Towards a tooled and proven formal requirements engineering approach

Steve Jeffrey Tueno Fotso

VERS UNE APPROCHE FORMELLE D'INGÉNIERIE DES EXIGENCES OUTILLÉE ET ÉPROUVÉE

par

Steve Jeffrey TUENO FOTSO

Thèse en cotutelle présentée à l'école doctorale MSTIC
en vue de l'obtention du grade de philosophiæ doctor (Ph.D.)

Groupe de Recherche en Informatique Fondamentale, Faculté des Sciences, Université de Sherbrooke, Sherbrooke QC, Canada

Laboratoire Algorithmique Complexité Logique
Université Paris Est Créteil, Créteil, France

Composition du Jury

Président

Jean-Michel BRUEL, Professeur, Université de Toulouse, France

Rapporteurs

Daniel AMYOT, Professeur, Université d'Ottawa, Canada
David CHEMOUIL, Chercheur, ONERA, France

Co-directeurs

Régine LALEAU, Professeure, Université Paris Est Créteil, France
Marc FRAPPIER, Professeur, Université de Sherbrooke, Canada
Amel MAMMAR, Professeure, Télécom-SudParis, Institut Polytechnique de Paris, France

Examinateurs

Lynda MOKDAD, Professeure, Université Paris Est Créteil, France
Michael BLONDIN, Professeur, Université de Sherbrooke, Canada

Soutenue à Créteil, France, le 22 octobre 2019

Sommaire

La méthode SysML/KAOS permet de modéliser les exigences d'un système sous forme d'hiérarchies de buts. *B System* est une méthode formelle qui permet de construire, vérifier et valider la spécification d'un système. Un modèle *B System* est constitué d'une *partie structurelle* (ensembles abstraits et énumérés, constantes et leurs propriétés, et variables et leur invariant) et d'une *partie comportementale* (évènements). Lors de travaux antérieurs, des liens de correspondance ont été établis entre SysML/KAOS et *B System* afin de produire une spécification formelle à partir de la modélisation des exigences. Cette spécification sert de base pour les tâches de vérification et de validation formelle afin de détecter et corriger les potentielles erreurs de spécification. Toutefois, il est nécessaire de fournir manuellement la partie structurelle du modèle *B System*.

L'objectif de cette thèse est d'enrichir SysML/KAOS avec un langage permettant de modéliser le domaine du système et qui serait compatible avec le langage de modélisation des exigences. Ceci inclut non seulement la définition du langage, mais aussi des mécanismes permettant d'exploiter la modélisation du domaine pour fournir la partie structurelle de la spécification *B System* issue de la formalisation des exigences. Le langage défini exploite la notion d'ontologie pour permettre la représentation formelle du domaine. Bien plus, les liens et règles de correspondance établis et formellement vérifiés permettent tant la propagation que la rétropropagation des ajouts et suppressions, entre modèles de domaine et spécifications *B System*. Un autre aspect essentiel de la thèse réside dans l'évaluation de la méthode SysML/-KAOS sur des études de cas. Par ailleurs, les systèmes, au vu de leur complexité, se doivent d'être décomposés en sous-systèmes. La thèse décrit en conséquence des mécanismes permettant de garantir formellement que chaque exigence affectée à un sous-système sera bien satisfaite par ce dernier, dans la limite définie par la spécification du système et des sous-systèmes.

La méthode SysML/KAOS, ainsi enrichie, a été implémentée au sein d'un outil libre en utilisant la plateforme de fédération de modèles *Openflexo*, et a été évaluée sur différentes études de cas d'envergure industrielle. Elle permet la vérification formelle des exigences et facilite leur validation par des parties prenantes non spécialistes

de méthodes formelles. Toutefois, les tâches de spécification des formules logiques du modèle de domaine, qui donnent lieu aux propriétés et invariants du modèle *B System*, et du corps des évènements *B System*, ainsi que les tâches de vérification et validation formelles sont coûteuses en temps et nécessitent l'implication d'experts en méthodes formelles. Il s'agit là du prix à payer pour des exigences formellement correctes.

Mots-clés: Ingénierie des exigences ; Modélisation du domaine ; Méthodes formelles ; Ontologies ; *SysML/KAOS* ; *B System* ; *Event-B*.

Abstract

The *SysML/KAOS* method allows to model system requirements through goal hierarchies. *B System* is a formal method used to construct, verify and validate system specifications. A *B System* model consists of a *structural part* (abstract and enumerated sets, constants with their associated properties, and variables with their associated invariant) and a *behavioral part* (events). Correspondence links are established in previous work between SysML/KAOS and *B System* to produce a formal specification from requirements models. This specification serves as a basis for formal verification and validation tasks to detect and correct inconsistencies. However, it is required to manually provide the structural part of the *B System* specification.

This thesis aims at enriching SysML/KAOS with a language that allows to model the application domain of the system and which would be compatible with the requirements modeling language. This includes the definition of the domain modeling language and of mechanisms for leveraging domain modeling to provide the structural part of the *B System* specification obtained from requirements models. The defined language uses ontologies to allow the formal representation of system domain. Moreover, the established correspondence links and rules, formally verified, allow both propagation and backpropagation of additions and deletions, between domain models and *B System* specifications. An important part of the thesis is also devoted to assessment of the SysML/KAOS method on case studies. Furthermore, since the systems naturally break down into subsystems (enabling the distribution of work between several agents: hardware, software and human), SysML/KAOS goal models allow the capture of assignments of requirements to subsystems responsible of their achievement. This thesis therefore describes the mechanisms required to formally guarantee that each requirement assigned to a subsystem will be well achieved by the subsystem, within the constraints defined by the high-level system and subsystem specifications.

Abstract

The SysML/KAOS method, thus enriched, has been implemented within an opensource tool using the model federation platform *Openflexo*, and has been evaluated on various industrial scale case studies. It enables the formal verification of requirements and facilitates their validation by the various stakeholders, including those with less or no expertise in formal methods. However, both the specification of the body of *B System* events and domain model logical formulas (that give *B System* properties and invariants) and the formal assessment (verification and validation) of the specification can only be manual. They are time consuming and require experts in formal methods. But this is the price to pay to achieve a formal verification and validation of requirements.

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

Remerciements

Cette thèse a été réalisée dans le cadre du projet *FORMOSE* [17] financé par *l'Agence Nationale de la Recherche française (ANR)*. Elle a également été financée par le *Conseil de Recherches en Sciences Naturelles et en Génie du Canada (CRSNG)*.

J'adresse un merci particulier à mes directrices et directeur de thèse la Pre *Régine Laleau*, le Pr *Marc Frappier* et la Pre *Amel Mammar*, pour leur soutien, leurs conseils et leur présence permanente à mes côtés tout au long de mon cheminement doctoral.

Merci aux équipes administratives de l'Université de Sherbrooke et de l'Université Paris-Est Créteil, pour toute l'assistance déployée sans cesse et sans répis afin de démystifier et faciliter ce processus complexe qu'est une cotutelle de thèse.

À ma famille, en l'occurrence ma maman, à tous ces amis et collègues qui m'ont accompagné et n'ont cessé de m'encourager, vous qui, du front aux yeux, n'avez cessé d'éponger sueurs et larmes : tout simplement, Merci !

Abréviations

AID Automated Incident Detection system (Système automatisé de détection d'incidents)

ANR Agence Nationale de la Recherche

ASTD Algebraic State Transition Diagram (Diagrammes d'états-transitions algébriques)

ERTMS European Railway Traffic Management System (Système européen de gestion du trafic ferroviaire)

ETCS European Train Control System (Système européen de contrôle des trains)

FML Federation Modeling Language (Langage de modélisation de fédérations)

KAOS Keep All Objectives Satisfied (Assurer la satisfaction de tous les objectifs)

MCAS Maneuvering Characteristics Augmentation System (Système d'amélioration des caractéristiques de manœuvrabilité)

MtQ Ministère des Transports du Québec

OWL Ontology Web Language (Langage des ontologies Web)

PLIB Part Library (Catalogue de composants)

SysML Systems Modeling Language (Langage de modélisation de systèmes)

VdM Ville de Montréal

Table des matières

Liste des figures

Liste des tableaux

Introduction

Contexte

L'ingénierie des exigences est la partie du génie logiciel qui s'intéresse aux activités d'élicitation, d'analyse, de spécification et de validation des exigences relatives à un système à mettre en place. Elle désigne les activités qui constituent la pierre angulaire de tout projet de développement logiciel ou système. L'occurrence de défaillances au cours de l'une de ces étapes a souvent des conséquences extrêmement désastreuses [94, 104]. Par exemple, en à peine six mois, deux vols, *Lion Air 610* et *Ethiopian Airlines 302*, se sont écrasés quelques minutes après le décollage. La cause de ces crashs successifs, impliquant des avions de type *Boeing 737 Max 8* et ayant engendré plus de 300 pertes en vie humaine, réside dans un conflit d'exigences (conjonction inadéquate de buts) impliquant le *pilote* et le système de stabilisation *MCAS (Maneuvering Characteristics Augmentation System)* [114]. En effet, le MCAS changeait sans cesse l'orientation de l'appareil en se basant sur une mesure éronnée de l'angle d'incidence, sans vraiment tenir compte des corrections effectuées par le pilote. Bahill *et al.* [130] font état de plusieurs autres désastres d'envergure liés à des défaillances impliquant l'ingénierie des exigences.

Le projet *FORMOSE* [17], financé par l'Agence Nationale de la Recherche (ANR) française, s'intéresse à cette problématique et vise l'élaboration d'une méthode outillée pour la modélisation, la vérification et la validation formelle des exigences de systèmes critiques et complexes. Cette méthode s'appuie sur *SysML/KAOS* [70, 92]. SysML/KAOS permet la modélisation des exigences d'un système sous forme de hiérarchies de buts. Afin de vérifier et valider formellement ces exigences, les travaux décrits dans [102] définissent une correspondance entre le langage SysML/KAOS de modélisation des exigences et la méthode formelle *Event-B* [8] permettant ainsi de produire des spécifications formelles à partir des modèles d'exigences. Cette spécification sert ensuite de base aux tâches de vérification et de validation formelles afin de détecter et corriger les potentielles défaillances. Il est à noter que ces tâches sont réalisables grâce aux outils associés aux méthodes Event-B voire *B* [10], outils largement éprouvés et utilisés sur des projets industriels depuis plus de 25 ans [93].

B System désigne une variante syntaxique d'Event-B proposée au sein de l'environnement de développement intégré *AtelierB* édité par *ClearSy* [41], un partenaire industriel au sein du projet FORMOSE [17]. C'est cette variante syntaxique, sémantiquement équivalente à Event-B, qui est considérée, tout au long de ce travail de thèse, pour produire une spécification formelle à partir des modèles SysML/KAOS.

Problématique

Les règles de correspondance définies dans [102] permettent d'obtenir l'ossature d'une spécification *B System* formalisant les exigences du système : chaque niveau de raffinement du modèle de buts se traduit par un composant *B System* et un squelette d'évènement est généré pour chaque but. En outre, plusieurs obligations de preuve sont générées afin de traduire la sémantique des liens de raffinement et permettre la vérification de la cohérence des raffinements de buts. Toutefois, il est nécessaire de fournir manuellement la *partie structurelle* du modèle *B System* (ensembles abstraits et énumérés, constantes et leurs propriétés, variables et invariant). En conséquence, il est difficile de lire, maintenir et même valider ces éléments complétés à la main par l'expert *B System*, surtout, comme c'est souvent le cas, lorsque les parties prenantes ne sont pas familières avec les méthodes formelles. Ceci est notamment accentué par l'absence d'une séparation claire entre ce qui est imposé par le domaine, et doit nécessairement figurer et être garanti, et ce qui est attendu du système [85]. De plus, la connaissance requise pour définir ces éléments, en l'occurrence ce qui a trait à la partie structurelle, est très souvent l'apanage de l'expert du domaine qui se distingue de l'expert *B System*.

Par ailleurs, pour gérer la complexité des systèmes, SysML/KAOS considère leur décomposition en sous-systèmes. Ces sous-systèmes peuvent, en outre, être déjà existants et avoir leurs propres fonctionnements. Le "premier" diagramme de buts SysML/KAOS construit est celui du système principal. La décomposition en sous-buts prend fin lorsqu'il est possible d'affecter chaque exigence à un sous-système qui est sous la responsabilité d'un agent. Les agents sont représentés au sein du modèle de buts et l'affectation d'une exigence à un sous-système passe par l'assignation de la responsabilité de la réalisation du but à l'agent responsable du sous-système. Des diagrammes de buts peuvent être définis pour les différents sous-systèmes. Ceux-ci peuvent comporter des buts propres, en plus de ceux provenant du système de niveau supérieur. Toutefois, aucun mécanisme ne permet pour l'instant de garantir que la spécification formelle construite à partir de la modélisation des exigences d'un sous-système assure que chaque exigence affectée à ce dernier sera bien satisfaite, dans la limite définie par la spécification formelle construite à partir de la modélisation des exigences du système de plus haut niveau. De plus, rien

ne permet de garantir formellement que les comportements propres aux différents sous-systèmes ne violeront pas les invariants décrivant un comportement adéquat du système de plus haut niveau, comme cela a été le cas pour les accidents des vols *Lion Air 610* et *Ethiopian Airlines 302* [114].

Objectifs

Étant donné que la partie structurelle d'une spécification *B System* constitue une caractérisation des propriétés du domaine d'application du système, l'objectif de ce travail de thèse est d'enrichir SysML/KAOS avec une approche ergonomique mais structurée et non ambigüe, supportée par un outil libre, permettant de modéliser le domaine du système et compatible avec le langage de modélisation des exigences. En effet, le choix du niveau de détail de la modélisation du domaine et des éléments qui doivent y figurer au regard des buts à satisfaire est du ressort de l'expert du domaine [19, 25]. Par conséquent, le langage de modélisation du domaine se doit d'être suffisamment simple et expressif afin de faciliter son utilisation par des personnes non expertes de méthodes formelles, en l'occurrence les experts du domaine. Par ailleurs, la modélisation du domaine permet également d'exhiber les potentielles omissions et incomplétudes introduites lors de l'élicitation des exigences [19].

Il s'agit également de définir et implémenter des mécanismes permettant d'exploiter la modélisation du domaine pour obtenir automatiquement la partie structurelle et maintenir l'adéquation entre cette dernière et les modèles SysML/KAOS auxquels elle est associée. En effet, comme l'affirment Lamsweerde *et al.* [140], spécifier formellement le corps des buts, de même que vérifier et valider cette spécification, conduit très souvent à des ajouts et modifications au sein de la partie structurelle, et donc des modèles de domaine.

Notons que la correction de ces approches et mécanismes doit être vérifiée formellement, au vu de la criticité des systèmes et domaines considérés, afin d'assurer que la méthode n'introduit pas d'incohérences ; c'est-à-dire que toute modélisation réalisée en respectant les contraintes établies doit être en mesure de produire, en temps fini, une spécification *B System* syntaxiquement correcte et sémantiquement équivalente et qui reste en adéquation avec les modèles auxquels elle est associée au fil des mises à jour effectuées.

Finalement, ce travail de thèse décrit des mécanismes permettant de garantir formellement que chaque exigence affectée à un sous-système sera bien satisfaite par ce dernier, dans la limite définie par la spécification formelle du système et des sous-systèmes.

Il est à préciser que différentes études de cas d'envergure industrielle ont été menées afin d'éprouver et valider la méthode SysML/KAOS ainsi enrichie.

Méthodologie

En ce qui concerne la modélisation du domaine, le travail décrit dans cette thèse a débuté par une étude approfondie des approches existantes afin de recenser les forces et faiblesses des langages actuellement exploités pour la modélisation du domaine dans le contexte de l'ingénierie des exigences, ainsi que leur adéquation vis-à-vis des objectifs formulés. Il s'est ensuite agit de définir un langage de modélisation du domaine, compatible avec le langage de modélisation des exigences, et conforme aux objectifs formulés. Par la suite, une étude a été réalisée afin de définir des liens de correspondance entre le nouveau langage et *B System*. Il est à noter que chaque définition (langage et règles) a été spécifiée et vérifiée formellement afin de garantir la prise en compte des objectifs formulés. Les définitions du langage et des règles ont finalement été intégrées à la plateforme *Openflexo* [109] qui fédère les diverses contributions au sein du projet *FORMOSE* [17]. Transversalement, plusieurs études de cas ont été réalisées aux fins d'évaluation, de robustification et de validation : le langage de modélisation initialement défini a connu plusieurs ajustements, au fil des études de cas, afin de lui assurer une expressivité, une utilisabilité ainsi qu'une sémantique formelle suffisantes.

La méthodologie ainsi décrite se décline en :

- Une étude approfondie des approches de modélisation du domaine dans le sillage de l'ingénierie des exigences (Chapitre 2).
- La définition, informelle puis formelle, d'un langage de modélisation de domaine répondant aux objectifs formulés (Chapitres 2 et 5).
- La définition, informelle puis formelle, des liens de correspondance entre le langage de modélisation introduit et la méthode *B System* (Chapitres 3 et 5 pour les règles de traduction et Chapitres 4 et 5 pour les règles permettant la prise en compte des mises à jour effectuées au sein d'un modèle **B System**).
- La vérification formelle du langage et des liens de correspondance introduits (Chapitres 2, 4 et 5).
- L'implémentation du langage et des liens de correspondance au sein d'Openflexo (Chapitre 9).
- L'évaluation et la validation de la méthode sur des études de cas, en l'occurrence celles décrites aux Chapitres 7 et 8, certaines ayant conduit à des ajustements du langage et des règles définis (conception itérative).

Afin de définir les mécanismes permettant de garantir la satisfaction des exigences affectées à des sous-systèmes, il a été question :

- D'étudier les approches de décomposition d'un modèle formel (Chapitre 6).
- De définir une approche permettant la décomposition du modèle formel d'un système en plusieurs sous-modèles correspondant aux sous-systèmes, tout en tenant compte des exigences propres aux sous-systèmes, de manière à garantir la satisfaction des exigences affectées aux sous-systèmes (Chapitre 6).
- D'évaluer l'approche introduite sur des études de cas.

Contributions

Ce travail de thèse a conduit à la mise en oeuvre :

1. D'un langage de modélisation de domaine structuré et non ambigu (formalisé avec *Event-B*), supporté par un outil libre et compatible avec le langage de modélisation d'exigences de SysML/KAOS. Ce langage est fondé sur *OWL (Ontology Web Language)* [118] et *PLIB (Part Library)* [112] et permet de représenter le domaine à l'aide d'ontologies. En outre, il permet d'expliciter les éléments variables du modèle de domaine, les changements d'états de ces variables, à mesure que le système satisfait ses exigences, pouvant être représentés graphiquement au travers de diagrammes d'états-transitions à l'exemple des diagrammes d'états-transitions algébriques (ASTDs) [66]. La version initiale du langage (Chapitre 2) a été étendue et ajustée (Chapitre 5), à travers les études de cas.

   La proposition initiale, évaluée sur l'étude de cas *Landing Gear System (système de contrôle du train d'atterissage d'un avion)* [29], a fait l'objet d'un article accepté et publié [61] dans le cadre de la 7^{e} édition du workshop international *Model-Driven Requirements Engineering (MoDRE)* qui s'est tenu en marge de la 25^{e} édition de la conférence internationale *Requirements Engineering (RE)*. Une extension de l'article dans laquelle la proposition est évaluée sur une étude de cas liée à la spécification du composant de localisation du véhicule autonome *Cycab*, a fait l'objet d'une soumission pour parution dans un livre qui tient lieu de compte rendu des échanges du colloque international *NII Shonan*, organisé par l'*Institut National d'Informatique du Japon (NII)*, qui s'est tenu au *Japon* en Novembre 2016.

   La version ajustée du langage, évaluée sur l'étude de cas *Saturn* [129] (Chapitre 5), a pour sa part fait l'objet d'un article accepté et publié [60] dans le cadre de la 14^{e} édition de la conférence internationale *International Conference on Software Technologies (ICSOFT)*. Une réédition de cet article, rédigée en français, a fait l'objet d'un article accepté et publié [62] dans le cadre des 18^{e} journées

Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL). En outre, elle introduit et illustre l'utilisation des diagrammes d'états-transitions algébriques (ASTDs) [66] afin de représenter graphiquement les changements d'états des variables du modèle de domaine au fur et à mesure que le système satisfait ses buts.

2. Des règles, formellement vérifiées et supportées par un outil libre, permettant d'exploiter la modélisation du domaine pour obtenir automatiquement la partie structurelle d'une spécification *B System* issue de la formalisation de modèles de buts SysML/KAOS. L'outil *Rodin* [35] a été utilisé pour spécifier et vérifier formellement le langage et les règles, ce qui a fait l'objet d'un article accepté et publié [65] dans le cadre de la 6^{e} édition de la conférence internationale *ABZ (ASM, Alloy, B, TLA, VDM, Z)* qui s'est déroulée à Southampton, Royaume-Uni en juin 2018 (*ABZ2018*). Les annexes A et B présentent le contenu complet de la spécification formelle du langage et des règles.

