the correspondent of a concept**	b_CO b_PCO	$b\_CO \in$ Constant $b\_PCO \in$ AbstractSet ∪ Constant $b\_CO \subseteq b\_PCO$ $o\_PCO \in$ Concept	o_CO	$o\_CO \in$ Concept Concept_parent_Concept(o_CO) = o_PCO
5	**Constant typed as item of the correspondent of a concept**	b_elt b_CO	$b\_elt \in$ Constant $b\_CO \in$ AbstractSet ∪ Constant $b\_elt \in b\_CO$ $o\_CO \in$ Concept	o_elt	$o\_elt \in$ Individual Individual_individualOf_Concept(o_elt) = o_CO
6	**Variable typed as subset of the correspondent of a concept**	b_CO b_PCO	$b\_CO \in$ Variable $b\_PCO \in$ AbstractSet ∪ Constant ∪ Variable $b\_CO \subseteq b\_PCO$ $o\_PCO \in$ Concept	o_CO	$o\_CO \in$ Concept Concept_parent_Concept(o_CO) = o_PCO Concept_isVariable(CO) = $TRUE$
7	**Variable typed as item of the correspondent of a concept**	b_elt b_CO	$b\_elt \in$ Variable $b\_CO \in$ AbstractSet ∪ Constant ∪ Variable $b\_elt \in b\_CO$ $o\_CO \in$ Concept	o_elt	$o\_elt \in$ Individual Individual_individualOf_Concept(o_elt) = o_CO Individual_isVariable(o_elt) = $TRUE$
8	**Constant typed as a relation**	b_AS b_CO1 b_CO2	$b\_AS \in$ Constant $\{b\_CO1, b\_CO2\} \subset$ AbstractSet ∪ Constant $b\_AS \in b\_CO1 \leftrightarrow b\_CO2$ $\{o\_CO1, o\_CO2\} \subset$ Concept	o_AS	$o\_AS \in$ Association Association_domain_Concept(o_AS) = o_CO1 Association_range_Concept(o_AS) = o_CO2 As usual, the cardinalities of o_AS are set according to the type of b_AS (*function*, *injection*, ...).

9	**Variable typed as a relation**	b_AS b_CO1 b_CO2	$b\_AS \in$ Variable $\{b\_CO1, b\_CO2\} \subset$ AbstractSet $\cup$ Constant $\cup$ Variable $b\_AS \in b\_CO1 \leftrightarrow b\_CO2$ $\{o\_CO1, o\_CO2\} \subset$ Concept	o_AS	$o\_AS \in$ Association Association_domain_Concept(o_AS) = o_CO1 Association_range_Concept(o_AS) = o_CO2 Association_isVariable(o_AS) = $TRUE$ As usual, the cardinalities of o_AS are set according to the type of b_AS (*function*, *injection*, ...).
10	**Constant typed as a maplet**	b_elt b_ant b_im	$b\_elt \in$ Constant $\{b\_ant, b\_im\} \subset$ Constant $b\_elt = b\_ant \mapsto b\_im$ $\{o\_ant, o\_im\} \subset$ Individual	o_elt	$o\_elt \in$ Individual MapletIndividual_antecedent_Individual(o_elt) = o_ant MapletIndividual_image_Individual(o_elt) = b_im
11	**Variable typed as a maplet**	b_elt b_ant b_im	$b\_elt \in$ Variable $\{b\_ant, b\_im\} \subset$ Constant $\cup$ Variable $b\_elt = b\_ant \mapsto b\_im$ $\{o\_ant, o\_im\} \subset$ Individual	o_elt	$o\_elt \in$ Individual MapletIndividual_antecedent_Individual(o_elt) = o_ant MapletIndividual_image_Individual(o_elt) = b_im Individual_isVariable(o_elt) = $TRUE$
12	**Variable initialised to the correspondent of an individual**	b_elt b_init	$b\_elt \in$ Variable $b\_init \in$ Constant **Initialisation:** $b\_elt := b\_init$ $\{o\_init, o\_elt\} \subseteq$ Individual		Individual_initialValue_Individual(o_elt) = o_init

The addition of a non typing logic formula (logic formula that does not contribute to the definition of the type of a formal element) in the *B System* specification is propagated through the definition of the same formula in the corresponding domain model, since both languages use first-order logic notations. This back propagation is limited to a syntactic translation.

A fresh *B System* constant or variable *b_x* is defined within the domain model, by default, as a defined concept (instance of DefinedConcept), until a typing *B System* logical formula is introduced (subset of the correspondence of a concept, relation, item of the correspondence of a concept or maplet). The concept *b_x* is defined with correspondence of *B System* logical formulas where *b_x* appears: there must be at least one.

Acknowledgment

This work is carried out within the framework of the *FORMOSE* project [2] 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] J. Abrial, Modeling in Event-B - System and Software Engineering, Cambridge University Press, 2010.

[2] ANR-14-CE28-0009, Formose ANR project (2017).
URL `http://formose.lacl.fr/`

[3] ClearSy, Atelier B: B System (2014).
URL `http://clearsy.com/`

[4] R. Laleau, F. Semmak, A. Matoussi, D. Petit, A. Hammad, B. Tatibouet, A first attempt to combine SysML requirements diagrams and B, Innovations in Systems and Software Engineering 6 (1-2) (2010) 47–54.

[5] A. Mammar, 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.

[6] M. Hause, et al., The SysML modelling language, in: Fifteenth European Systems Engineering Conference, Vol. 9, Citeseer, 2006.

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

[8] K. Sengupta, P. Hitzler, Web ontology language (OWL), in: Encyclopedia of Social Network Analysis and Mining, 2014, pp. 2374–2378.

[9] G. Pierra, The PLIB ontology-based approach to data integration, in: IFIP 18th World Computer Congress, Vol. 156 of IFIP, Kluwer/Springer, 2004, pp. 13–18.

[10] S. Tueno, R. Laleau, A. Mammar, M. Frappier, The SysML/KAOS Domain Modeling Approach, ArXiv e-prints, cs.SE, 1710.00903.
URL https://arxiv.org/pdf/1710.00903.pdf

[11] S. Tueno, R. Laleau, A. Mammar, M. Frappier, Towards using ontologies for domain modeling within the SysML/KAOS approach, in: RE Workshops, IEEE Computer Society, 2017, pp. 1–5.

[12] I. UL, Owlgred home (2017).
URL http://owlgred.lumii.lv/

[13] S. Tueno, R. Laleau, A. Mammar, M. Frappier, SysML/KAOS Domain Modeling Tool (2017).
URL https://github.com/anonym21/SysML_KAOS_Domain_Model_Parser

[14] Jetbrains, Jetbrains mps (2017).
URL https://www.jetbrains.com/mps/

[15] A. Roques, Plantuml: Open-source tool that uses simple textual descriptions to draw uml diagrams (2015).

[16] S. Tueno, M. Frappier, R. Laleau, A. Mammar, From SysML/KAOS Domain Models to B System Specifications, ArXiv e-prints, cs.SE, 1803.01972.
URL https://arxiv.org/pdf/1803.01972.pdf