   Le langage de modélisation introduit et les règles ont été évalués, conjointement au langage de modélisation des buts, dans le cadre de la spécification formelle des exigences du protocole de transport ferroviaire *hybrid ERTMS/ETCS level 3* [79] (Chapitre 7). Cette évaluation a fait l'objet d'un article accepté et publié [58] dans le cadre de la conférence ABZ2018. Sous invitation, une extension de l'article a fait l'objet d'une publication dans une édition du journal international *STTT (International Journal on Software Tools for Technology Transfer)* [135]. Le protocole *hybrid ERTMS/ETCS level 3* a également été spécifié, directement en Event-B, afin de mieux évaluer les avantages et limites inhérents à l'utilisation de la méthode SysML/KAOS. Cette approche classique de spécification d'un système avec Event-B a fait l'objet d'un article accepté et publié [96] dans le cadre de la conférence ABZ2018. Cet article a également fait l'objet d'une publication dans une édition du journal international *STTT*.

   La méthode SysML/KAOS, ainsi enrichie, a également été évaluée dans le cadre de la spécification formelle des exigences d'un système de transport urbain pour le compte de la *Ville de Montréal* (Chapitre 8). Cette évaluation a fait l'objet d'un article accepté et publié dans le cadre de la 21^{e} édition de la conférence internationale sur les méthodes formelles d'ingénierie *ICFEM (International Conference on Formal Engineering Methods)*.

3. Des règles, formellement vérifiées, contribuant à maintenir l'adéquation entre la partie structurelle d'une spécification *B System* et les modèles SysML/KAOS auxquels elle est associée. Ces règles, évaluées sur l'étude de cas *steam-boiler control specification problem* (problème de spécification du contrôleur d'une chaudière à vapeur) [22], ont fait l'objet d'un article accepté et publié [57] dans le cadre de la 23^{e} édition de la conférence internationale *ICECCS (International Conference on Engineering of Complex Computer Systems)*.
4. D'une approche permettant la décomposition du modèle formel d'un système en plusieurs sous-modèles correspondant aux sous-systèmes, tout en tenant compte des exigences propres aux sous-systèmes, de manière à garantir la satisfaction des exigences affectées aux sous-systèmes. Cette approche (Chapitre 6), évaluée sur l'étude de cas *steam-boiler control specification problem*, a fait l'objet d'un article accepté et publié [59] dans le cadre de la 14^{e} édition de la conférence internationale *iFM (integrated Formal Methods)*.

Plan de la thèse

Le Chapitre 1 présente succinctement les éléments fondamentaux sur lesquels repose ce travail de thèse. Par la suite, le Chapitre 2 introduit, justifie et illustre la version initiale du langage de modélisation de domaine tandis que les chapitres 3 et 4 se focalisent sur la définition, informelle puis formelle, et la vérification, des règles de correspondance entre modèles de domaine et spécifications *B System*. Le Chapitre 5 décrit quant à lui les principaux ajustements réalisés au sein du langage, au fil des études de cas. Finalement, tandis que le Chapitre 6 introduit la spécification et la vérification formelle des assignations d'exigences fonctionnelles, les Chapitres 7 et 8 décrivent les principales études de cas réalisées et le Chapitre 9 décrit succinctement l'outillage de la méthode SysML/KAOS construit sous *Openflexo*.

Il est à noter que les annexes A et B complètent les définitions informelle et formelle du langage de modélisation du domaine et des règles de correspondance. L'annexe C décrit quant à elle le scénario principal d'utilisation de l'outil Openflexo qui supporte la méthode SysML/KAOS.

Chapitre 1

Préliminaires

Résumé

Ce chapitre présente succinctement les éléments fondamentaux sur lesquels repose ce travail de thèse. Il s'agit en l'occurrence des méthodes *Event-B* et *B System* et des langages *SysML/KAOS* de modélisation des buts fonctionnels et non-fonctionnels. Le chapitre s'achève sur des définitions, générales puis contextuelles, de la notion d'ontologie.

1.1 Méthodes formelles

Pour Bjørner [26], une *spécification formelle* est un ensemble constitué d'une part de définitions de collections ou types et de leurs éléments, de fonctions et comportements et, d'autre part, des axiomes et obligations de preuve qui contraignent ces définitions. Chaque spécification formelle est construite en conformité avec les règles lexicales, syntaxiques et sémantiques d'un langage formel. Les contextes et principes de spécification sont quant à eux décrits par la méthode formelle associée au langage. Cette section présente succinctement les méthodes et langages formels sélectionnés dans le cadre du projet FORMOSE, en l'occurrence *Event-B* et *B System*, et considérés tout au long de ce travail de thèse.

1.1.1 Event-B

La méthode *B* est une méthode formelle proposée par *J. R. Abrial* pour la spécification, la vérification et la validation de logiciels critiques [10]. Elle repose sur la théorie des ensembles et la logique des prédicats et a permis la mise en place de systèmes d'envergure dans des domaines aussi divers que variés [23, 49]. L'approche utilisée permet de garantir un fonctionnement correct du logiciel spécifié et d'aboutir à une implémentation de ce dernier conforme à sa spécification. Elle est supportée par l'environnement de développement intégré *Atelier B* édité par *ClearSy* [42], un partenaire industriel au sein du projet FORMOSE [17].

Tout modèle *B* est construit de façon incrémentale, par raffinements successifs. Il est constitué de composants appelés *machines*. Chaque raffinement permet de concrétiser la spécification d'une machine dite abstraite au sein d'une autre dite concrète. La correction du raffinement est apportée par un ensemble d'obligations de preuve [10], la génération et le déchargement des obligations de preuve étant supportés par l'*Atelier B*.

Chaque machine *B* est constituée d'une partie *statique* et d'une partie *dynamique*. La partie statique décrit les types et éléments constants contraints par des propriétés. La partie dynamique décrit quant à elle les variables contraintes par des invariants ainsi que les opérations qui définissent les conditions de mise à jour de l'état des variables. D'autres obligations de preuve sont définies afin de garantir la non-violation des invariants à chaque exécution d'une opération. Chaque opération est caractérisée par une *précondition* qui est un préalable à son exécution et par une *postcondition* qui décrit l'impact de l'exécution sur l'état des variables. Le raffinement *B* préserve le nombre d'opérations définies au sein de la machine abstraite. De

plus, lorsqu'une opération `0c` définie au sein d'une machine concrète raffine une opération abstraite `0a`, il est nécessaire que la précondition de `0c` soit plus faible que celle de `0a` contrairement à la postcondition de `0c` qui peut être plus forte que celle de `0a`.

La méthode *Event-B* est quant à elle une méthode formelle utilisée pour la modélisation de systèmes critiques [8] ; un système étant défini comme un agglomérat d'éléments (matériels, logiciels, humains, etc.) en interaction suivant des règles bien précises. Elle repose sur les mêmes concepts mathématiques que la méthode *B* et a été utilisée dans de nombreux projets industriels pour la construction incrémentalle des spécifications formelles de systèmes et la vérification de propriétés [93]. Toutefois, la sémantique du langage Event-B repose sur le déclenchement d'évènements au sein d'un système tandis que celle de *B* repose sur l'exécution d'opérations au sein d'un logiciel. La méthode Event-B est supportée par l'environnement de développement intégré *Rodin* [35] qui permet tant l'édition et la validation des modèles Event-B que la génération et la décharge des obligations de preuve.

Un modèle Event-B comprend une partie statique définie au sein de *contextes* et une partie dynamique définie au sein de *machines*. Le contexte contient la définition des ensembles abstraits et énumérés, des constantes et de leurs propriétés. La machine, quant à elle, contient la définition des variables contraintes par des invariants et des évènements agissant sur l'état des variables. Chaque évènement est caractérisé par une garde qui est un préalable à son déclenchement et par une postcondition qui décrit l'impact du déclenchement sur l'état des variables. L'état initial des variables est défini par un évènement spécial appelé *évènement d'initialisation*. Un lien de raffinement défini entre une machine dite *abstraite* et une autre dite *concrète* permet à la machine concrète d'accéder au contenu de la machine abstraite afin d'enrichir ou concretiser la dynamique du système. De la même manière, un lien d'extension peut être défini entre deux contextes afin de permettre à l'un d'accéder au contenu de l'autre dans le but de les étendre. Il est enfin possible de préciser, au sein d'une machine, un ensemble de contextes afin de permettre à la machine d'accéder aux éléments qui y sont définis. Des invariants dits *de collage*, définis au sein d'une machine, permettent de caractériser la relation entre les variables introduites au sein de cette dernière et celles introduites dans des machines plus abstraites.

Contrairement à *B*, un raffinement Event-B peut augmenter le nombre d'évènements définis au sein de la machine abstraite. Chaque nouvel évènement défini est supposé raffiner un évènement spécial appelé *skip* qui est supposé maintenir l'état des variables abstraites inchangé. De plus, lorsqu'un évènement `Ec` défini au sein d'une machine concrète raffine un évènement abstrait `Ea`, il est nécessaire que les garde et postcondition de `Ec` ne soient pas plus faibles que les garde et postcondition de `Ea` respectivement.

1.1.2 B System

B System désigne une variante d'*Event-B* proposée au sein de l'environnement de développement intégré *Atelier B*. Les langages *B System* et Event-B partagent la même sémantique mais diffèrent par leurs syntaxes.

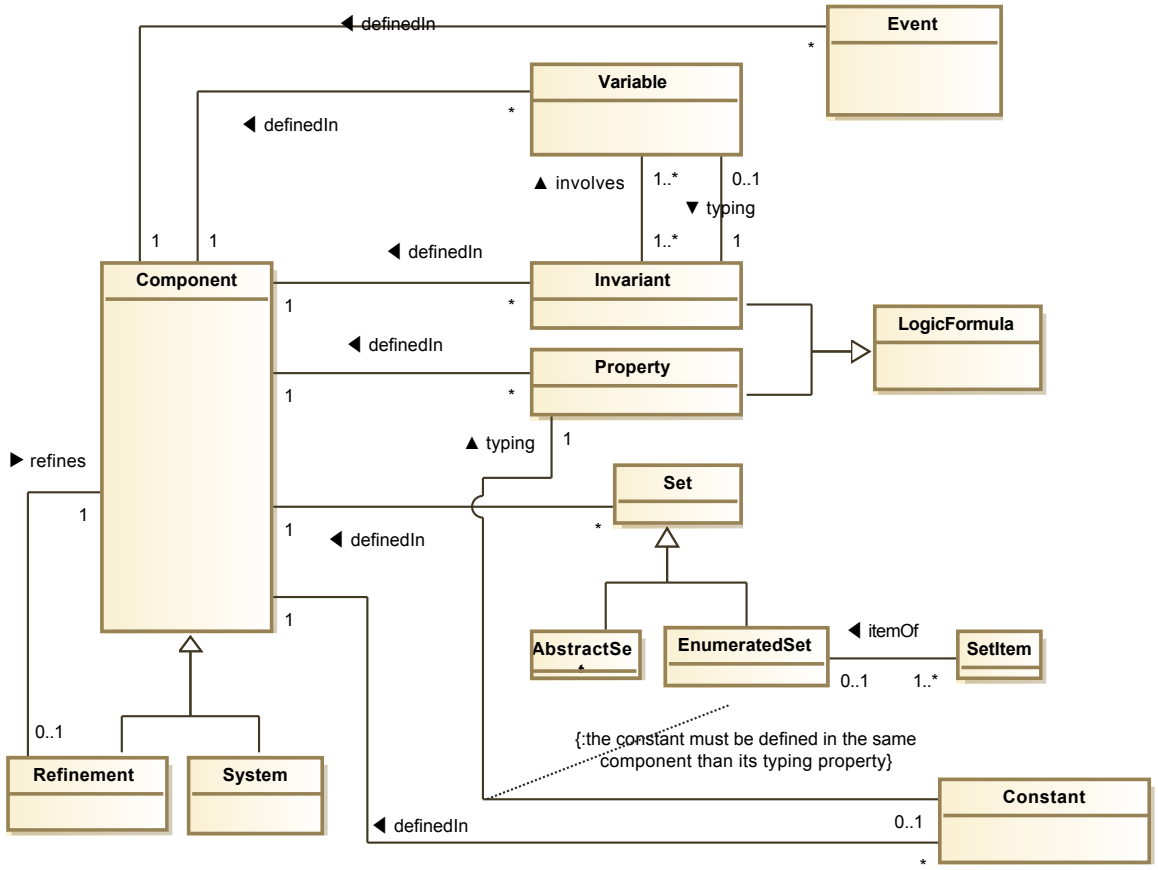


figure 1.1 – Éléments principaux du langage *B System* considérés dans le cadre du travail de thèse

La Figure 1.1 présente les principaux éléments du langage *B System* considérés dans le cadre du travail décrit dans cette thèse. Une spécification *B System* est constituée de composants. Un composant *B System* peut être un système ou un raffinement (s'il raffine un autre composant). De plus, chaque composant peut servir à définir des éléments de la partie statique (ensembles abstraits et énumérés, constantes et propriétés) ou des éléments de la partie dynamique (variables, invariant et évènements).

De la même manière qu'Event-B, chaque évènement *B System*
G $\widehat{=}$ **SELECT** *X* **WHERE** *G_Guard* **THEN** *Act* **END**
est caractérisé par sa garde *G_Guard*, qui représente la condition qui doit être vérifiée avant que G ne soit déclenché, et par sa post-condition *G_Post*, qui représente l'état du système après que l'action *Act* de G ait été effectuée.

1.2 SysML/KAOS

SysML/KAOS [70, 92] est une méthode formelle d'ingénierie des exigences. Elle permet initialement la modélisation, sous forme de buts, des exigences (1) fonctionnelles et (2) non-fonctionnelles d'un système. La formalisation du modèle des buts fonctionnels permet d'obtenir une spécification *B System* qui sert de base aux tâches de vérification et de validation formelles afin de détecter et corriger les potentielles incohérences.

1.2.1 Modélisation des exigences fonctionnelles

Une exigence fonctionnelle décrit un comportement attendu du système, à l'occurrence d'une condition précise. Le langage de modélisation des exigences fonctionnelles de SysML/KAOS [92] associe la traçabilité offerte par *SysML* [78] à l'expressivité du langage de modélisation d'exigences de *KAOS* [138]. Il permet la représentation des exigences fonctionnelles d'un système ainsi que des attentes vis-à-vis de l'environnement sous forme d'hiérarchies de buts. Parmi les opérateurs intervenant dans la hiérarchisation des buts, on distingue l'opérateur ***And*** (*Et*), l'opérateur ***Or*** (*Ou*) et l'opérateur ***Milestone*** (*Séquence*). L'opérateur *And* apparaît lorsque la condition nécessaire et suffisante, pour la réalisation d'un but, est la réalisation de chacun de ses sous-buts. Lorsque la condition nécessaire et suffisante pour la réalisation d'un but se limite à la réalisation de l'un de ses sous-buts, alors c'est l'opérateur *Or* qui apparaît. L'opérateur *Milestone* permet quant à lui de séquencer un ensemble de sous-buts dont la réalisation ordonnée est nécessaire pour garantir la satisfaction du but parent. SysML/KAOS considère également le *raffinement de données* qui intervient lorsque des buts apparaissant dans un niveau de raffinement sont réexprimés, au sein d'un niveau de raffinement subséquent, du fait du raffinement de certains éléments de données intervenant dans leur spécification.

Pour tenir compte de la complexité des systèmes, la méthode SysML/KAOS considère que le "premier" diagramme de buts fonctionnels construit, ou diagramme de plus haut niveau, est celui du système principal. La décomposition en sous-buts prend fin lorsqu'il est possible d'affecter chaque but de plus bas niveau, dit but

élémentaire, à un composant ou agent du système ou de l'environnement (sous-système). Par la suite, au besoin, des diagrammes de buts peuvent être définis pour les différents sous-systèmes. Ceux-ci peuvent en outre comporter des buts propres, en plus de ceux provenant du système de niveau supérieur.

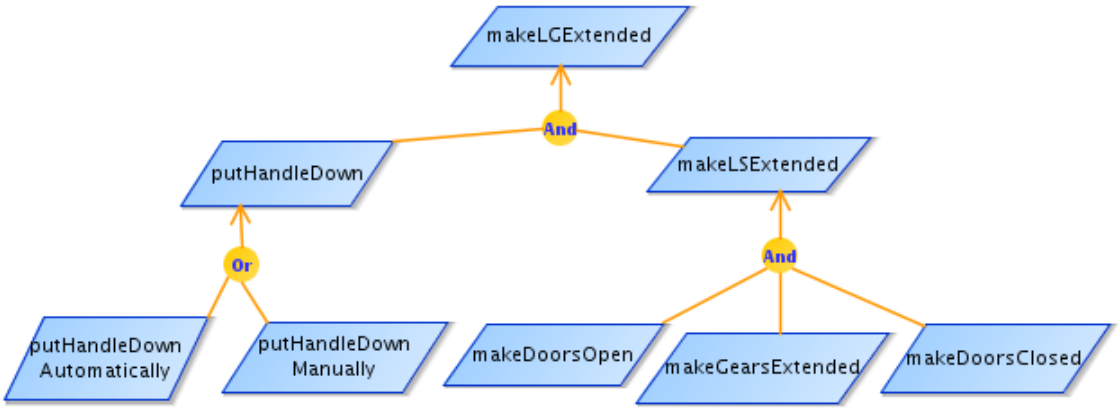


figure 1.2 – Extrait du diagramme des buts fonctionnels d'un système de gestion du train d'atterrissage d'un avion

La Figure 1.2 illustre l'utilisation du langage de modélisation des buts fonctionnels de SysML/KAOS sur une étude de cas intitulée *Landing Gear System* [29] proposée dans le cadre de la 4^e édition de la conférence *ABZ (ASM, Alloy, B, TLA, VDM, Z)*. L'objectif de l'étude de cas est de spécifier un système en charge de l'extension et de la rétraction du train d'atterrissage d'un avion. Le diagramme de la Figure 1.2 est axé sur l'objectif fonctionnel d'extension du train d'atterrissage (but fonctionnel *makeLGExtended*). L'opérateur de raffinement *And* est utilisé afin de spécifier les sous-buts à satisfaire pour garantir la satisfaction du but parent : pour satisfaire l'extension du train d'atterrissage, il faut abaisser la poignée de commande (but *putHandleDown*) et effectuer l'extension (but *makeLSExtended*). De même, la satisfaction du but *makeLSExtended* passe par l'ouverture de la porte du train d'atterrissage (but *makeDoorsOpen*), par l'extension du dispositif physique train d'atterrissage (but *makeGearsExtended*) et par la fermeture de la porte (but *makeDoorsClosed)*. Par contre, l'abaissement de la poignée de commande peut se faire automatiquement (but *putHandleDownAutomatically*), par un automate système, ou manuellement (but *putHandleDownManually*), par le pilote.

1.2.2 Formalisation des buts fonctionnels

La formalisation des modèles de buts fonctionnels SysML/KAOS est décrite dans [102]. Les règles proposées permettent de générer un modèle *B System* dont la structure réflète la hiérarchie du modèle des buts fonctionnels : un composant est associé à chaque niveau de raffinement de la hiérarchie, ce composant définissant le squelette d'un évènement pour chaque but du niveau de raffinement. La méthode *B System* est choisie dans le cadre de la formalisation des modèles de buts fonctionnels SysML/KAOS car, contrairement à *B*, elle permet de modéliser les évènements qui ponctuent le cycle de vie d'un système et auxquels les buts fonctionnels correspondent naturellement. De plus, le raffinement *B System* permet l'ajout de nouveaux évènements de la même manière que le raffinement SysML/KAOS fait apparaître de nouveaux buts. Par ailleurs, *B System* est supportée par l'*Atelier B* édité par *ClearSy* [42], un partenaire industriel au sein du projet FORMOSE [17]. Bien plus, une simple réécriture syntaxique permet de convertir une spécification *B System* en spécification *Event-B* afin de profiter également de l'outillage offert par la plateforme *Rodin* [35].

En *B System*, la sémantique des liens de raffinement entre buts est exprimée par de nouvelles obligations de preuve, qui sont fonction des opérateurs de raffinement utilisés, et qui complémentent les obligations de preuve classiques de *préservation d'invariant* et de *faisabilité d'action* définies dans [8]. Par exemple, pour un but G se décomposant en deux sous-buts G_1 et G_2, les obligations de preuve sont [1] :

- Dans le cas d'un raffinement *And* (les variables intervenant dans la spécification des sous-buts doivent être distinctes) :
  - $G_1\_Guard \Rightarrow G\_Guard$: si la garde de G_1 est vraie, alors la garde de G doit l'être aussi.
  - $G_2\_Guard \Rightarrow G\_Guard$
  - $(G_1\_Post \wedge G_2\_Post) \Rightarrow G\_Post$: si la conjonction des post-conditions de G_1 et G_2 est vraie, alors la post-condition de G doit l'être aussi.
- Dans le cas d'un raffinement *Or* :
  - $G_1\_Guard \Rightarrow G\_Guard$
  - $G_2\_Guard \Rightarrow G\_Guard$
  - $G_1\_Post \Rightarrow G\_Post$
  - $G_2\_Post \Rightarrow G\_Post$
  - $(G_1\_Guard \wedge G_1\_Post) \Rightarrow \neg G_2\_Guard$: la satisfaction de G_1 ne doit pas conduire le système dans un état où G_2 peut être déclenché.
  - $(G_2\_Guard \wedge G_2\_Post) \Rightarrow \neg G_1\_Guard$
- Dans le cas d'un raffinement *Milestone* :
  - $G_1\_Guard \Rightarrow G\_Guard$
  - $G_2\_Post \Rightarrow G\_Post$

1. pour un évènement G, G_Guard représente la garde de G et G_Post représente sa post-condition

  - $\Box((G_1\_Guard \wedge G_1\_Post) \Rightarrow \Diamond G2\_Guard)$: la satisfaction de G_1 doit être suivie, directement ou indirectement, par le déclenchement de G_2.
- Dans le cas d'un raffinement **de données**, les obligations de preuve correspondent aux obligations de preuve classiques de renforcement de la garde et de simulation de l'action [8].

1.2.3 Modélisation des exigences non-fonctionnelles

Une exigence non-fonctionnelle désigne une propriété ou une caractérisation du système [69]. Elle permet de définir des contraintes sur la façon avec laquelle le système atteint ses objectifs.

La méthode SysML/KAOS représente les exigences non-fonctionnelles à travers un langage similaire à celui utilisé pour la représentation des exigences fonctionnelles [68,71] et qui réutilise des notions du *NFR Framework* [38]. Ainsi, la hiérarchie des buts non-fonctionnels est construite par raffinements successifs à l'aide des opérateurs de raffinement *And* et *Or*. Toutefois, cette hiérarchie est construite au sein d'un modèle distinct de celui qui structure les buts fonctionnels. Chaque but non-fonctionnel est représenté sous la forme ***NFRType[Sujet]*** où ***NFRType*** désigne le type de la contrainte définie par le but (sécurité, sureté, etc.) et ***Sujet*** désigne l'entité du système ciblée par la contrainte. Un but ***NFRType[Sujet]*** peut être raffiné soit par les sous-buts $\boldsymbol{NFRType_i[Sujet]}$ (*raffinement par type*) ou par les sous-buts $\boldsymbol{NFRType[Sujet_i]}$ (*raffinement par sujet*), sachant que $\boldsymbol{NFRType_i}$ est un sous-type de ***NFRType*** et $\boldsymbol{Sujet_i}$ est une sous-entité de ***Sujet***. Par exemple, le but non-fonctionnel ***Sécurité[Système]*** peut être raffiné par les sous-buts ***Confidentialité[Système]***, ***Intégrité[Système]*** et ***Disponibilité[Système]*** en conformité avec la taxonomie des types de buts non-fonctionnels [38]. Il s'agit là d'un raffinement par type. Un raffinement par sujet du but ***Sécurité[Système]*** produirait les sous-buts ***Sécurité[Hardware]*** et ***Sécurité[Software]*** pour un système constitué d'une partie matérielle (Hardware) et d'une partie logicielle (Software). Le processus de raffinement prend fin lorsqu'il est possible de proposer des solutions de satisfaction, appelées *buts de contribution*, aux buts feuilles du modèle des buts non-fonctionnels.

Chaque but de contribution identifié peut contribuer *positivement* (+) ou *négativement* (-) à la satisfaction d'un but non-fonctionnel. De même, chaque but de contribution peut avoir un impact *positif* (+) ou *négatif* (-) sur la satisfaction d'un but fonctionnel. Les impacts des buts de contribution sont représentés au sein d'un modèle distinct appelé *modèle intégré* qui fédère les modèles de buts fonctionnels et non-fonctionnels [68]. Ils peuvent représenter (1) une contrainte de raffinement d'un but fonctionnel, (2) l'introduction d'un but fonctionnel ou (3) une contrainte sur la façon avec laquelle un but fonctionnel élémentaire est satisfait par l'agent à qui il est assigné.

1.3 Ontologies

Dans son article [75], Gruber définit une ontologie comme une spécification explicite d'une conceptualisation. Bjørner [26] quant à lui définit une ontologie comme une représentation formelle des catégories, propriétés et relations caractérisant des entités d'un ou plusieurs domaines. Il est à noter qu'une entité désigne un phénomène ou un élément descriptible d'un domaine.

Les ontologies sont principalement utilisées pour représenter la connaissance d'un domaine. Suivant le formalisme de définition adopté, il est possible de bénéficier de mécanismes automatiques d'inférence (déduction de nouvelles assertions) et de vérification (contrôle de la cohérence). Les ontologies se répartissent en trois grandes catégories : (1) les **ontologies fondamentales** qui sont des ontologies de haut-niveau indépendantes de tout domaine ; (2) les **ontologies de domaine** qui se restreignent à un domaine donné ; et (3) les **ontologies d'application** définies dans le contexte d'une application particulière. En représentation de connaissances, l'objectif d'une ontologie de domaine est de permettre l'interopérabilité sémantique entre plusieurs systèmes opérant au sein du même domaine : permettre des échanges d'informations entre différents systèmes de telle sorte que le sens d'une information produite par un système puisse être automatiquement inféré par tout autre système de façon à la rendre exploitable par ce dernier [20]. Une telle ontologie peut être conçue ou interprétée sous l'hypothèse *Closed World Assumption (CWA)* ou sous l'hypothèse *Open World Assumption (OWA)*. L'hypothèse *CWA* est considérée lorsque tout fait ne se déduisant pas de l'ontologie est supposé faux jusqu'à ce qu'il soit explicitement déclaré vrai. En ce qui concerne l'hypothèse *OWA*, un fait n'est considéré comme étant faux que s'il est possible de déduire son invalidité à partir de l'ontologie.

Dans le cadre de ce travail, les ontologies sont exploitées non pas pour représenter de la connaissance, comme par exemple dans le cas du web sémantique, ou pour formaliser le sens des choses, mais pour modéliser les domaines de systèmes d'ingénierie. Une conséquence immédiate de cette distinction est que chaque ontologie doit permettre de distinguer les entités dynamiques du domaine, dont l'état est succeptible d'être modifié par action du système, des entités statiques. L'ontologie désigne alors un modèle formel représentant des entités d'un domaine (en l'occurrence le système et son environnement dans le cas de systèmes ouverts ou le système tout court dans le cas de systèmes fermés), pouvant être regroupées en catégories à travers des relations de généralisation/spécialisation, leurs instances, leurs contraintes et attributs ainsi que les relations existantes entre elles. Un attribut définit une caractéristique objectivement mesurable d'une entité [26]. Pour Jackson

[84], seul un attribut peut être déclaré dynamique : l'attribut est dit dynamique lorsque sa valeur est succeptible de varier. Les termes *classes* ou *concepts* peuvent être utilisés pour désigner des entités et le terme *individu* pour désigner une instance d'entité.

Première partie

Modélisation du domaine

Chapitre 2

Langage SysML/KAOS de modélisation du domaine

Résumé

Un moyen de construire des systèmes critiques sûrs consiste à modéliser formellement les exigences formulées par les parties prenantes et à assurer leur cohérence en tenant compte des caractéristiques du domaine d'application.

Ce chapitre enrichit la méthode SysML/KAOS en introduisant un langage de spécification d'ontologies, défini par son métamodèle, pour la modélisation du domaine d'application d'un système dont les exigences sont capturées au moyen des langages de buts de SysML/KAOS. Il est construit à partir d'*OWL* et *PLIB*. L'explicitation et la vérification formelles de la sémantique des langages SysML/KAOS se font à travers la méthode *B System* : les modèles de buts fournissent les composants et la partie comportementale (évènements) de la spécification *B System*, tandis que les modèles de domaine fournissent sa partie structurelle.

La proposition est illustrée à travers une étude de cas portant sur la spécification du composant de localisation du véhicule autonome *Cycab*.

Commentaires

La contribution ici réside dans la définition, au sein de la méthode SysML/-KAOS, d'un langage de spécification d'ontologies pour la modélisation du domaine d'application d'un système. Le langage est défini de façon à (i) garantir sa compatibilité avec les langages SysML/KAOS de modélisation des exigences et (ii) assurer que tout modèle de domaine pourra être exploité afin de produire la partie structurelle de la spécification *B System* issue de la formalisation des modèles d'exigences SysML/KAOS.

La proposition, évaluée sur l'étude de cas *Landing Gear System (système de contrôle du train d'atterissage d'un avion)* [29], a fait l'objet d'un article accepté et publié [61] dans le cadre de la 7^{e} édition du workshop international *Model-Driven Requirements Engineering (MoDRE)* qui s'est tenu en marge de la 25^{e} édition de la conférence internationale *Requirements Engineering (RE)* qui s'est déroulée à Lisbon, Portugal en septembre 2017. Le contenu de ce chapitre est une extension de cet article dans laquelle la proposition est évaluée sur une étude de cas liée à la spécification du composant de localisation du véhicule autonome *Cycab*. Cette extension a fait l'objet d'une soumission pour parution dans un livre qui tient lieu de compte rendu des échanges du colloque international *NII Shonan*, organisé par l'*Institut National d'Informatique du Japon (NII)*, qui s'est tenu au *Japon* en Novembre 2016.

Cette contribution et les articles sus-cités ont été élaborés par mes soins en tenant compte des remarques et commentaires issus de mon équipe d'encadrement.

Integrating Domain Knowledge in Formal Requirements Engineering

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

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

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

Marc Frappier
Département d'informatique,
Université de Sherbrooke, Sherbrooke, QC J1K 2R1, Canada
Marc.Frappier@usherbrooke.ca

Keywords: Requirements Engineering, Formal Models, Domain Modeling, *SysML/KAOS*, *B System*, *Event-B*, Autonomous Vehicle, *Cycab*

Abstract

One way to build safe critical systems is to formally model the requirements formulated by stakeholders and to ensure their consistency with respect to domain properties. This paper describes a metamodel for a domain modeling language built from *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 specified, verified and validated using the *Event-B* method. Goal models provide machines and events of the Event-B specification while domain models provide its structural part (sets and constants with their properties and variables with their invariant). Our proposal is illustrated with a case study dealing with the specification of a localization component for an autonomous vehicle.

2.1 Introduction

Computer science is a relatively young science, but it does not prevent it from tackling huge challenges such as implementation of critical and complex software or cyberphysical systems. Such systems require careful analysis and design to ensure they do not cause disasters. Literature is full of disasters caused by failures at one of these stages [138]. The purpose of the *ANR FORMOSE* project [17] 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 according to the defined requirements engineering method requires the representation of its requirements as well as of entities and 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 [92] focuses on modeling of functional and non-functional requirements through goal hierarchies. Furthermore, Matoussi *et al.* [102] report on the explicit representation of the semantics of SysML/KAOS goal models with *Event-B* [8].

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 and describe the defined SysML/KAOS domain modeling language. The illustration is performed on *TACOS* [16], 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 [117], an autonomous ground transportation system.

The remainder of this paper is structured as follows: Section 2 briefly describes Event-B and SysML/KAOS. Section 3 summarises existing work [98, 102] on the explicit representation of the semantics of SysML/KAOS models. Section 4 presents the relevant state of the art on domain modeling in requirements engineering and defines our expectations regarding the SysML/KAOS domain modeling language. Finally, Section 5 describes and illustrates the domain modeling language while Section 6 reports our conclusions and discusses future work.

2.2 Background

2.2.1 Event-B and B System

Event-B [8] 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 a static part called *context* and a dynamic part called *machine*. 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 [8].

Through this paper, we use *B System* [41], a variant of Event-B proposed by *ClearSy*, an industrial partner in the *FORMOSE* project, in its integrated development environment *Atelier B* [21]. *B System* and Event-B share the same semantics but are syntactically different.

2.2.2 SysML/KAOS

SysML/KAOS [92] is a requirements engineering method which combines the traceability provided by *SysML* [78] with goal expressiveness provided by KAOS [138]. 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 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.

KAOS captures domain entities and properties within a model called the **object model** which is a *UML* class diagram. Its expressiveness is however considered insufficient by *FORMOSE* industrial partners [17], regarding the complexity and the criticality of the systems of interest.

Within SysML/KAOS, a functional goal describes the *expected behaviour* of the system once a certain condition holds [98]: *[if **CurrentCondition** then] sooner-or-later **TargetCondition***. A functional goal can also be defined without specifying a *CurrentCondition*. In this case, the expected behaviour can be observed from any system state.

Figure 2.1 represents a SysML/KAOS goal diagram for the *Cycab* localization component. Its main purpose is vehicle localization.

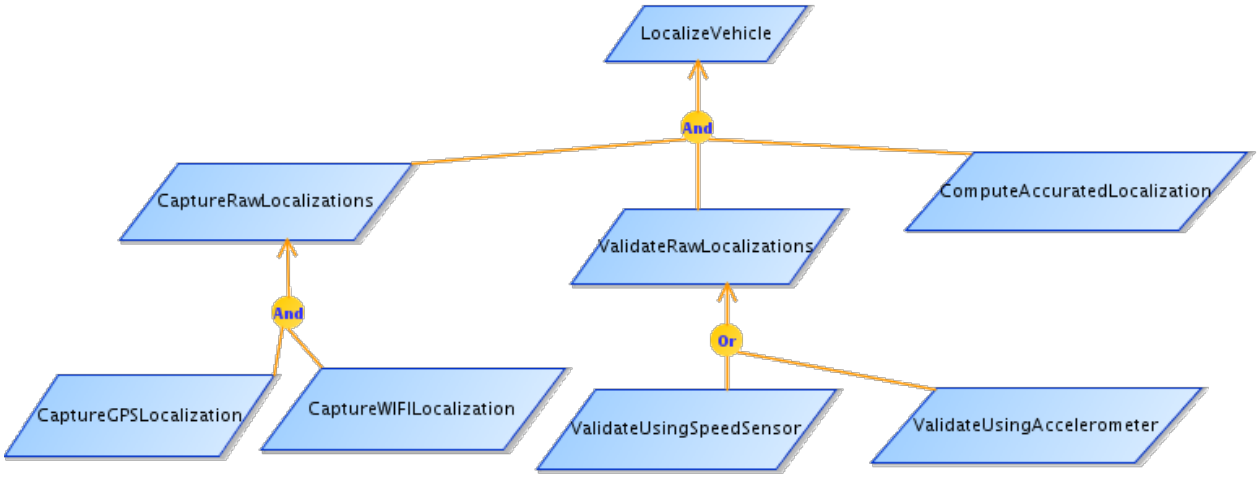


Figure 2.1 – Excerpt from the localization component goal diagram

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**).

2.3 B System Explicitation of the Semantics of SysML/KAOS Models

2.3.1 Semantics of Goal Models

The formalisation of SysML/KAOS goal models is the purpose of the work of Matoussi *et al.* [102]. Each refinement level of a goal diagram gives a *B System* component. 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 [8]. The other classic proof obligations are not relevant for our purpose [102]. 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 2.2 and 2.3 represent the *B System* components obtained respectively from the root level of the goal diagram of Fig. 2.1 and from its first refinement level. The structural part of the *B System* specification (constants constrained by properties and variables constrained by an invariant) and the body of events must be manually provided. The objective of our study is to automatically derive the structural part 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:

$$CaptureRawlocalizations\_Guard \Rightarrow LocalizeVehicle\_Guard \tag{2.1}$$

$$ValidateRawlocalizations\_Guard \Rightarrow LocalizeVehicle\_Guard \tag{2.2}$$

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

$$ComputeAccuratedlocalization\_Guard \Rightarrow LocalizeVehicle\_Guard \quad (2.3)$$

$$CaptureRawlocalizations\_PostCondition \wedge ValidateRawlocalizations\_PostCondition \wedge ComputeAccuratedlocalization\_PostCondition \Rightarrow LocalizeVehicle\_PostCondition \quad (2.4)$$

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

Figure 2.2 – Formalisation of the root level of the goal diagram of Fig. 2.1

2.3.2 Towards a Formal Expression of the Semantics of Domain Models

A domain model is a conceptual model capturing the topics related to a specific problem domain [30]. 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 [24], *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 [113], Pierra defines a domain model 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.

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

Figure 2.3 – Formalisation of the first refinement level of the goal diagram of Fig. 2.1

A first attempt at modeling domains within SysML/KAOS is achieved in [98]. Domain modeling involves *UML* class diagrams, *UML* object diagrams and ontologies. The case study presented reveals the use of ontologies for domain knowledge representation; 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 to translate some domain model elements to Event-B. However, the proposal involves *UML* diagrams which are semi-formal graphical representations [103, 106]. Moreover, it uses several languages which is an extra source of complexity.

2.4 State of the Art on Domain Modeling in Requirements Engineering

2.4.1 Existing Domain Modeling Approaches

In *KAOS* [138], 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 [103], is difficult to exploit in case of critical systems [106].

In [51], Devedzic proposes to model the domain knowledge through either formulae of first-order logic or ontologies. He considers ontologies as a more structured and extensible representation of domain knowledge.

In [89], 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 [106], 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 [120].

In [50], Dermeval *et al.* proposes a systematic literature review 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.

2.4.2 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 within 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 within the ontology. As [15], 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* [15] because our domain models must be translatable to Event-B specifications.

Several ontology modeling languages exist. The main ones are ***OWL (Ontology Web Language)*** [118], ***PLIB (Part LIBrary)*** [112] and ***F-Logic (Frame Logic)*** [88]. A summary of similarities and differences between these languages is described in Table 2.1:

- *PLIB*, *OWL* and *F-Logic* implement modularisation mechanisms. *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 *F-Logic* use the total import: when an ontology *A* refers to an ontology *B*, all 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 multiple inheritance and instantiation while *PLIB* implements 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 several representations or view points for a concept while neither *OWL* nor *F-Logic* do.

Table 2.1 – 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

— The knowledge modeled using *OWL*, *PLIB* and *F-Logic* is always considered static because there is no distinguishing mechanism. It is for instance impossible to specify that the localization of a vehicle can change dynamically while its brand cannot.

As stated in [143], *all the studied languages emphasize more on modeling static domain knowledge*. None of these languages allows to specify that knowledge described must remain unchanged or that it is likely to be updated. Moreover, none of the languages fully meets our requirements. For instance, *OWL* assumes the *OWA* assumption, *PLIB* is weakly expressive, etc. The most aligned are *OWL* and *PLIB*.

2.5 Our Approach for Domain Modeling

We choose 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, to represent the domain of a system whose requirements are captured using the SysML/KAOS method. The domain

modeling language makes the *Unique Name Assumption (UNA)* [15]: the name of an element is sufficient to uniquely identify it among all others. Furthermore, the metamodel is designed to allow the specification of knowledge that is likely to evolve over time.

2.5.1 Presentation

Figures 2.4, 2.5, 2.6 and 2.7 present the main part of the metamodel associated with the SysML/KAOS domain modeling language. The yellow elements are those that have an equivalence in *OWL*, while the red ones are the ones that have been inserted or customized. In addition, some constraints and associations, such as the *parentConcept* association, come from the *PLIB* metamodel. Due to space consideration, we will not highlight all the elements and constraints of the metamodel.

Concepts and Individuals, Data Sets and Data Values

Domain models are built around instances of Concept which represent sets of individuals sharing common characteristics (Fig. 2.4). 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. It should be noted that when a variable concept `C0` is a subconcept of another variable concept `PC0`, the set of elements that `C0` can contain, over its whole existence, is included in the set of elements that `PC0` can contain. However, this version of the domain modeling language allows that, at some point, because of the variability of `C0` and `PC0`, an element present in `C0` is not present in `PC0`. The adjusted version of the domain modeling language considers a different approach in which inclusion of a variable concept into another one implies that at any point, elements of the variable subconcept must be elements of the variable parent concept.

Data sets (instances of DataSet) are used to group data values (instances of DataValue) having the same type (Fig. 2.5). Default data sets are `INTEGER`, `NATURAL` for positive integers, `FLOAT`, `STRING` or `BOOL` for booleans. The easiest way to build a data set is to list its elements. This can be done by defining instances of EnumeratedDataSet.

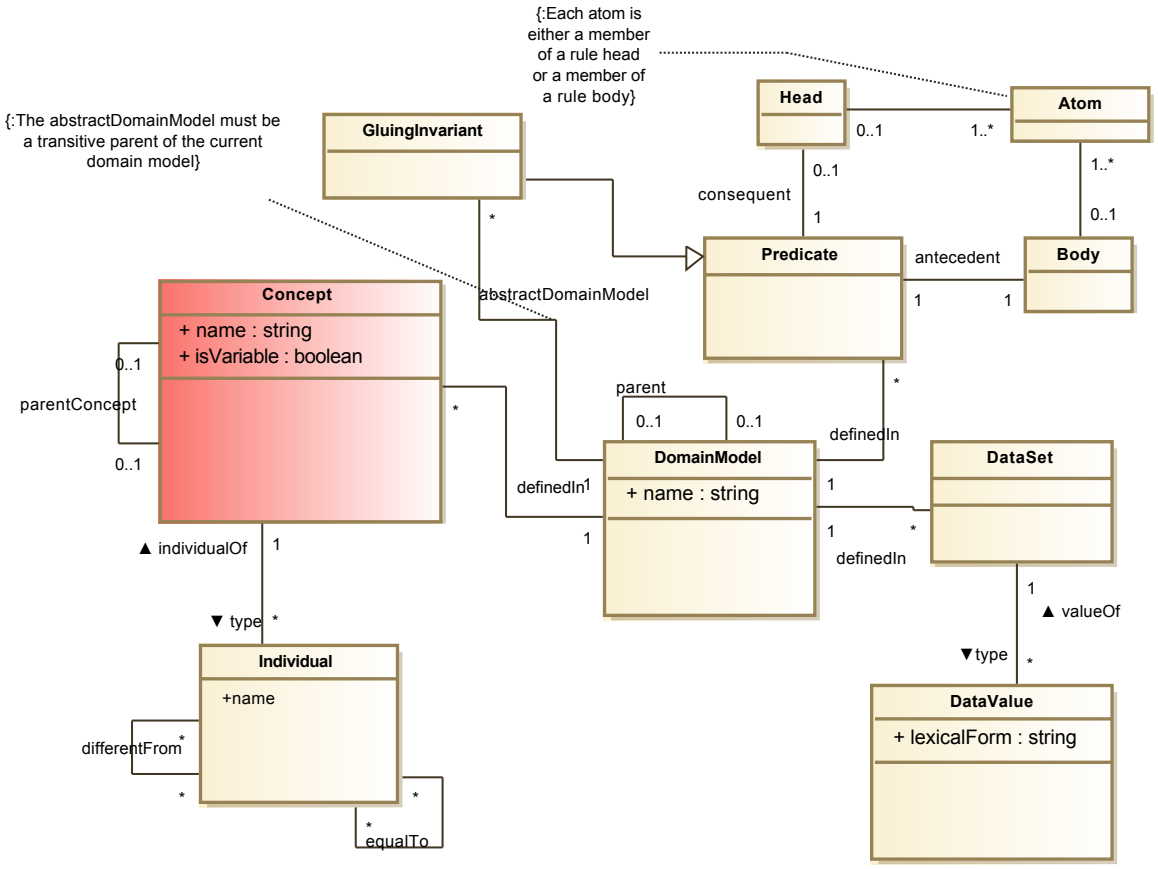


Figure 2.4 – First part of the metamodel associated with the domain modeling language

Relations and Attributes

Relations (instances of Relation) are used to capture links between concepts (Fig. 2.6) while attributes (instances of Attribute) capture links between concepts and data sets (Fig. 2.7). A relation (Fig. 2.6) or an attribute (Fig. 2.7) can be *variable* if its set of maplets can be updated through addition or deletion. Otherwise, it is *constant*. Relations are characterized by their *cardinalities*: DomainCardinality and RangeCardinality (Fig. 2.6). Each instance of DomainCardinality (respectively RangeCardinality) makes it possible to define, for a relation `re`, the minimum and maximum limits of the number of individuals, having the domain (respectively range) of `re` as *type*, that can be put in relation with one individual, having the range (respectively domain) of `re` as *type*. The following constraints are associated with these limits: $(minCardinality \geq 0) \wedge (maxCardinality = \infty \vee maxCardinality \geq$

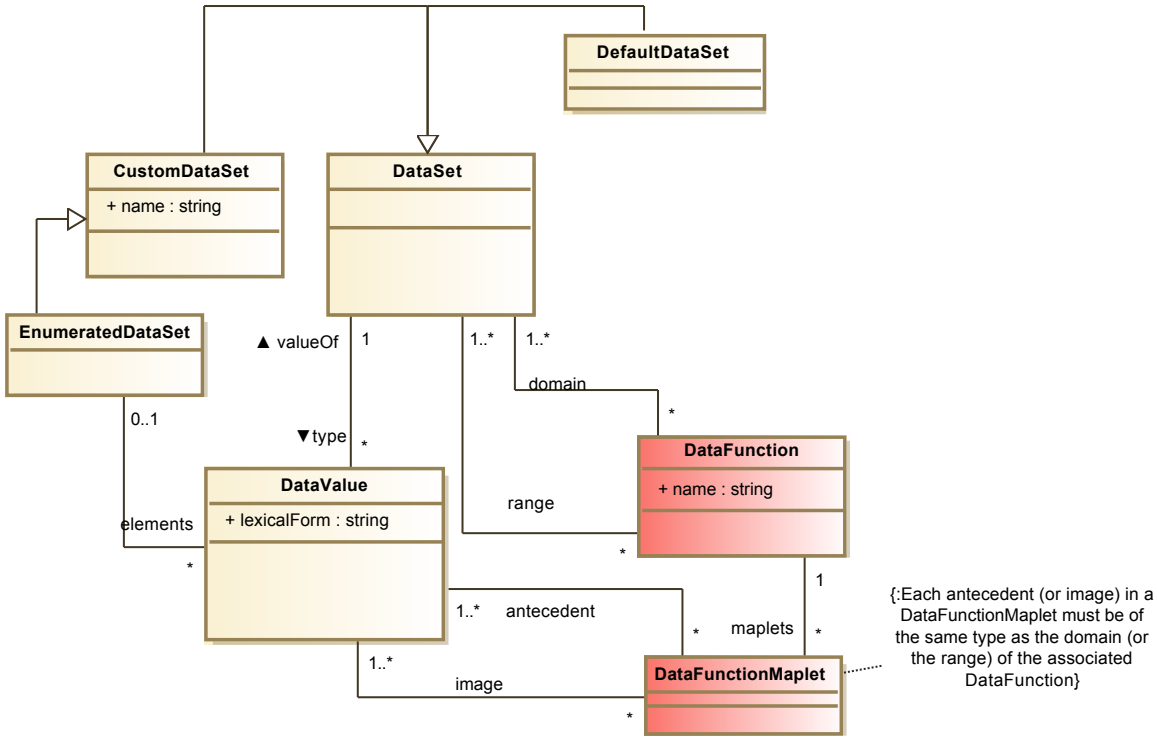


Figure 2.5 – Fourth part of the metamodel associated with the domain modeling language

minCardinality), knowing that if $maxCardinality = \infty$, then there is no maximum limit. Relation maplets (instances of RelationMaplet) define associations between individuals through relations. In an identical manner, attribute maplets (instances of AttributeMaplet) define associations between individuals and data values through attributes.

Optional characteristics can be specified for a relation (Fig. 2.6): *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 said to be

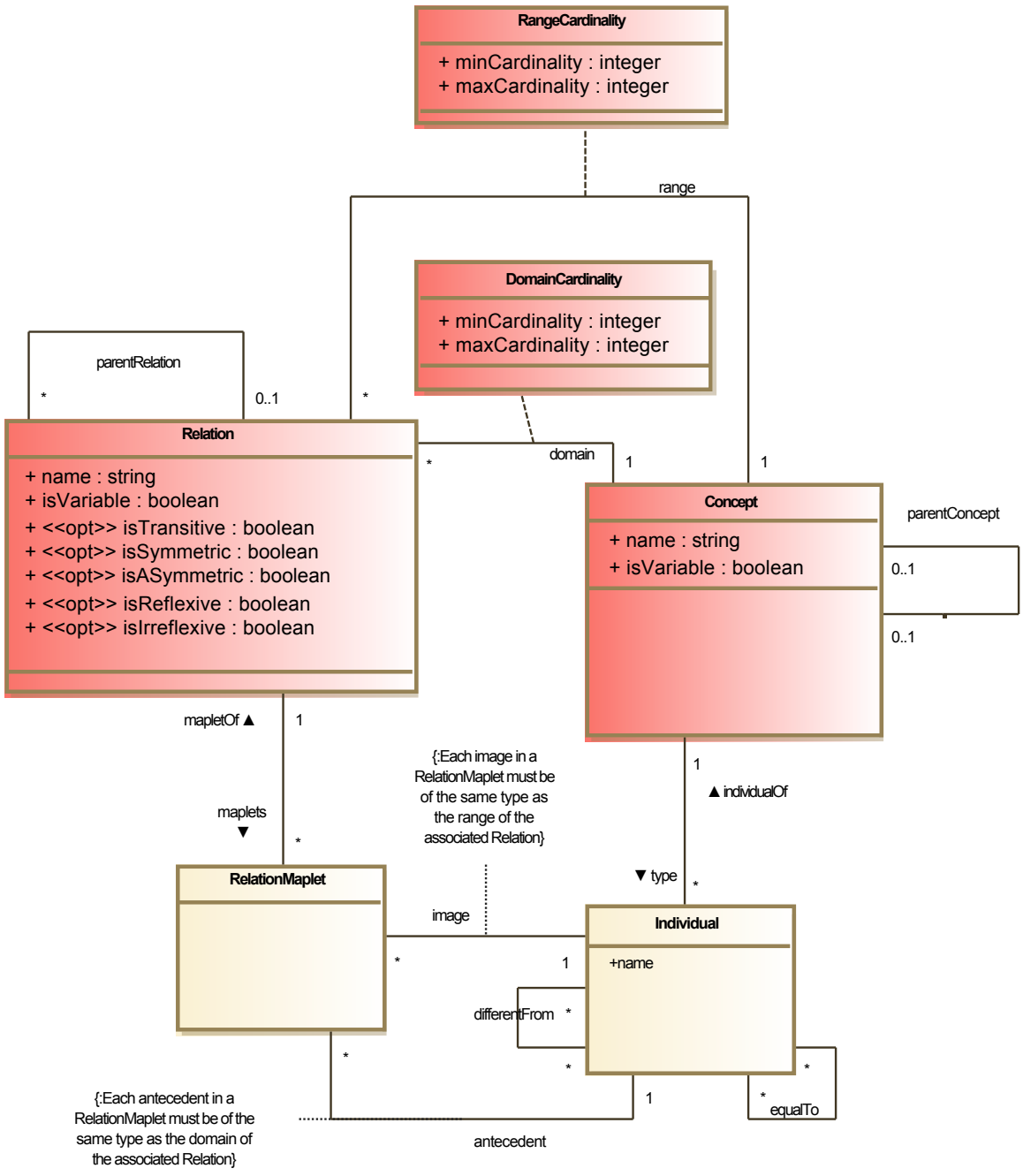


Figure 2.6 – Second part of the metamodel associated with the domain modeling language

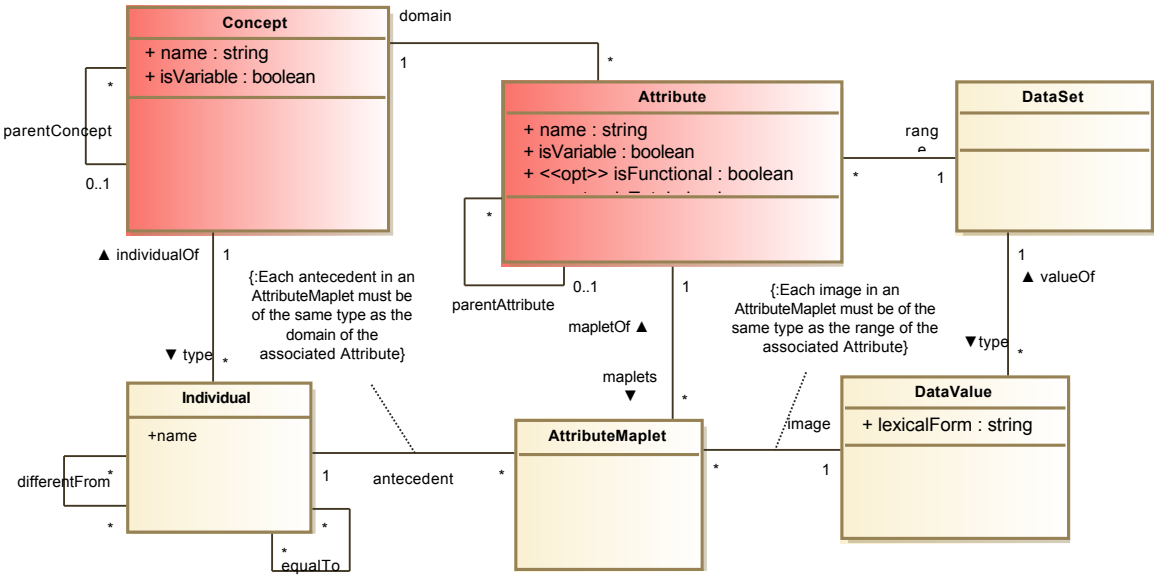


Figure 2.7 – Third part of the metamodel associated with the domain modeling language

irreflexive when it does not authorize any association 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.

Functions and Predicates

Data functions (Instances of DataFunction) (Fig. 2.5) 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***: a data function named `multiply` can be defined to produce, given two integers (individuals of `INTEGER`) `x` and `y`, the integer representing $x * y$. On the other side, predicates (instances of Predicate) (Fig. 2.4) 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* defines the type of a term: ConceptAtom for individuals and DataSetAtom for data values (Fig. 2.8). An *association atom* defines an association between terms: RelationAtom for associations through instances of Relation, AttributeAtom for

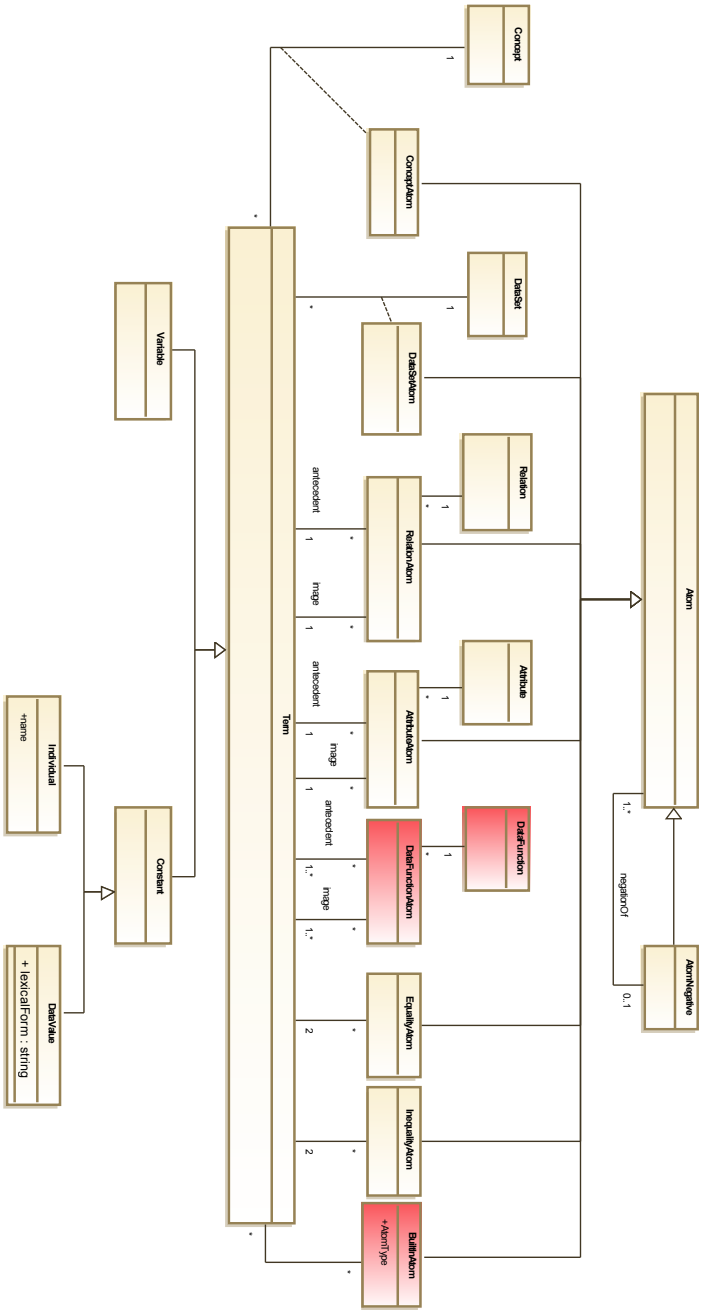


Figure 2.8 – Fifth part of the metamodel associated with the domain modeling language

associations through instances of Attribute and DataFunctionAtom for associations through instances of DataFunction (Fig. 2.8). For each case, types of the related terms must correspond to domains/ranges of the considered link. A *comparison atom* defines comparison relationships between terms: EqualityAtom for equality and InequalityAtom for difference (Fig. 2.8). Built in atoms are specialized atoms, characterized by identifiers captured through the AtomType enumeration, and used to represent special constraints between terms (Fig. 2.8) such as arithmetic constraints between several integers (eg: $a + b < c$). Predicates can also be used to represent constraints required for *parameterized/dependent* relations or attributes. For example, knowing that each material resistance depends on medium temperature, resistance and temperature are dependent attributes.

Domain Model and Goal Model

Each domain model is associated with a refinement level of the SysML/KAOS functional goal model and can have, as its parent, another domain model (Fig. 2.4). This allows the child domain model to access and extend some elements defined within the parent domain model. It should be noted that the parent domain model must be associated with the refinement level directly above the one to which the child domain model is associated.

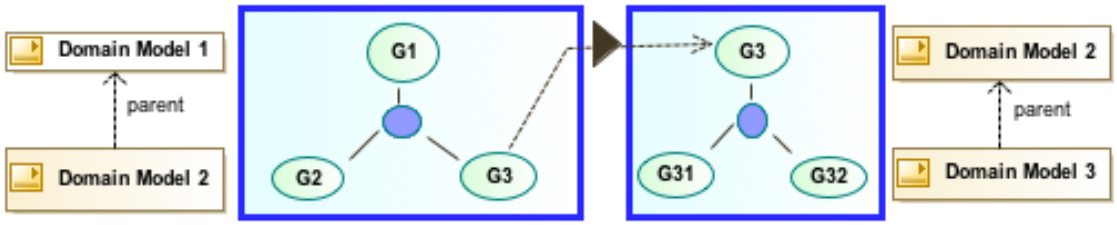


Figure 2.9 – Management of the partitioning of a SysML/KAOS goal model

To be used for large complex systems, SysML/KAOS allows the refinement of a leaf goal of a goal diagram in another diagram having the goal as root. For example, in Figure 2.9, goal *G3*, which is a leaf goal 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 2.9: `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.

2.5.2 Illustration

We have identified two graphical syntaxes to represent ontologies: the syntax proposed by *OntoGraph* [55] and the one proposed by *OWLGred* [136]. The *OntoGraph* syntax is the one used in [98]. Unfortunately, it does not allow the representation of some domain model elements such as attributes or cardinalities. For this 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 2.10, 2.11 and 2.12 represent respectively the domain model associated with the root level of the goal diagram of Fig. 2.1 (*localization_component_0*), that associated with the first refinement level (*localization_component_1*) and that associated with the second one (*localization_component_2*).

Ontology Associated with the Root Level

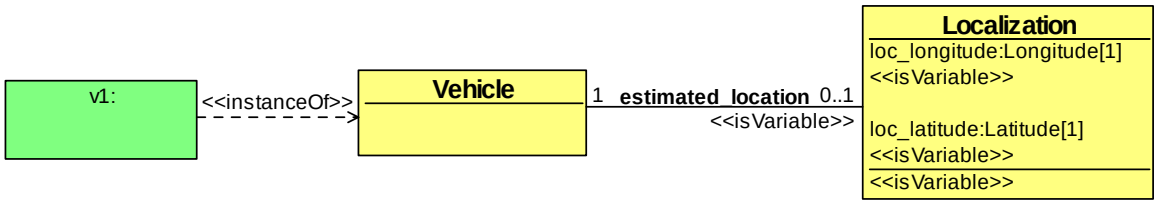


Figure 2.10 – ***localization_component_0***: ontology associated with the root level of the goal diagram of Fig. 2.1

In ontology *localization_component_0* (Fig. 2.10), 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 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 domain cardinality has *minCardinality=maxCardinality*=1, and its associated range cardinality has *minCardinality=0* and *maxCardinality=1*.

Ontology Associated with the First Refinement Level

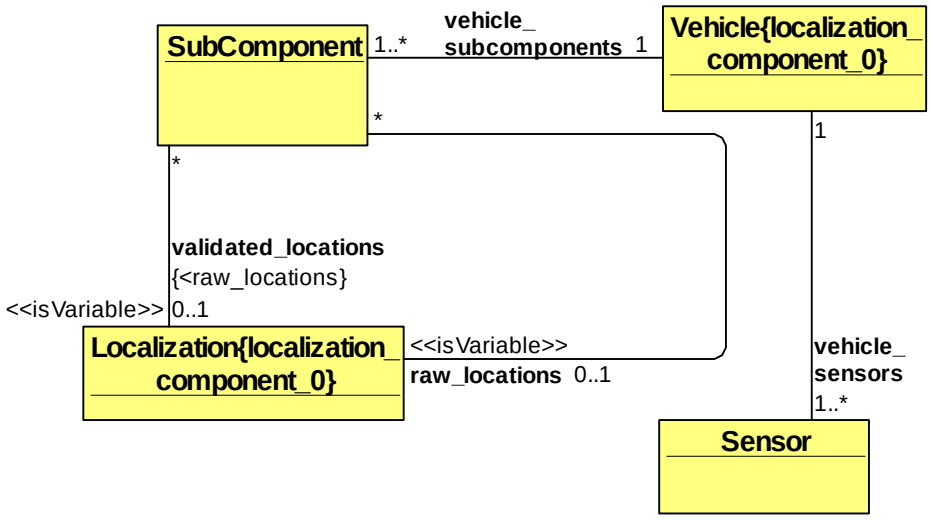


Figure 2.11 – ***localization_component_1***: ontology associated with the first refinement level of the goal diagram of Fig. 2.1

Ontology *localization_component_1* (Fig. 2.11) has ontology *localization_component_0* (Fig. 2.10) 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.

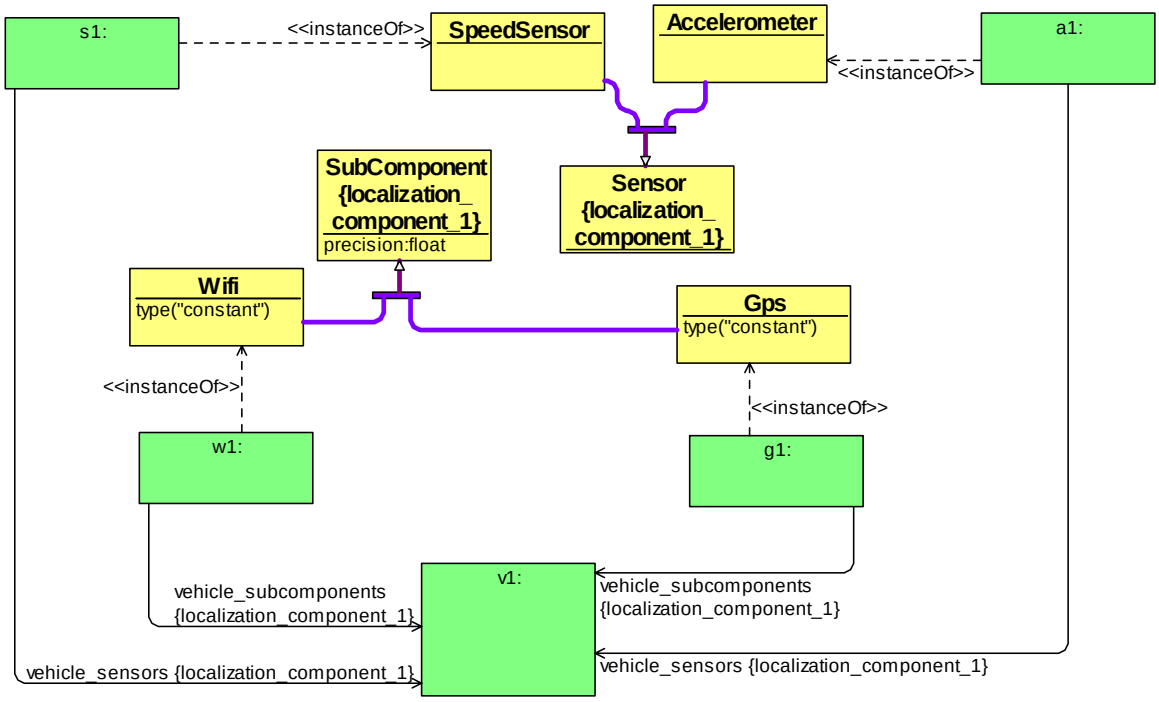


Figure 2.12 – ***localization_component_2***: ontology associated with the second refinement level of the goal diagram of Fig. 2.1

Ontology Associated with the Second Refinement Level

Ontology *localization_component_2* (Fig. 2.12) has ontology *localization_component_1* (Fig. 2.11) 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 2.5: 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 $py > px$.

$$greaterThan(py, px) \leftarrow Wifi(x) \wedge precision(x, px) \wedge Gps(y) \wedge precision(y, py) \quad (2.5)$$

2.6 Conclusion

In this paper, we have first presented the explicitness of the semantics of SysML/KAOS goal models 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 entities and constraints using an ontology modeling language for which a metamodel is defined. The proposal is illustrated through the specification of a localization component for a *Cycab* vehicle.

Work in progress is aimed at developing mechanisms for the explicitness of the semantics of SysML/KAOS domain models in Event-B. We are also working on integrating the language within the open-source platform *Openflexo* [109] which federates the various contributions of *FORMOSE* project partners [17].

Chapitre 3

Du modèle de domaine vers une spécification B System

Résumé

Ce chapitre traite de la traduction des modèles de domaine *SysML/KAOS* en spécifications *B System*. Ses contributions sont de deux ordres. La première réside dans la définition d'une sémantique formelle pour le langage SysML/KAOS de modélisation du domaine. Ce dernier, nous le rappelons, permet la représentation des éléments caractéristiques du domaine d'application d'un système à l'aide d'ontologies. La deuxième contribution réside dans l'élaboration d'une définition, d'abord informelle puis formelle, des règles de traduction de modèles de domaine *SysML/KAOS* en spécifications *B System*. Ces règles permettent d'exploiter la modélisation du domaine afin de produire la partie structurelle de la spécification *B System* issue de la formalisation des modèles d'exigences SysML/KAOS.

Tant le langage que les règles sont formellement définis en utilisant la méthode *Event-B*. Leur cohérence, leur complétude ainsi que diverses propriétés caractéristiques ont été prouvées sous *Rodin*. Il ressort que les règles sont convergentes. Bien plus, elles préservent la structure : les correspondances de deux éléments liés au sein d'un modèle de domaine sont également liées au sein du modèle *B System* correspondant.

Commentaires

La première contribution ici réside dans l'utilisation de la méthode *Event-B* afin de définir formellement, d'un point de vue syntaxique et sémantique, le langage SysML/KAOS de modélisation du domaine. Elle est suivie par l'élaboration d'une définition, d'abord informelle puis formelle, des règles permettant d'exploiter la modélisation du domaine afin de produire la partie structurelle de la spécification *B System* issue de la formalisation des modèles d'exigences SysML/KAOS. Il s'agit également ici de décrire les activités ayant permis de prouver la cohérence, la complétude, la convergence et l'isomorphisme des règles au travers de la plateforme *Rodin*. L'annexe A étend les définitions introduites dans ce chapitre à l'entièreté du langage et des règles.

Les contributions décrites dans ce chapitre ont fait l'objet d'un article accepté et publié [65] dans le cadre de la 6^e édition de la conférence internationale *ABZ (ASM, Alloy, B, TLA, VDM, Z)* qui s'est déroulée à Southampton, Royaume-Uni en juin 2018.

Les contributions et l'article sus-cités ont été élaborés par mes soins en tenant compte des remarques et commentaires issus de mon équipe d'encadrement.

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

Steve Jeffrey Tueno Fotso
Université Paris Est Créteil, LACL, Créteil, France
Université de Sherbrooke, GRIL, Québec, Canada
steve.tuenofotso@univ-paris-est.fr

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

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

Marc Frappier
Université de Sherbrooke, GRIL, Québec, Canada
Marc.Frappier@usherbrooke.ca

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

Abstract

This paper is about an extension of the *SysML/KAOS* requirements engineering method with a language for domain modeling using ontologies. More precisely, it concerns the translation of these domain models into *B System* for system construction. The contributions of this paper are twofold. The first one is a formal semantics for the domain modeling language. The second one is the informal and formal definition of translation rules between SysML/KAOS domain models and *B system* specifications. They are required to provide the structural part of a *B system* specification obtained from the formalisation of SysML/KAOS goal models. The translation rules are formally specified in *Event-B*. Their consistency and completeness are proved using *Rodin*. We show that they are convergent and structure preserving (two related elements within the source model remain related within the target model).

3.1 Introduction

Our study, part of the *FORMOSE* project [17], focuses on an approach for designing systems in critical areas such as railway or aeronautics. The development of such systems, in view of their complexity, requires several verifications and validation steps, more or less formal, with regard to the current regulations. In [102], rules have been defined in order to produce a formal specification from *SysML/KAOS* goal models [70, 92]. Nevertheless, the generated specification did not contain the system state. This is why in [98], we have presented the use of ontologies and *UML* class and object diagrams for domain properties representation; we have also introduced rules to derive the system state from these domain representations. Unfortunately, the proposed approach raised several concerns such as the use of several modeling formalisms for the representation of domain knowledge or the disregard of the variability aspect of domain models. In addition, the proposed rules were incomplete and informal. We have therefore proposed in [61] a language for domain knowledge representation through ontologies that meets the shortcomings of [98]. The language allows a high-level modeling of domain properties. This enables the expression of more precise and complete properties. In this paper, we propose rules for translating SysML/KAOS domain models into *B System* specifications. These rules have all been defined and the most relevant ones have been formally specified with *Event-B* [8] and verified with *Rodin* [35]. The formalisation activity is necessary to assess the quality of the SysML/KAOS domain modeling language and of the translation rules, given the criticality of application domains. The *Event-B* method has been chosen because it involves intuitive mathematical concepts and has a powerful refinement logic. It has also been chosen because it is supported by industrial-strength tools. This work contributes to define a formal semantics for the SysML/KAOS domain modeling language, through the definition of its metamodel and its associated constraints in the form of *Event-B* specifications. In the paper, we provide the formal definition of some translation rules, chosen because they are representative of our work and summarise the benefits and difficulties of their expression and verification with Rodin. SysML/KAOS has been used to deal with the *Hybrid ERTMS/ETCS level 3* case study [79]. It has also been applied on the landing gear system case study [29] and on other case studies (see [133]). The presentation of the work done on the case studies is out of the scope of this paper, but we use an excerpt from the landing gear system case study to illustrate our work.
The remainder of this paper is structured as follows: Section 2 briefly describes the key concepts related to the study. This is followed by a presentation, in Section 3, of the specification in *Event-B*, of the *B System* and SysML/KAOS domain modeling languages. In Section 4, we describe some representative translation rules and we

provide their formal definition. Section 5 underlines the benefits of using the *Event-B* method to express and validate the rules and some challenges encountered. It ends with a positioning of our work with regard to the state of the art. Finally, Section 6 reports our conclusions and discusses future work.

3.2 Context

3.2.1 SysML/KAOS

SysML/KAOS [70, 92] is a requirements engineering method which extends the *SysML* UML profile with a set of concepts from KAOS [138] allowing to represent functional and non-functional requirements. It combines the traceability features provided by *SysML* with goal expressiveness provided by *KAOS*. SysML/KAOS goal models allow the representation of requirements to be satisfied by the system and of expectations with regard to the environment through a goal hierarchy. The hierarchy is built through a succession of refinements using different operators: *AND* and *OR*. An **AND refinement** decomposes a goal into subgoals, and all of them must be achieved to realise the parent goal. 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. The formalisation of SysML/KAOS goal models is detailed in [102]. The proposed rules allow the generation of a formal model whose structure reflects the hierarchy of goal diagrams: one component is associated with each level of the goal hierarchy; this component defines one event for each goal. Proof obligations are defined to formalise the semantics of refinement links between goals.

In this paper, we use the landing gear system case study to illustrate some elements of our approach [29, 133]. Figure 3.1 is an excerpt from its goal diagram focused on the purpose of landing gear expansion (**makeLGExtended**). To achieve it, the handle must be put down (**putHandleDown**) and landing gear sets must be extended (**makeLSExtended**).

3.2.2 Domain Modeling in SysML/KAOS

The SysML/KAOS domain modeling language [61, 64] uses ontologies to represent domain models. It is based on *OWL* [118] and *PLIB* [112], two well-known and complementary ontology modeling languages. Figure 4.3 is an excerpt of its metamodel. The *parent* association represents the hierarchy of domain models. Each domain model corresponds to a refinement level in the SysML/KAOS goal model. A *concept* (instance of metaclass Concept) represents a collection of individuals with

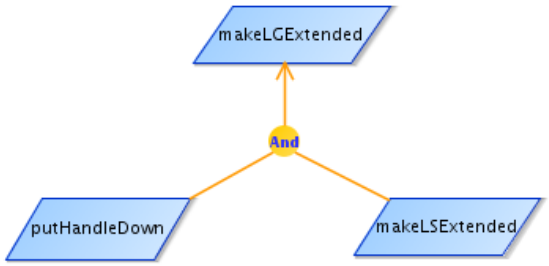


Figure 3.1 – Excerpt from the landing gear system goal diagram

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

    individuals:
       LG1
  }
}
```

Figure 3.2 – Excerpt from the ontology associated with the root level of the goal model

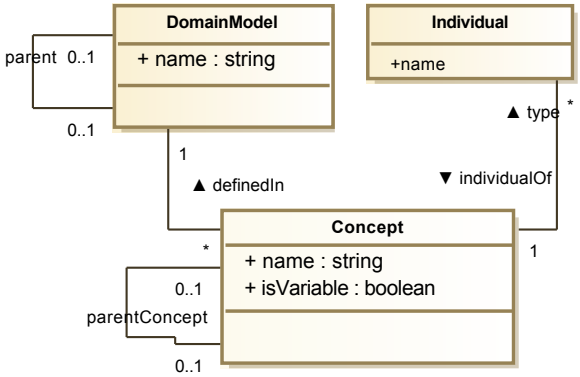


Figure 3.3 – Excerpt from the Metamodel Associated with the domain modeling language

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 *constant* (*isVariable=false*). Figure 3.2 gives an excerpt from the domain model associated to the root level of the landing gear system goal model.
In the rest of this paper, *source* is used in place of SysML/KAOS domain model.

3.2.3 Event-B and B System

Event-B [8] 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 [17], and supported by *Atelier B* [41]. *Event-B* and *B System* have the same semantics defined by proof obligations [8].

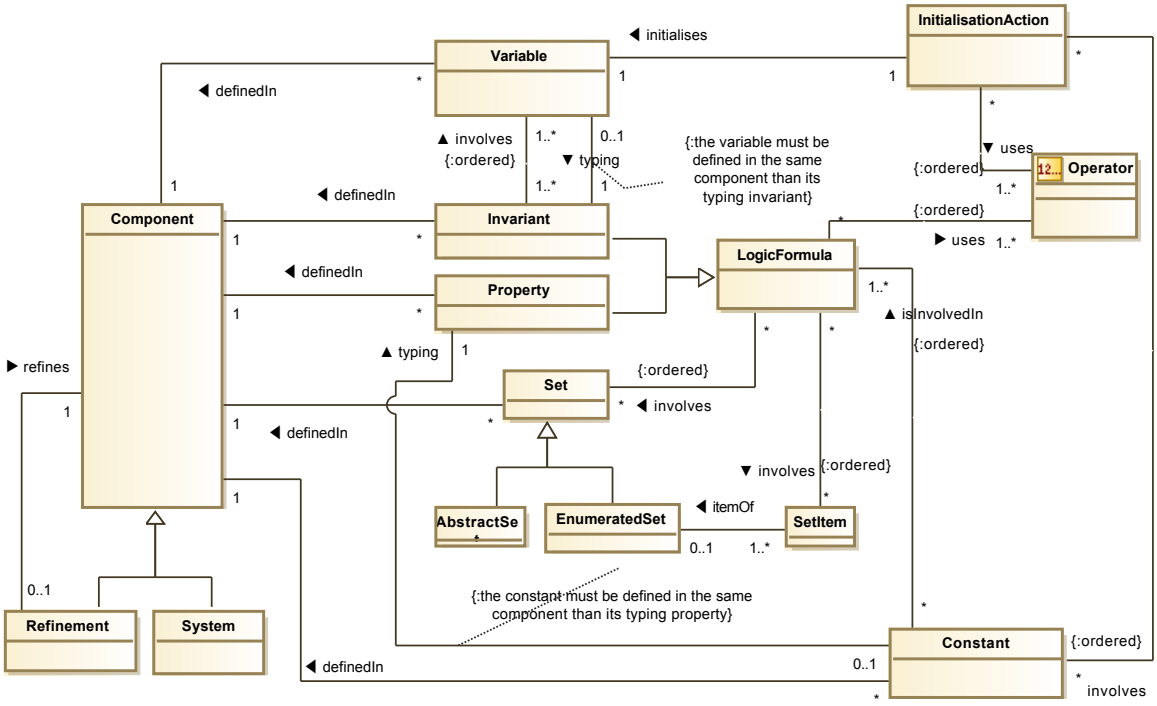


Figure 3.4 – Metamodel of the *B System* specification language

Figure 4.1 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. Operators include, but are not limited to [1], ***Inclusion_OP*** which is used to assert that the first operand is a subset of the second operand ($(Inclusion\_OP, [op_1, op_2]) \Leftrightarrow op_1 \subseteq op_2$) and ***Belonging_OP*** which is used to assert that the first operand is an element of the second operand ($(Belonging\_OP, [op_1, op_2]) \Leftrightarrow op_1 \in op_2$).
In the rest of this paper, *target* is used in place of *B System*.

3.3 Specification of Source and Target Metamodels in Event-B

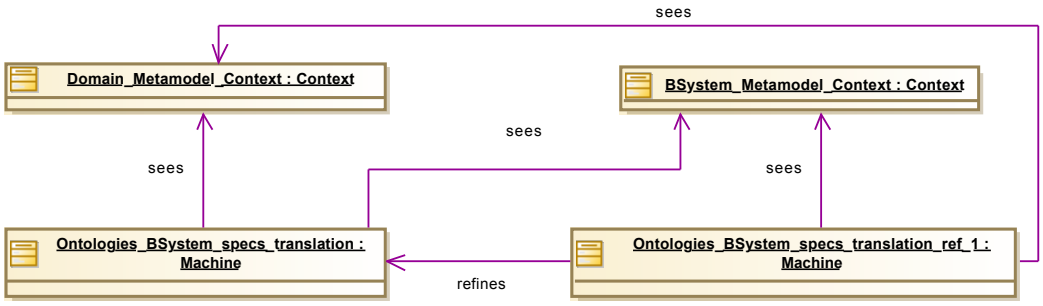


Figure 3.5 – Structure of the *Event-B* specification

As we have chosen *Event-B* to express and verify the translation rules between the source and target metamodels, the first step is to specify them in *Event-B*. This also allows us to formally define the semantics of SysML/KAOS domain models. Figure 3.5 represents the structure of the whole *Event-B* specification. This specification can only be split into two abstraction levels because all the translation rules use the class LogicFormula, except those related to the class DomainModel. The first machine, `Ontologies_BSystem_specs_translation`, contains the rules for the translation of instances of DomainModel into instances of Component. The other rules are defined in the machine `Ontologies_BSystem_specs_translation_ref_1`. We have defined static elements of the target metamodel in a context named `BSystem_Metamodel_Context` and static elements of the source metamodel in the

1. The full list can be found in annex A

one named `Domain_Metamodel_Context`. The two machines have access to the definitions of the contexts. For the sake of concision, we provide only an illustrative excerpt of these *Event-B* specifications. For instance, the model `Ontologies_BSystem_specs_translation_ref_1` contains more than a hundred variables, a hundred invariants and fifty events and it gives rise to a thousand proof obligations. The full version can be found in annex A.

For the translation of some metamodel elements, we have followed the rules proposed in [91, 124], such as: classes which are not subclasses give rise to abstract sets, each class gives rise to a variable typed as a subset and containing its instances and each association or property gives rise to a variable typed as a relation. For example, in the following specification, class DomainModel of the source metamodel and class Component of the target metamodel give rise to abstract sets representing all their possible instances. Variables are introduced and typed (`inv0_1, inv0_2` and `inv0_3`) to represent sets of defined instances.

CONTEXT Domain_Metamodel_Context
SETS DomainModel_Set
END

CONTEXT BSystem_Metamodel_Context
SETS Component_Set
END

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

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

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

Variables are also used to represent attributes and associations [91, 124] such as the attribute isVariable of the class Concept in the source metamodel (`inv1_5`) and the association definedIn between the classes Constant and Component in the target metamodel (`inv1_7`). To avoid ambiguity, we have prefixed and suffixed each element name with that of the class to which it is attached (**e.g.** `Concept_isVariable` or `Constant_definedIn_Component`). Furthermore, for better readability of the specification, we have chosen to add *"s"* to the name of all *Event-B* relations for which an image is a set (**e.g.** `Constant_isInvolvedIn_LogicFormulas` or `Invariant_involves_Variables`).

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

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

3.4 Translation Rules

3.4.1 Overview of Translation Rules

Table 3.1 summarises the translation rules. They are fully described in annex A. These rules cover the formalisation of all elements of the source metamodel, from domain models with or without parents to concepts with or without parents, including relations, individuals or attributes. It should be noted that o_x designates the result of the translation of x and that *abstract* is used for "without parent".

We are not interested in validating the transformation rules of predicates because both source and target metamodels express them using first-order logic notations. The translation of a predicate is a syntactic rewrite. The rules are outlined in [63].

Table 3.1 – Summary of the translation rules

		Domain Model		B System	
		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 $DomainModel\_parent\_DomainModel(DM) = PDM$ PDM has already been translated	o_DM	$o\_DM \in$ Refinement $Refinement\_refines\_Component(o\_DM) = o\_PDM$
3	**Abstract concept**	CO	$CO \in$ Concept $CO \notin dom(Concept\_parent\_Concept\_Concept)$	o_CO	$o\_CO \in$ AbstractSet
4	**Concept with parent**	CO PCO	$\{CO, PCO\} \subseteq$ Concept $Concept\_parentConcept\_Concept(CO) = PCO$ PCO has already been translated	o_CO	$o\_CO \in$ Constant $o\_CO \subseteq o\_PCO$
5	**Relation**	RE CO1 CO2	$\{CO1, CO2\} \subseteq$ Concept $RE \in$ Relation $Relation\_domain\_Concept(RE) = CO1$ $Relation\_range\_Concept(RE) = CO2$ $CO1$ and $CO2$ have already been translated	o_RE	**IF** $(RE \mapsto FALSE) \in Relation\_isVariable$ **THEN** $o\_RE \in$ Constant **ELSE** $o\_RE \in$ Variable **END** $o\_RE \in o\_CO1 \leftrightarrow o\_CO2$ [2]
6	**Attribute**	AT CO DS	$CO \in$ Concept $DS \in$ DataSet $AT \in$ Attribute $Attribute\_domain\_Concept(AT) = CO$ $Attribute\_range\_Concept(AT) = DS$ CO and DS have already been translated	o_AT	**IF** $(AT \mapsto FALSE) \in Attribute\_isVariable$ **THEN** $o\_AT \in$ Constant **ELSE** $o\_AT \in$ Variable **END** $o\_AT \in o\_CO \leftrightarrow o\_DS$ [3]
7	**Concept variability**	CO	$CO \in$ Concept $Concept\_isVariable(CO) = TRUE$ CO has already been translated	X_CO	$X\_CO \in$ Variable $X\_CO \subseteq o\_CO$
8	**Individual**	Ind CO	$Ind \in$ Individual $CO \in$ Concept $Individual\_individualOf\_Concept(Ind) = CO$ (CO has already been translated)	o_Ind	$o\_Ind \in$ Constant $o\_Ind \in o\_CO$

The translation of the ontology of Fig. 3.2 produces the specification below:

SYSTEM *landing_gear_system_ref_0*
SETS *LandingGear*
CONSTANTS *LG1*
PROPERTIES
$LG1 \in LandingGear \wedge LandingGear = \{LG1\}$
END

The root domain model is translated into a system component named `landing_gear_system_ref_0` (line 1 of Table 3.1). The abstract set `LandingGear` appears because *LandingGear* is an instance of the class Concept (line 3). The individual `LG1` gives rise to a constant $LG1 \in LandingGear$ (line 8). The property $LandingGear = \{LG1\}$ translates the fact that the *isVariable* property of *LandingGear* is set to *false*.

2. As usual, this relation becomes a *function*, an *injection*, ... according to the cardinalities of *RE*.
3. Depending on attribute properties, this relation may become a partial or total function.

3.4.2 Event-B Specification of Translation Rules

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

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

Translation rules have been modeled as *convergent* events, this guarantees that each rule will be triggered a finite number of times [8] (see Section 3.5.1). Each event execution translates an element of the source into the target. Variants and event guards and type have been defined such that when the system reaches a state where no transition is possible (deadlock state), all translations are done (see Section 3.5.1). Up to fifty events have been specified. The rest of this section provides an overview of the specification of some of these events in order to illustrate the formalisation process and some of its benefits and difficulties. The full specification can be found in annex A.

Translating a Domain Model with Parent (line 2 of table 3.1)

The corresponding event is called *domain_model_with_parent_to_component*. It states that a domain model, associated with another one representing its parent, gives rise to a refinement component.

MACHINE Ontologies_BSystem_specs_translation
INVARIANT
inv0_6: $Refinement\_refines\_Component \in Refinement \rightarrowtail Component$
inv0_7: $\forall xx, px \cdot ((xx \in dom(DomainModel\_parent\_DomainModel)$
$\wedge\ px = DomainModel\_parent\_DomainModel(xx)$
$\wedge\ px \in dom(DomainModel\_corresp\_Component)$
$\wedge\ xx \notin dom(DomainModel\_corresp\_Component))$
$\Rightarrow DomainModel\_corresp\_Component(px) \notin ran(Refinement\_refines\_Component))$
Event domain_model_with_parent_to_component ⟨convergent⟩ $\widehat{=}$
any DM PDM o_DM
where
grd0: $dom(DomainModel\_parent\_DomainModel) \setminus dom(DomainModel\_corresp\_Component) \neq \emptyset$
grd1: $DM \in dom(DomainModel\_parent\_DomainModel) \setminus dom(DomainModel\_corresp\_Component)$
grd2: $dom(DomainModel\_corresp\_Component) \neq \emptyset$
grd3: $PDM \in dom(DomainModel\_corresp\_Component)$
grd4: $DomainModel\_parent\_DomainModel(DM) = PDM$

grd5: $Component\_Set \setminus Component \neq \emptyset$
grd6: $o\_DM \in Component\_Set \setminus Component$
then
act1: $Refinement := Refinement \cup \{o\_DM\}$
act2: $Component := Component \cup \{o\_DM\}$
act3: $Refinement\_refines\_Component(o\_DM) := DomainModel\_corresp\_Component(PDM)$
act4: $DomainModel\_corresp\_Component(DM) := o\_DM$
END
END

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

Translating a Concept with Parent (line 4 of table 3.1)

This rule leads to two events: the first one for when the parent concept corresponds to an abstract set (the parent concept does not have a parent: line 3 of table 3.1) and the second one for when the parent concept corresponds to a constant (the parent concept has a parent: line 4 of table 3.1). Below is the specification of the first event [4].

Event concept_with_parent_to_constant_1 ⟨convergent⟩ ≙
any CO o_CO PCO o_lg o_PCO
where
grd1: $CO \in dom(Concept\_parentConcept\_Concept) \setminus dom(Concept\_corresp\_Constant)$
grd2: $PCO \in dom(Concept\_corresp\_AbstractSet)$
grd3: $Concept\_parentConcept\_Concept(CO) = PCO$
grd4: $Concept\_definedIn\_DomainModel(CO) \in dom(DomainModel\_corresp\_Component)$
grd5: $o\_CO \in Constant\_Set \setminus Constant$
grd6: $o\_lg \in LogicFormula\_Set \setminus LogicFormula$
grd7: $o\_PCO = Concept\_corresp\_AbstractSet(PCO)$
then
act1: $Constant := Constant \cup \{o\_CO\}$
act2: $Concept\_corresp\_Constant(CO) := o\_CO$
act3: $Constant\_definedIn\_Component(o\_CO) := DomainModel\_corresp\_Component($
$Concept\_definedIn\_DomainModel(CO))$
act4: $Property := Property \cup \{o\_lg\}$
act5: $LogicFormula := LogicFormula \cup \{o\_lg\}$
act6: $LogicFormula\_uses\_Operators(o\_lg) := \{1 \mapsto Inclusion\_OP\}$
act7: $Constant\_isInvolvedIn\_LogicFormulas(o\_CO) := \{1 \mapsto o\_lg\}$

4. Some guards and actions have been removed for the sake of concision

act8: $LogicFormula\_involves\_Sets(o\_lg) := \{2 \mapsto o\_PCO\}$
act9: $Constant\_typing\_Property(o\_CO) := o\_lg$
END

The rule asserts that any concept, associated with another one, with the parentConcept association, gives rise to a constant, typed as a subset of the *B System* element corresponding to the parent concept. We use an instance of LogicFormula, named *o_lg*, to capture this constraint linking the concept and its parent correspondents (*o_CO* and *o_PCO*). Guard `grd2` constrains the parent correspondent to be an instance of AbstractSet. Guard `grd4` ensures that the event will not be triggered until the translation of the domain model containing the definition of the concept. Action `act3` ensures that *o_CO* is defined in the component corresponding to the domain model where *CO* is defined. Action `act6` defines the operator used by *o_lg*. Because the parent concept corresponds to an abstract set, *o_CO* is the only constant involved in *o_lg* (`act7`); *o_PCO*, the second operand, is a set (`act8`). Finally, action `act9` defines *o_lg* as the typing predicate of *o_CO*.

Example:

SysML/KAOS domain model	B System specification
concept pco	**SETS** *pco* **CONSTANTS** *co*
concept co **parent concept** pco	**PROPERTIES** co ⊆ pco

The specification of the second event (when the parent concept corresponds to a constant) is different from the specification of the first one in some points. The three least trivial differences appear at guard `grd2` and at actions `act7` and `act8`. Guard `grd2` constrains the parent correspondent to be an instance of Constant: $PCO \in dom(Concept\_corresp\_Constant)$. Thus, the first and the second operands involved in *o_lg* are constants:

act7: $Constant\_isInvolvedIn\_LogicFormulas := Constant\_isInvolvedIn\_LogicFormulas \Leftarrow \{$
$(o\_CO \mapsto \{1 \mapsto o\_lg\}), o\_PCO \mapsto Constant\_isInvolvedIn\_LogicFormulas(o\_PCO) \cup \{2 \mapsto o\_lg\}\}$
act8: $LogicFormula\_involves\_Sets(o\_lg) := \emptyset$

This approach to modeling logic formulas allows us to capture all the information conveyed by the predicate which can then be used to make inferences and semantic analysis. It is especially useful when we deal with rules to propagate changes made to a generated *B System* specification back to the domain model (ie, propagate changes made to the target into the source). The study of these propagation rules will be the next step in our work.

3.5 Discussion and Experience

The rules that we propose allow the automatic translation of domain properties, modeled as ontologies, to *B System* specifications, in order to fill the gap between the system textual description and the formal specification. It is thus possible to benefit from all the advantages of a high-level modeling approach within the framework of the formal specification of systems: decoupling between formal specification handling difficulties and system modeling; better reusability and readability of models; strong traceability between the system structure and stakeholder needs. Applying the approach on case studies [133] allowed us to quickly build the refinement hierarchy of the system and to determine and express the safety invariants, without having to manipulate the formal specifications. Furthermore, it allows us to limit our formal specification to the perimeter defined by the expressed needs. This step also allowed us to enrich the domain modeling language expressiveness.

3.5.1 Benefits

Formally defining the SysML/KAOS domain modeling language, using *Event-B*, allowed us to completely fulfill the criteria for it to be an ontology modeling formalism [15]. Furthermore, formally defining the rules in *Event-B* and discharging the associated proof obligations allowed us to prove their consistency, to animate them using *ProB* and to reveal several constraints (guards and invariants) that were missing when designing the rules informally or when specifying the metamodels. For instance: (1) if an instance of Concept x, with parent px does not have a correspondent yet and if px does, then, the correspondent of px should not be refined by any instance of Component (`inv0_7` defined in `Ontologies_BSystem_specs_translation` and described in Sect. 3.4.2); (2) elements of an enumerated data set should have correspondents if and only if the enumerated data set does; (3) if a concept, given as the domain of an attribute (instance of Attribute), is variable, then the attribute must also be variable; the same constraint is needed for the domain and the range of a relation. In case of absence of this last constraint, it is possible to reach a state where an attribute maplet (instance of AttributeMaplet) is defined for a non-existing individual (because the individual has been dynamically removed). These constraints have been integrated in the SysML/KAOS domain modeling language in order to strengthen its semantics.

There are two essential properties that the specification of the rules must ensure and that we have proved using Rodin. The first one is that the rules are isomorphisms and it guarantees that established links between elements of the ontologies are preserved between the corresponding elements in the *B System* specification and vice versa. To do this, we have introduced, for each considered link between elements,

an invariant guaranteeing the preservation of the corresponding link between the correspondences and we have discharged the associated proof obligations. This leads to fifty invariants. For example, to ensure that for each domain model *pxx*, parent of *xx*, the correspondent of *xx* refines the correspondent of *pxx* and vice versa, we have defined the following invariants:

inv0_8: *For each domain model* **pxx**, *parent of a domain model* **xx**, *when* **xx** *and* **pxx** *will be translated, then the correspondence of* **xx** *will refine the correspondence of* **pxx** :

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

Its dual version is defined by

inv0_9: *For each component* **o_xx**, *which refines a component* **o_pxx**, *if* **o_xx** *and* **o_pxx** *are introduced by translation rules, then the domain model corresponding to* **o_pxx** *is the parent of the domain model corresponding to* **o_xx**:

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

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

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

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

The second essential property is to demonstrate that the system will always reach a state where all translations have been established (*P0*).

To manually demonstrate *P0*, we have proven that all events can be disabled if and only if all translations have been done. For example, let's consider rules dealing about the translation of instances of DomainModel (lines 1 and 2 of table 3.1); the negation of guards results in

$$
\begin{array}{l}
DomainModel \setminus (dom(DomainModel\_corresp\_Component) \cup dom(DomainModel\_parent\_DomainModel)) = \emptyset \\
\quad \wedge\ dom(DomainModel\_parent\_DomainModel) \setminus dom(DomainModel\_corresp\_Component) = \emptyset \\
\qquad \Leftrightarrow \\
DomainModel \subseteq (dom(DomainModel\_corresp\_Component) \cup dom(DomainModel\_parent\_DomainModel)) \\
\quad \wedge\ dom(DomainModel\_parent\_DomainModel) \subseteq dom(DomainModel\_corresp\_Component) \\
\qquad \Leftrightarrow \\
DomainModel \subseteq dom(DomainModel\_corresp\_Component) \\
\qquad \Leftrightarrow \\
DomainModel = dom(DomainModel\_corresp\_Component) \\
\text{because } dom(DomainModel\_corresp\_Component) \subseteq DomainModel
\end{array}
$$

To automatically prove it, we have introduced, within each machine, a *variant* defined as the difference between the set of elements to be translated and the set of elements already translated. Then, each event representing a translation rule has been marked as *convergent* and we have discharged the proof obligations ensuring that each of them decreases the *variant*. For each rule, the number of elements to be translated is defined and finite; since we are sure, regarding the event convergence, that each triggering of the rule translates an untranslated element, we are guaranteed that in a finite number of triggerings, all elements will be translated. For example, in machine `Ontologies_BSystem_specs_translation` containing the definition of translation rules from domain models to *B System* components, the variant was defined as

$DomainModel \setminus dom(DomainModel\_corresp\_Component)$

Thus, at the end of system execution, we will have

$dom(DomainModel\_corresp\_Component) = DomainModel$

which will reflect the fact that each domain model has been translated into a component.

3.5.2 Challenges

There is no predefined type for ordered sets in *Event-B*. This problem led us to the definition of composition of functions in order to define relations on ordered sets. Moreover, because of the size of our model (about one hundred invariants and about fifty events for each machine), we noted a rather significant performance reduction of *Rodin* during some operations such as the execution of auto-tactics or proof replay on undischarged proof obligations that have to be done after each update in order to discharge all previously discharged proofs. Table 3.3 summarises the key characteristics of the Rodin project corresponding to the *Event-B* specification of

metamodels and rules. The proof obligations have been discharged using the Rodin tool extended with *Atelier B provers* [115] and *SMT solvers* [127]. The automatic provers seemed least comfortable with functions ($\nrightarrow, \rightarrowtail, \rightarrow, \twoheadrightarrow$) and become almost useless when those operators are combined in definitions as for ordered associations ($r \in (A \rightarrow (\mathbb{N}_1 \nrightarrow B))$).

Table 3.3 – Key characteristics of the *Event-B* specification Rodin project

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

3.5.3 Related Work

The study of correspondence links between domain models or ontologies and formal methods has been the subject of numerous works.

In [24], domain models consist of entities and operations which can be atomic or composite. Atomic entities correspond to states of the formal model. Composite entities correspond to sets, groups, lists or associations of states. Furthermore, operations are translated into state-changing actions, composite operations corresponding to composition of actions.

The work presented in [24] is interested in describing entities, their mereology, their behaviours and their transformations. Rules are provided for the formalisation of these elements. On the other hand, our study is focused on the description of entities of a system application domain and their instances, of their constraints and of their attributes and associations. Moreover, our modeling is done through successive refinements and the translation rules integrate the refinement links between modules. In [141], an approach is proposed for the automatic extraction of domain knowledge, as *OWL* ontologies, from *Z/Object-Z (OZ)* models: OZ types and classes are transformed into OWL classes. Relations and functions are transformed into OWL properties, with the *cardinality* restricted to *1* for total functions and the *maxCardinality* restricted to *1* for partial functions. OZ constants are translated into OWL individuals. Rules are also proposed for subsets and state schemas. A similar approach is proposed in [53], for the extraction of *DAML* ontologies from *Z* models. These approaches are interested in correspondence links between formal

methods and ontologies, but their rules are restricted to the extraction of domain model elements from formal specifications. Furthermore, all elements extracted from a formal model are defined within a single ontology component, while in our approach, each ontology refinement level corresponds to a formal model component.

Some rules for passing from an *OWL* ontology representing a domain model to *Event-B* specifications are proposed in [12], in [13] and through a case study in [98]. In [12], domain properties are described through data-oriented requirements for concepts, attributes and associations and through constraint-oriented requirements for axioms. Possible states of a *variable* element are represented using UML state machines. Concepts, attributes and associations arising from data-oriented requirements are modeled as UML class diagrams and translated to Event-B using *UML-B* [124]: nouns and attributes are represented as UML classes and relationships between nouns are represented as UML associations. *UML-B* is also used for the translation of state machines to Event-B variables, invariants and events. The approaches in [12] and [13] require a manual transformation of the ontology before the possible application of translation rules to obtain the formal specifications: In [12], it is necessary to convert OWL ontologies into UML diagrams; in [13], the proposal requires the generation of an *ACE (Attempto Controlled English)* version of the *OWL* ontology which serves as basis for development of the *Event-B* specification. Furthermore, for this to be completed, the names of ontology elements must necessarily be expressed in English. Moreover, since the OWL formalism supports weak typing and multiple inheritance, the approaches define a unique *Event-B* abstract set named *Thing*. Thus, all sets, corresponding to OWL classes, are defined as subsets of *Thing*. Our formalism, on the other hand, imposes strong typing and simple inheritance; which makes it possible to translate some concepts into *Event-B* abstract sets. In [98], the case study reveals that each ontology class, having no individual, is modeled as an Event-B abstract set. If the class has individuals, then it is modeled as an enumerated set. Finally, each object property between two classes is modeled as a constant defines as a relation.

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

Many studies have been done on the translation of UML diagrams into B specifications such as [91, 124]. They inspired many of our rules, like those dealing with the translation of concepts (classes) and of attributes and relations (associations). But, our work differs from them because of the distinctions between ontologies and UML diagrams: within an ontology, concepts or classes and their instances are represented within the same model as well as the predicates defining domain

constraints. Moreover, these studies are most often interested in the translation of model elements and not really in handling links between models. Finally, in the case of the SysML/KAOS domain modeling language, the variability properties (attributes characterising the belonging of an element to the static or dynamic knowledge) are first-class citizens, as well as association characteristics. As a result, they are explicitly represented.

In [27], an approach to model the theoretical foundations of Event-B using Event-B is sketched in order to validate some Event-B extensions related to distribution, to composition and to decomposition. However, the proposal considers neither Event-B contexts (Sets, Constants, Properties) nor refinement links and the definition of predicates makes their representation too abstract.

3.6 Conclusion and Future Work

This paper proposes an *Event-B* formalisation of translation rules between SysML/KAOS domain models and *B System* specifications. Their consistency was proven through Rodin [35]. This allowed us to ensure some properties regarding rules such as convergence and isomorphisms and to determine some relevant guards and invariants missing in informal definitions. The rules are implemented within an open source tool [133] which support construction of domain models and generation of the corresponding *B System* specifications. The tool is built on top of *Jetbrains Meta Programming System* [87], a platform to design domain specific languages using language-oriented programming. It was assessed on three major case studies [133].

This work allows the complete extraction of the structural part of the *B System* specification obtained from SysML/KAOS goal models and the initialisation of state variables. However, it remains necessary to manually provide the body of events, which can lead to updates of the specification obtained from domain models.

Work in progress is aimed at evaluating the impact of updates performed within a *B System* specification on the corresponding SysML/KAOS models. We are also interested in integration of the translation rules within the open-source platform *Openflexo* [109] which federates the various contributions of *FORMOSE* project partners [17] and which currently supports the construction of SysML/KAOS goal and domain models.

Chapitre 4

Prise en compte de l'évolution d'un modèle B System : cas de la propagation des ajouts d'éléments structurels

Résumé

De nos jours, l'utilité des méthodes formelles pour la vérification et la validation formelles des spécifications de systèmes est bien établie, du moins en ce qui concerne les systèmes critiques. Toutefois, l'un des principaux obstacles à leur vulgarisation réside dans l'obtention de la spécification formelle du système, et, dans le cas d'une méthode formelle basée sur le raffinement à l'exemple de *B System* et d'*Event-B*, dans l'obtention de la spécification la plus abstraite. La méthode formelle d'ingénierie des exigences *SysML/KAOS* a été élaborée afin de surmonter cette difficulté. Elle comprend un langage de modélisation des buts permettant de représenter les exigences du système. Des règles de traduction ont de plus été définies afin d'obtenir une spécification *B System* à partir des modèles de buts SysML/KAOS : cette spécification constitue l'ossature de la modélisation formelle des exigences du système. Pour la compléter, un langage de modélisation du domaine d'application du système a été défini. La

formalisation des modèles de domaine ainsi construits permet d'obtenir la partie structurelle de la spécification *B System* issue des modèles de buts. Il s'agit par la suite de spécifier le corps des évènements, puis de vérifier et valider le modèle *B System* obtenu.

Cependant, il apparaît très souvent que de nouveaux éléments doivent être ajoutés à la spécification *B System* issue des modèles SysML/KAOS. Cette nécessité peut survenir tant de la spécification du corps des événements que des tâches de vérification et de validation formelles. Ce chapitre définit en conséquence un ensemble de règles permettant de propager tout ajout d'élément au sein de la partie structurelle d'une spécification *B System*, vers les modèles de domaine impactés. Le chapitre décrit également comment les règles ont été spécifiées formellement en utilisant *Event-B*. Ceci a permis de prouver leur cohérence et leur convergence, au travers de la plateforme *Rodin*. Il a également été possible de démontrer qu'elles préservent la structure des modèles : les correspondances de deux éléments liés au sein d'un modèle *B System* sont également liées au sein du modèle de domaine correspondant.

Commentaires

La contribution ici réside dans l'élaboration d'une définition, d'abord informelle puis formelle, des règles permettant de propager des ajouts d'éléments au sein de la partie structurelle d'une spécification *B System*, vers les modèles de domaine correspondants. Il s'agit également ici de décrire les activités ayant permis de prouver la cohérence, la convergence et l'isomorphisme des règles au travers de la plateforme *Rodin*. L'annexe A étend les définitions introduites dans ce chapitre à l'entièreté des règles.

Les contributions décrites dans ce chapitre ont fait l'objet d'un article accepté et publié [57] dans le cadre de la 23^{e} édition de la conférence internationale *ICECCS (International Conference on Engineering of Complex Computer Systems)* qui s'est déroulée à Melbourne, Australie en Décembre 2018.

Les contributions et l'article sus-cités ont été élaborés par mes soins en tenant compte des remarques et commentaires issus de mon équipe d'encadrement.

Back Propagating *B System* Updates on SysML/KAOS Domain Models

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

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

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

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

Abstract

Nowadays, the usefulness of the formal verification and validation of system specifications is well established, at least for critical systems. However, one of the main obstacles to their adoption lies in obtaining the formal specification of the system, and, in the case of refinement-based formal methods such as *B System* or *Event-B*, in obtaining the most abstract specification that heads the development of the system. The *SysML/KAOS* requirements engineering method is proposed to overcome this difficulty. It includes a goal modeling language to model requirements from stakeholders needs. Translation rules from a goal model to a *B System* specification have already been defined. They allow to obtain a skeleton of the system specification. To complete it, a language has been defined to express the domain model associated to the goal model. Its translation gives the structural part of the *B System* specification.

However, it very often appears that new elements must be added in the *B System* specification obtained from SysML/KAOS models, discovered for instance when specifying the body of events and/or by using formal validation and/or verification tools. We have therefore defined a set of rules allowing the back propagation, within domain models, of every newly added element. This paper describes these rules and how they are specified

in *Event-B*. Their consistency is proved using the *Rodin* tool. We show that they are structure preserving: two related elements within the *B System* specification remain related within the domain model. This is done by proving various isomorphisms between the *B System* specification and the domain models.

4.1 Introduction

Our work helps to bring three modeling activities closer together: requirements modeling, domain modeling and formal specification. It focuses on the formal requirements modeling of systems in critical areas such as railway or aeronautics. It is developed in the French research project, called *FORMOSE* [17]. In [102], Matoussi *et al.* have defined translation rules to produce formal specifications from *SysML/KAOS* goal models [92]. They generate a skeleton of the system specification. Nevertheless, it remains to define the structural part of the specification (such as user-defined types, variables or constants and their properties) and the body of events. Tueno *et al.* have therefore proposed in [61] a language for domain knowledge representation which combines the expressivity of the ***Ontology Web Language (OWL)***, the constraints provided by the standard ***Part Library (PLIB)*** and the **extensions** needed to guarantee some relevant properties [61]. Translation rules to obtain a *B System* specification from domain models have been defined and formally verified in [65]. They allow the completion of the formalisation of the SysML/KAOS goal model with the structural part of the formal specification and the initialisation of state variables. The specification of the body of events, as illustrated in [58], completes the formal model. The *B System* specification, thus obtained from the needs formulated, can then be verified and validated using the whole range of tools that support the *B* method [10], largely and positively assessed on industrial projects for more than 25 years [93].

The work done on case studies [58, 133] reveals that, very often, new elements need to be added to the structural part of the formal specification. These additions may be required during the specification of the body of events or during the verification and validation of the formal model (e.g. to define an invariant or a theorem required to discharge a proof obligation). Moreover, modeling is often done through several backwards and forwards between the *B System* specification and SysML/KAOS models. These lead us to the definition of a set of rules allowing the back propagation, within the domain model, of elements introduced in the structural part of the *B System* specification. They prevent these additions from introducing inconsistencies between a domain model and its *B System* specification. The most relevant rules have been formally specified with *Event-B* [8] and verified

with *Rodin* [35]. We have proved that the back propagation rules are consistent and structure preserving, by discharging the proof obligations and by proving various isomorphisms between *B System* and domain models. The full *Event-B* specification can be found in annex A. The contribution of this paper is the description of these back propagation rules, based on the definition of metamodels of the SysML/KAOS domain modeling language and of the *B System* specification language. We also provide the formal definition of some rules, chosen because they are representative of our work, and we summarise the benefits and difficulties of their expression and verification with Rodin. Throughout this paper, we use an excerpt of the *steam-boiler control specification problem*, proposed by *J. C. Bauer* in [22] for the *Dagstuhl* workshop, to illustrate our proposal.

The remainder of this paper is structured as follows: Section 2 briefly describes the *Event-B* and *B System* formal methods, the SysML/KAOS requirements engineering method and its goal and domain modeling languages and the rules for translating the models into *B System* specifications. Section 3 describes and illustrates some representative back propagation rules with their formal definition. Section 4 discusses the *Event-B* verification of back propagation rules and the use of the SysML/KAOS method with regards to some related work. Finally, Section 5 reports our conclusion and discusses future work.

4.2 Context

4.2.1 Event-B and B System

Event-B [8] is an industrial-strength formal method for *system modeling*. It is used to incrementally construct a system specification, using refinement, and to prove properties. An *Event-B* model includes a static part called context and a dynamic part called machine. A machine can refine another machine, a context can extend other contexts and a machine can see contexts. *B System* is an *Event-B* syntactic variant proposed by *ClearSy*, an industrial partner in the *FORMOSE* project [17], and supported by *Atelier B* [41].

Figure 4.1 is an excerpt from the metamodel of the *B System* specification language. 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. Constants, abstract and enumerated sets, and their properties, constitute the static part. The dynamic part includes variables constrained through invariants. Properties and invariants are logic formulas (instances of LogicFormula). In our case, it is sufficient to consider that logic formulas are successions of operands in relation through operators. Thus, a logic formula references its operators (instances

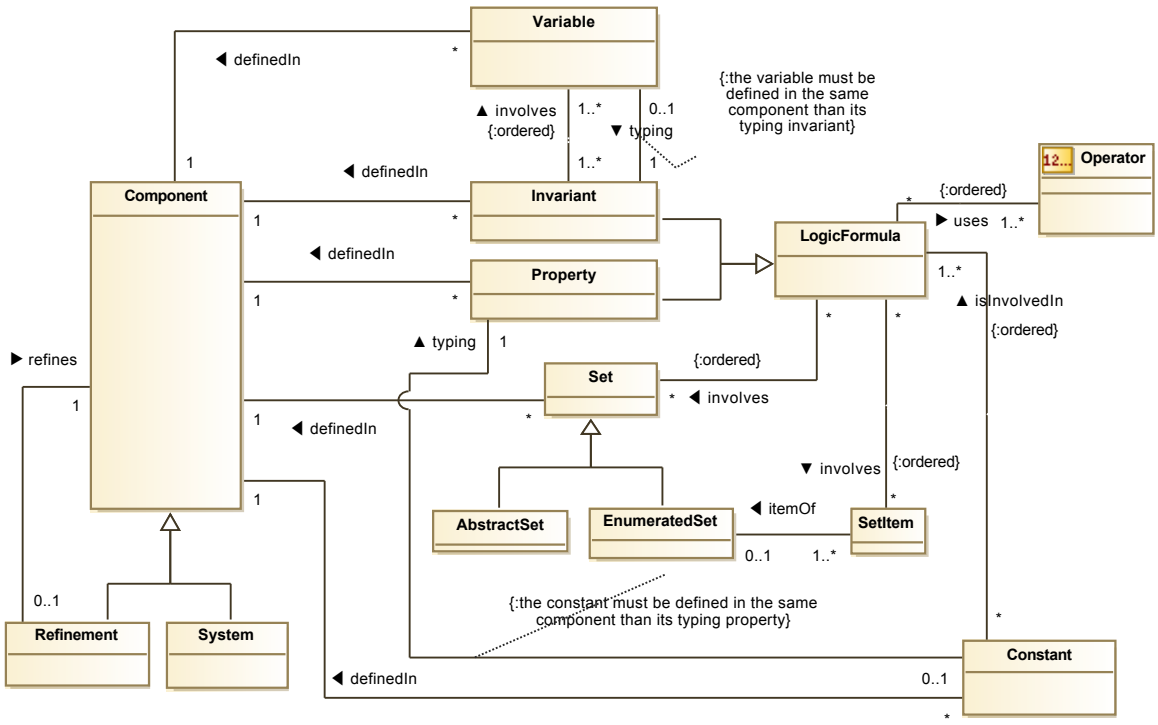


Figure 4.1 – Excerpt from the metamodel of the *B System* specification language

of Operator) and its operands that may be instances of Variable, Constant, Set or SetItem. For instance, operator ***Inclusion_OP*** is used to assert that the first operand is a subset of the second operand $((Inclusion\_OP, [op_1, op_2]) \Leftrightarrow op_1 \subseteq op_2)$. Therefore, we are able to represent logic formulas involving one or more operators such as $([Inclusion\_OP, Cprod\_OP], [op_1, op_2, op_3]) \Leftrightarrow op_1 \subseteq op_2 \times op_3$ which involves operators *Inclusion_OP* (inclusion: $\subseteq$) and *Cprod_OP* (cartesian product: $\times$) and operands op_1, op_2 and op_3. The full list of operators, that we consider, can be found in annex A.

4.2.2 SysML/KAOS

Presentation

SysML/KAOS [92,98] is a requirements engineering method based on *SysML* [78] and KAOS [138]. It combines the traceability features provided by *SysML* with goal expressiveness provided by *KAOS*. *SysML* allows for the capturing of requirements and the maintaining of traceability links between those requirements and design deliverables. *KAOS* defines a requirements modeling language for the representation of requirements to be satisfied by the 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 different refinement operators. For instance, an ***AND refinement*** decomposes a goal into subgoals, and all of them must be achieved to realise the parent goal.

Illustration

The challenge of the steam-boiler control specification problem [22] is to specify a system controlling the level of water in a steam-boiler. The system deals with a steam-boiler (*SB*), a water unit to measure the quantity of water in *SB*, a pump with its controller to provide *SB* with water and a steam unit to measure the quantity of steam flowing out of *SB*. The system can operate in several modes. For instance, in the *normal* mode, the system tries to keep the amount of water in a specific range, with all the units behaving correctly. When a failure occurs on the water unit, the mode is set to *rescue*. In the *rescue* mode, the system tries to keep the amount of water in a range different from the normal range, despite of a possible failure of the water unit. It estimates the water quantity, using the measurement of the pump controller and that of the steam unit. When all failures are repaired, the mode is set to *normal*.

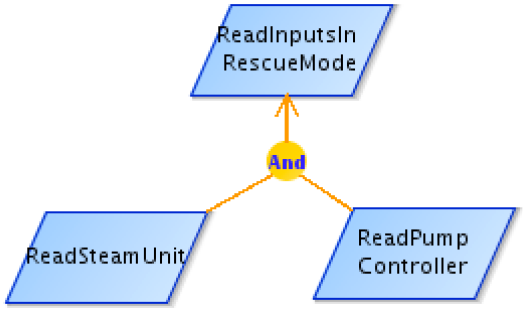


Figure 4.2 – Excerpt from the steam-boiler control system goal diagram

Figure 6.4 is an excerpt from the SysML/KAOS goal diagram representing the functional goals of the steam-boiler control system. To illustrate the SysML/KAOS method, we focus on the water level determination feature, when system operates in the *rescue* mode (goal **ReadInputsInRescueMode**). To achieve it, the system must read values from the steam unit (**ReadSteamUnit**) and pump controller (**ReadPumpController**), in order to estimate the amount of water in the boiler, since the water unit is unavailable.

In addition, *SysML/KAOS* includes a domain modeling language which combines the expressiveness of *OWL* [118], the constraints of *PLIB* [112] and the extensions needed to guarantee some relevant properties [61].

4.2.3 The SysML/KAOS Domain Modeling Language

Presentation

Domain models in SysML/KAOS are represented using ontologies. The domain modeling language [61,64] is built based on *OWL* [118] and *PLIB* [112]. Figure 4.3 is an excerpt from its metamodel. Each domain model corresponds to a refinement level in the SysML/KAOS goal model. The *parent* association represents the hierarchy of domain models. It allows a child domain model to access and refine elements defined in the parent domain model. A *concept* (instance of Concept) represents a collection of individuals with common properties. It can be *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

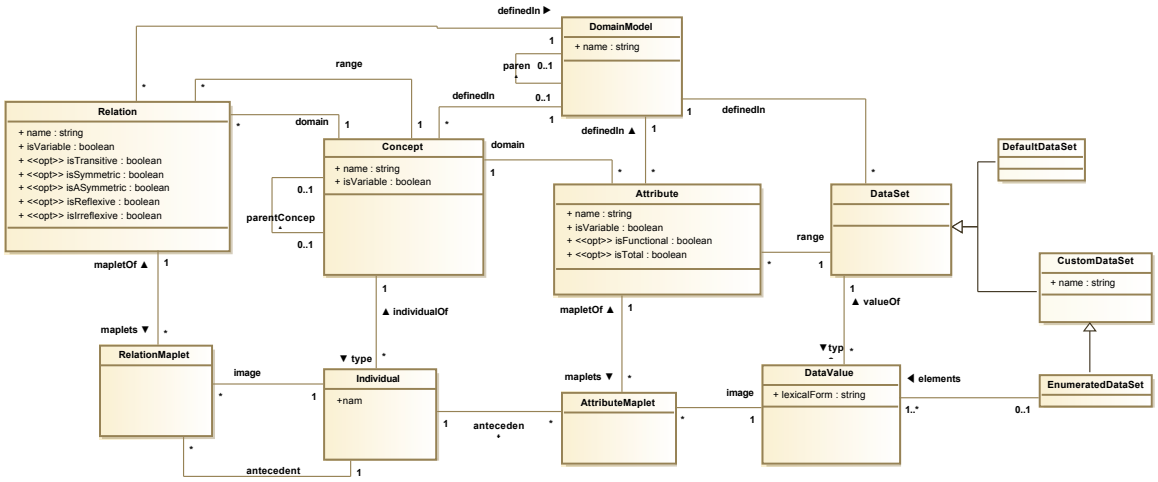


Figure 4.3 – Excerpt from the metamodel associated with the SysML/KAOS domain modeling language

Relation) capture links between concepts. An *attribute* (instance of Attribute) captures a link between a concept and a data set, knowing that a data set represents a collection of data values (instances of DataValue). A relation is instantiated between two individuals with a *relation maplet* (instance of RelationMaplet). An *attribute maplet* (instance of AttributeMaplet) instantiates an attribute between an individual (the antecedent) and a data value (the image). A relation or an attribute can be *variable* (*isVariable=TRUE*), if the set of its maplets can be updated. Otherwise, it is *constant* (*isVariable=FALSE*).

Illustration

Figure 6.5 represents the SysML/KAOS domain model associated with the root level of the goal diagram of Fig. 6.4. For readability purposes, we have decided to hide the representation of optional properties such as *isTransitive* or *isFunctional*. The steam-boiler entity is modeled as an instance of Concept named `SteamBoiler`. As in the case study, adding or deleting a steam-boiler is not considered, the property isVariable of `SteamBoiler` is set to `false`. The concept `SteamBoiler` has one individual named `SB`, representing the steam-boiler under the supervision of the system. The operating mode is modeled as an instance of Attribute named `operatingMode`, having `SteamBoiler` as domain, and as range, an instance of EnumeratedDataSet containing two data values (`normal` and `rescue`). The `isVariable` property of `operatingMode` is set to `true`, since it is possible to dynamically change

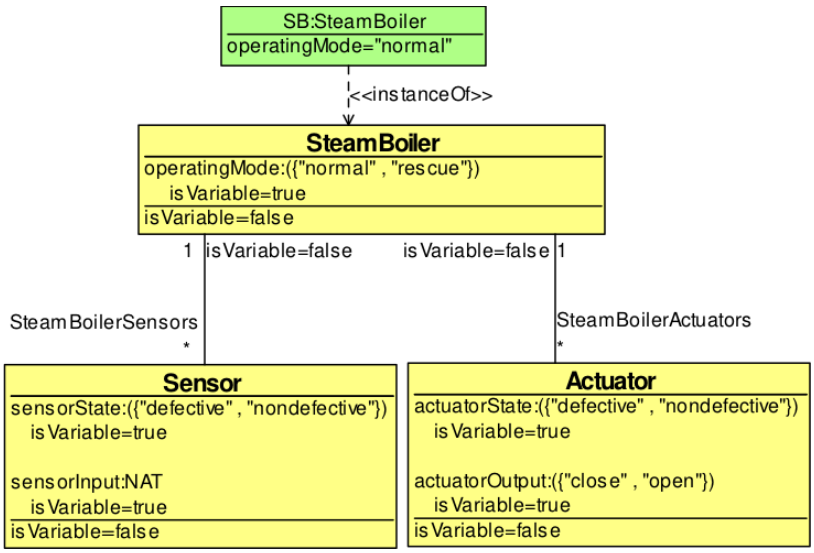


Figure 4.4 – **steam_boiler_controller_domain_model**: ontology associated with the root level of the goal diagram of Fig. 6.4

the mode in which the system operates. For SB, operatingMode is initialised to normal, since we consider that system starts in the normal mode. Associations between a steam-boiler and its sensors and actuators are modeled as instances of Relation. The relation named SteamBoilerSensors links the steam-boiler to its sensors and the one named SteamBoilerActuators links the steam-boiler to its actuators. Attribute sensorInput captures measurements obtained from sensors. Thus, its domain is the concept Sensor and its range is the instance of DefaultDataSet named NAT, representing the set of natural numbers [1]. Since the case study does not consider the dynamic addition or deletion of devices to a steam-boiler, properties isVariable of SteamBoilerSensors and SteamBoilerActuators are set to false.

At the root level, we consider that a steam-boiler has any number of sensors and any number of actuators, each of them belonging to one and only one steam-boiler. However, in the domain model of the first refinement level (steam_boiler_controller_domain_model_ref_1), we refine these constraints by enforcing that each steam-boiler has exactly three sensors and exactly one actuator. Concept Sensor is specialised into concepts WaterUnit, SteamUnit and

1. Data set NAT is used for simplification purposes. A more rigorous domain modeling, would represent the range of sensorInput as an instance of CustomDataSet called INCH, representing the set of lengths expressed in inches.

PumpController, while concept Actuator is specialised into concept Pump. We introduce three sensors (one individual of SteamUnit, one individual of PumpController, and one individual of WaterUnit) and one actuator (individual of Pump), linked to SB.

4.2.4 Translation of SysML/KAOS Goal and Domain Models to B System Specifications

Regarding the specification of the steam-boiler control system, the full *B System* model, verified using the *Rodin* tool [35], can be found in [131]. Each refinement level is the result of the translation of goal and domain models, except the body of events that are provided manually.

Translation of Domain Models

The translation rules are fully described in annex A.

SETS SteamBoiler; Sensor; Data_Set_1={normal, rescue}; Data_Set_2={defective, nondefective}
CONSTANTS SB, SteamBoilerSensors
PROPERTIES
axm1: $SB \in SteamBoiler$
axm2: $SteamBoilerSensors \in Sensor \rightarrow SteamBoiler$
VARIABLES operatingMode, sensorState, sensorInput
INVARIANT
inv1: $operatingMode \in SteamBoiler \rightarrow Data\_Set\_1$
inv2: $sensorState \in Sensor \rightarrow Data\_Set\_2$
inv3: $sensorInput \in Sensor \rightarrow \mathbb{N}$

Figure 4.5 – Excerpt of the *B System* specification obtained from the domain model of Fig. 6.5

Domain models give the structural part of the *B System* model and the initialisation of state variables. For instance, Figure 6.6 represents an excerpt of the *B System* specification obtained from the translation of the domain model of Fig. 6.5. Abstract concepts (concept without parent) such as SteamBoiler and enumerated sets such as {defective, nondefective} give *B System* sets. Individuals such as SB give *B System* constants defined as set items (axm1). Constant relations such as SteamBoilerSensors give *B System* constants defined as relations (axm2: operator "$\rightarrow$ " denotes a total function). Variables attributes such as operatingMode give *B System* variables defined as relations (inv1).

Translation of Goal Models

The formalisation of SysML/KAOS goal models is detailed in [102]. The proposed rules allow the generation of a formal model whose structure reflects the hierarchy of the SysML/KAOS goal diagram: one *B System* component is associated with each level of the goal hierarchy; this component defines one event for each goal. As the semantics of the refinement between goals is different from that of the predefined refinement between *B System* components, new proof obligations for goal refinement are generated in [102]. They complete the classic proof obligations.

Event ReadInputsInRescueMode $\widehat{=}$ **any** values
where
grd0: $operatingMode(SB) = rescue$
grd1: $values \in (SteamBoilerSensors^{-1}[\{SB\}] \cap sensorState^{-1}[\{nondefective\}]) \rightarrow \mathbb{N}$
then
act: $sensorInput := sensorInput \Leftarrow values$
END

Figure 4.6 – Excerpt of the *B System* specification obtained from the root level of the goal diagram of Fig. 6.4

Figure 4.6 represents an excerpt of the *B System* specification obtained from the translation of the goal diagram of Fig. 6.4. The specification includes event `ReadInputsInRescueMode` which represents the root goal. Guard `grd0` ensures that the event can only be triggered when the operating mode is *rescue*. In addition, guard `grd1` ensures that measurements are obtained from sensors of `SB` that are not defective, as natural numbers, with respect to the definition of attribute `sensorInput`. Finally, action `act` updates attribute `sensorInput` with the new measurements ("$\Leftarrow$" is the overload operator used to update associations in a relation).

4.3 Back Propagation of New B System Elements Into Domain Models

4.3.1 Motivations

Regarding the steam-boiler controller, up to now, by strictly interpreting the description we provided, we have considered two operating modes for the system: *normal* and *rescue*. Each of these modes assumes that the system is still able to determine the water level: using the water unit (*normal*) or using the pump controller and the steam unit (for the rescue mode). However, if a failure occurs on all sensors, the system will not be able to determine the water level. To ensure the consistency of system behaviour in this configuration, we need to add another operating mode: the *emergency* mode. It is the mode where the system must enter when a critical failure occurs [22]. Another operating mode which is required is the *degraded* mode where the water unit is available while the other sensors are defective: the system is able to detect the water level, but cannot check other indicators.

When these missings/inconsistencies are identified during the verification/validation of the *B System* model, by experts used to handling formal specifications, a common and widely observed behavior among industrials (including our industrial partners [17]) is to directly update the formal specification. To prevent these additions from introducing inconsistencies between a domain model and its *B System* specification, we must proceed with the back propagation of the updates. This is the purpose of the rules described in this section.

Other cases of additions, regarding the steam-boiler control specification problem, include: the addition of variables to distinguish between environment variables, which represent the actual state of the real environment and controller variables, which represent the measured value of the environment, as seen by the controller. This distinction becomes necessary when the specifier needs to handle measurement errors and control delays [111], in more concrete refinement levels. We could also consider

- the addition of the boiler length and width to allow the computation of its volume which is required for a better implementation of an estimation of the water level in case of a failure of the water unit;
- the addition of backup pumps with their controllers since one pump is insufficient to handle a rapid water evaporation (the case study description [22] outlines the use of four pumps).

Most of these updates are design choices introduced by experts who process the formal specification, during the system design process that is downstream of the requirements engineering process. These additions can also be needed to allow the construction of validation scenarios or the discharge of proof obligations.

4.3.2 Presentation of Back Propagation Rules

Up to now, we choose to support only the most common operations that can be performed within the formal specification, the domain model remaining the one to be updated in case of any major change such as additions or deletions of refinement levels. We are only interested here in the propagation of additions made within the *B System* specification. The back propagation rules are fully described in annex A.

This part provides an informal description of some typical back propagation rules with illustrations related to the specification of the steam boiler controller. The rules have been chosen because they reveal the subtleties related to back propagation concerns. Each rule defines its inputs (elements added to the *B System* specification) and constraints that inputs must fulfill. It also defines its outputs, that are 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, *d_x* designates the domain model element corresponding to *b_x*. In addition, when used, qualifier *abstract* denotes "without parent".

Rule 1 - Addition of Abstract Sets

Description: back propagation of the addition of an abstract set in the *B System* specification **AbstractSet:** *b_CO*

gives rise to

Concept: *d_CO* **Constraint 1:** *d_CO* is not associated to a parent concept **Constraint 2:** property *isVariable* of *d_CO* is set to *FALSE*

An abstract set *b_CO* (instance of class AbstractSet of the metamodel of Fig. 4.1) introduced in the *B System* specification gives a concept *d_CO* (instance of class Concept of the metamodel of Fig. 4.3) having its property *isVariable* set to *FALSE*.

For instance, the addition of an abstract set to represent the equipments of a steam-boiler, including its sensors and actuators, is back propagated with the introduction of a concept `Equipment` in the corresponding domain model. This allows for example to specify some event guards on all equipments to reduce the complexity and length of the specification.

Rule 2 - Addition of Constants or Variables Typed as Relations

Description: back propagation of the addition of a constant relation in the *B System* specification
Constant: *b_RE*
Property: $b\_RE \in b\_CO1 \leftrightarrow b\_CO2$ (resp. $b\_RE \in b\_CO1 \leftrightarrow b\_DS$) [a]
Constraint 1: *b_CO1* is the correspondence of a concept (instance of class Concept) *d_CO1*
Constraint 2: *b_CO2* (resp. *b_DS*) is the correspondence of a concept *d_CO2* (resp. data set (instance of class DataSet) *d_DS*)

a. the type of *b_RE* can be more constrained ($\nrightarrow$, $\rightarrowtail$, $\twoheadrightarrow$, etc.)

gives rise to

Relation (resp. **Attribute**): *d_RE* [a]
Constraint 1: the domain of *d_RE* is *d_CO1*
Constraint 2: the range of *d_RE* is *d_CO2* (resp. *d_DS*)
Constraint 3: property *isVariable* of *d_RE* is set to *FALSE*

a. properties of *d_RE* are set according to the type of *b_RE* ($\nrightarrow$, etc.)

The introduction in the *B System* specification of a constant *b_RE* typed as a relation can be back propagated, within the domain model, with the definition of a constant attribute (instance of class Attribute) or relation (instance of class Relation) *d_RE*:

(1) if the range of *b_RE* is the correspondence of a data set (instance of class DataSet), then *d_RE* must be an attribute;
(2) however, if the range of *b_RE* is the correspondence of a concept (instance of class Concept), then *d_RE* must be a relation.

For instance, the addition of constant $equipmentSteamBoiler \in Equipment \rightarrow SteamBoiler$, to link an equipment (individual of concept `Equipment` introduced in A.3.1) to its steam-boiler, is back propagated with the definition of a relation `equipmentSteamBoiler` between concepts `Equipment` and `SteamBoiler`.

If *b_RE* is a variable, then property *isVariable* of *d_RE* must be set to *true*. For instance, the addition of variable $lastMeasurementTimestamp \in Sensor \nrightarrow NAT$, to represent the timestamp of the last measurement reported by a sensor, is back propagated with the definition of a variable attribute `lastMeasurementTimestamp` in concept *Sensor*.

Rule 3 - Addition of Constants or Variables, Subsets of Correspondences of Concepts

Description: back propagation of the addition of a constant (resp. variable) typed as a subset of the correspondence of a concept
Constant (resp. **Variable**): *b_CO*
Property (resp. **Invariant**): $b\_CO \subseteq b\_PCO$
Constraint: *b_PCO* is the correspondence of a concept *d_PCO*

gives rise to

Concept: *d_CO*
Constraint 1: *d_CO* is associated to *d_PCO* with association parentConcept
Constraint 2: property *isVariable* of *d_CO* is set to *FALSE* (resp. *TRUE*)

A constant *b_CO* introduced in the *B System* specification and defined as a subset of *b_PCO*, the correspondent of a concept *d_PCO*, gives a concept *d_CO* having *d_PCO* as its parent concept (association parentConcept of the metamodel of Fig. 4.3).

For instance, the addition of a constant to represent the subclass of sensors that are pump controllers is back propagated with the introduction of a concept `PumpController` linked to concept `Sensor` using parentConcept.

If *b_CO* is a variable, then it is possible to dynamically add/remove individuals from concept *d_CO*. Thus, property *isVariable* of *d_CO* must be set to *TRUE*.

Rule 4 - Addition of Set Items

Description: back propagation of the addition of a set item in an enumerated set
SetItem: *b_elt*
Constraint 1: *b_elt* is an item of set *b_ES*
Constraint 2: *b_ES* is the correspondence of an enumerated data set *d_ES*

gives rise to

DataValue: *d_elt*
Constraint: *d_elt* is defined as an element of *d_ES*

An item *b_elt* (instance of class SetItem of the metamodel of Fig. 4.1) added to a set *b_ES* gives a data value *d_elt* (instance of class DataValue of the metamodel of Fig. 4.3) linked to the enumerated data set corresponding to *b_ES* with association element.

For instance, the addition of items *emergency* and *degraded* in the enumerated data set containing the operating modes of the steam boiler controller is back propagated with the definition of two data values linked to attribute `operatingMode` of concept *SteamBoiler*.

4.3.3 Formal Specification of Back Propagation Rules

The *Event-B* method has been chosen because it involves intuitive mathematical concepts, has a powerful refinement logic and is supported by industrial-strength tools. We have modeled back propagation rules as *Event-B convergent* events; each triggering of an event propagates an addition. In an *Event-B* specification, a *convergent* event is an event that can only be activated a finite number of times [8]. The system specification is consistent if it is proved that each convergent event will not be activated after a finite number of iterations. Thus, the system will always reach a state where all back propagations are done.

The formal specification of rules is based on the formal specification of *B System* and SysML/KAOS domain metamodels [65]. As a reminder, following the rules proposed in [91, 124], each class is translated into an *Event-B* variable, typed as a subset and containing the correspondences of the class instances. Furthermore, an association *r* from a class *A* to a class *B* to which the *ordered* constraint is attached is represented as a variable *r* typed through the invariant $r \in (A \rightarrow (\mathbb{N}_1 \rightarrow\!\!\!\!\!+ B))$ (operator "$\rightarrow$ " denotes a total function and operator "$\rightarrow\!\!\!\!\!+$ " denotes a partial function). If instances of *B* may have the same sequence number, then the invariant becomes $r \in (A \rightarrow \mathbb{P}_1 (\mathbb{N}_1 \times B))$ ("$\mathbb{P}_1 (A)$" denotes the set of non-empty parts of A). This is for example the case of association *Constant_isInvolvedIn_LogicFormulas* of the *B System* metamodel. To avoid ambiguity, the name of a variable, representing an attribute or association, is prefixed and suffixed with that of the class to which it is attached (**e.g.** `Concept_isVariable` or `Constant_definedIn_Component`).

Correspondence links between instances of a class A of the SysML/KAOS domain metamodel and instances of a class B of the *B System* metamodel are captured in a variable named *A_corresp_B* typed by the invariant $A\_corresp\_B \in A \rightarrowtail\!\!\!\!\!+ B$ (symbol "$\rightarrowtail\!\!\!\!\!+$" denotes a partial injective function). A partial injection is used because each instance, on both sides, must have at most one correspondence; it is partial because all additions are not back propagated at the same time. Moreover, we have proved that all additions made within a *B System* model will always be back propagated, in

a finite number of iterations of back propagation rules (See Section 4.4.1). Thus, the correspondence links will be total injections when the system will reach a deadlock state. The rest of this section provides an overview of the specification of some back propagation rules in order to illustrate the formalisation process and some of its benefits and difficulties. The full specification can be found in annex A.

Addition of a Constant, Subset of the Correspondence of an Instance of Concept (Rule 3)

This rule leads to two events: the first one is applied for a superset that is an abstract set and the second one for a superset that is a constant. Below is the specification of the first event.

Event constant_subset_concept_1 ⟨convergent⟩ ≙
  any d_CO b_CO d_PCO b_lg b_PCO
  where
  grd1: $b\_CO \in dom(Constant\_typing\_Property) \setminus ran(Concept\_corresp\_Constant)$
  grd2: $b\_lg = Constant\_typing\_Property(b\_CO)$
  grd3: $LogicFormula\_uses\_Operators(b\_lg) = \{1 \mapsto Inclusion\_OP\}$
  grd4: $(2 \mapsto b\_PCO) \in LogicFormula\_involves\_Sets(b\_lg)$
  grd5: $b\_PCO \in ran(Concept\_corresp\_AbstractSet)$
  grd6: $d\_PCO = Concept\_corresp\_AbstractSet^{-1}(b\_PCO)$
  grd7: $d\_CO \in Concept\_Set \setminus Concept$
  grd8: $Constant\_definedIn\_Component(b\_CO) \in ran(DomainModel\_corresp\_Component)$
  then
  act1: $Concept := Concept \cup \{d\_CO\}$
  act2: $Concept\_corresp\_Constant(d\_CO) := b\_CO$
  act3: $Concept\_parentConcept\_Concept(d\_CO) := d\_PCO$
  act4: $Concept\_isVariable(d\_CO) := FALSE$
  act5: $Concept\_definedIn\_DomainModel(d\_CO) := DomainModel\_corresp\_Component^{-1}(Constant\_definedIn\_Component(b\_CO))$
  END

The rule asserts that in order to propagate the addition of a constant, we need to evaluate its typing predicate. When it is typed as a subset of the correspondence of an instance of Concept, then it gives rise to an instance of Concept. We use an instance of LogicFormula, named *b_lg*, to represent the typing predicate of *b_CO* (grd2). Guards grd3 and grd4 ensure that *b_CO* is typed as a subset of *b_PCO*: grd3 ensures that b_lg is an inclusion predicate and grd4 ensures that b_lg involves b_PCO as second operand (Section 4.2.1). Guard grd5 ensures that the superset, *b_PCO*, is an abstract set corresponding to an instance of Concept. Guard grd6 constrains

d_PCO to be the correspondence of *b_PCO*. Guard grd7 ensures that *d_CO* is an instance of Concept which has never been used and action act2 defines *b_CO* as its correspondence. Finally, act3 defines *d_PCO* as its parent concept. Guard grd8 ensures that the event will be triggered only if the *B System* component, where *b_CO* is defined, corresponds to an existing domain model. Action act5 ensures that *d_CO* is defined in that domain model. In order to ensure the convergence of the rule, guard grd1 and action act2 ensure that the rule is triggered only for a constant which is already typed with a property and whose addition has not already been propagated ($b\_CO \notin ran(Concept\_corresp\_Constant)$).

The specification of the second event (when the superset is a constant) is different from the specification of the first one in four points:

grd4: $b\_PCO \in dom(Constant\_isInvolvedIn\_LogicFormulas)$
grd5: $(2 \mapsto b\_lg) \in Constant\_isInvolvedIn\_LogicFormulas(b\_PCO)$
grd6: $b\_PCO \in ran(Concept\_corresp\_Constant)$
grd7: $d\_PCO = Concept\_corresp\_Constant^{-1}(b\_PCO)$

Guard grd4 constrains the superset, *b_PCO*, to be a constant involved in a logic formula. Guard grd5 ensures that *b_PCO* is involved as the second operand of *b_lg*. Finally, guard grd6 ensures that *b_PCO* has a correspondence in the domain model and grd7 constrains *d_PCO* to be the correspondence of *b_PCO*.

Addition of a Variable, Subset of the Correspondence of an Instance of Concept (Rule 3 - dualed version)

Like the previous rule (Section 4.3.3), this rule leads to two events: the first one for when the superset is an abstract set and the second one for when the superset is a constant. Event specifications are different from the ones of the previous rule in four points:

grd1: $b\_CO \in dom(Variable\_typing\_Invariant) \setminus ran(Concept\_corresp\_Variable)$
grd2: $b\_lg = Variable\_typing\_Invariant(b\_CO)$
then
act2: $Concept\_corresp\_Variable(d\_CO) := b\_CO$
act4: $Concept\_isVariable(d\_CO) := TRUE$

In order to propagate the addition of a variable b_C0, we need to evaluate its typing invariant b_lg (grd2). The evaluation is done as in 4.3.3, since invariants and properties are generalised as logic formulas. However, the *isVariable* property of the concept has to be set to *TRUE* (act4). In addition, in order to ensure the convergence of the rule, guard grd1 and action act2 ensure that the rule is triggered only for a variable which is already typed with an invariant and whose addition has not already been propagated.

4.4 Discussion

4.4.1 Formal Verification and Validation

The translation rules, proposed in [65], allow the automatic translation of domain properties, modeled as ontologies, to *B System* specifications. The back propagation rules that we propose in this paper allow the automatic propagation of the addition of formal elements, in the structural part of a *B System* specification, within the SysML/KAOS domain models to which it is associated. The rules contribute to maintain a strong consistency between the domain models and the associated *B System* specification. It should be noted that a back propagation rule does not generally correspond to the inverse of a translation rule. The differences include for instance the number of elements to evaluate before a rule can be applied. For example, a variable concept *d_CO* results in the definition of an abstract set *b_CO* and a variable $x\_CO \subseteq b\_CO$, within the *B System* specification, while *(rule-i)* the addition of an abstract set *b_CO* is back propagated with the definition of a concept *d_CO*; *(rule-ii)* the addition of a variable $b\_x\_CO \subseteq b\_CO$ is back propagated with the definition of a variable concept *d_x_CO* having *d_CO* as parent concept.

Formally verifying the back propagation rules allowed us to prove their consistency, and to reveal several constraints (guards and invariants) that were missing when designing the rules informally (Section 4.3.2). This is for instance the case of guard grd1 of event constant_subset_concept_1 (Section 4.3.3), needed to ensure the convergence of the back propagation rule (each activation of the rule will always treat a constant whose addition has not yet been propagated). It is also the case for guards grd7 and grd8 and actions act4 and act5 that were necessary to discharge proof obligations.

We have proved that back propagation rules are isomorphisms (structure preserving), which guarantees that established links between elements of the *B System* specification are preserved between the corresponding elements in the domain models. These proofs have needed additional invariants. For instance, to prove that for each *B System* constant `b_xx` (correspondence of the concept `d_xx`), subset of the abstract set `b_pxx` (correspondence of the concept `d_pxx`), the concept `d_pxx` is the parent of the concept `d_xx`, considering that the parent concept corresponds to an abstract set (event `constant_subset_concept_1` of Section 4.3.3), the following invariant has been defined:

$$\begin{array}{l}\forall b\_xx, b\_pxx, b\_lg \cdot ((b\_xx \in dom(Constant\_typing\_Property) \cap ran(Concept\_corresp\_Constant) \\ \wedge\ b\_lg = Constant\_typing\_Property(b\_xx) \\ \wedge\ LogicFormula\_uses\_Operators(b\_lg) = \{1 \mapsto Inclusion\_OP\} \\ \wedge\ b\_pxx \in ran(Concept\_corresp\_AbstractSet) \\ \wedge\ (2 \mapsto b\_pxx) \in LogicFormula\_involves\_Sets(b\_lg)) \\ \Rightarrow (Concept\_corresp\_Constant^{-1}(b\_xx) \in dom(Concept\_parentConcept\_Concept) \\ \wedge\ Concept\_corresp\_AbstractSet^{-1}(b\_pxx) \\ = Concept\_parentConcept\_Concept(Concept\_corresp\_Constant^{-1}(b\_xx))))\end{array}$$

For each constant `b_xx`, typed by a property `b_lg` which defines `b_xx` as a subset of a constant `b_pxx`:

(1) `b_lg` is an inclusion predicate:
$LogicFormula\_uses\_Operators(b\_lg) = \{1 \mapsto Inclusion\_OP\}$;

(2) `b_lg` types `b_xx`:
$b\_lg = Constant\_typing\_Property(b\_xx)$;

(3) `b_lg` involves `b_pxx` as second operand:
$(2 \mapsto b\_pxx) \in LogicFormula\_involves\_Sets(b\_lg)$

the correspondence of `b_pxx` in the domain model is the parent concept of the correspondence of `b_xx`:

$Concept\_corresp\_AbstractSet^{-1}(b\_pxx)$
$= Concept\_parentConcept\_Concept(Concept\_corresp\_Constant^{-1}(b\_xx))$.

We have also proved that a finite number of additions of elements in a *B System* specification will be propagated in a finite number of iterations of back propagation rules. To prove it, we have defined *variants* that express the differences between the sets of added elements and the sets of elements already propagated. For instance, the part of the variant related to events handling the back propagation of variable subsets (Section 4.3.3), relations and attributes, is

$dom(Variable\_typing\_Invariant) \setminus (ran(Concept\_corresp\_Variable)$
$\cup\ ran(Relation\_corresp\_Variable) \cup ran(Attribute\_corresp\_Variable))$

(the set of variables defined within the *B System* model minus the union of the sets of variables that are correspondences of variable concepts, variable relations and variable attributes). Thus, at each activation of one of these events, we ensure that the addition of a formal variable will be propagated (to a variable concept, to a variable relation or to a variable attribute) and a link will be added to *Variable_typing_Invariant*.

Table 4.1 summarises the key characteristics of the Rodin project corresponding to the *Event-B* specification of metamodels and rules (translation and back propagation rules). The specification includes two refinement levels.

Table 4.1 – Key characteristics of the *Event-B* specification of rules

Characteristics	Root level	First refinement level
Events	3	50
Invariants	11	104
Proof Obligations (PO)	37	1123
Automatically Discharged POs	27	257
Interactively Discharged POs	10	866

The validation of the consistency of the formal specification required the discharge of *1160* proof obligations of which *876* (75.52 %) have required manual proofs. Many proofs were not discharged automatically due to the use of function operators ($\nrightarrow, \rightarrowtail, \rightarrow, \twoheadrightarrow$) in variable definitions and in invariants.

4.4.2 Specification Process

Figure 4.7 provides an overview of the specification process. The first step is to use SysML/KAOS languages to build models of the system and of its application domain. The SysML/KAOS goal modeling language [92] is used to extract and represent system requirements from artifacts that describe stakeholder needs. The SysML/KAOS domain modeling language [61] is used to extract and represent application domain entities and their properties. The second step is to translate the goal model into a *B System* specification, following the rules provided in [102], and to complete the specification with the result of the translation of domain models, following the formally verified rules provided in [65]. Goal models provide the **behavioral part** (events) of the specification and domain models provide its **structural part** (sets, constant and their properties, variables and their invariant) and the initialisation of state variables. It remains to manually specify the body of events and to formally verify and validate the specification with *B System* tools (step *3* of Fig. 4.7). Tueno *et al.* [58] illustrate this part of the specification process on the hybrid ERTMS/ETCS level 3 case study. However, the structural part is very

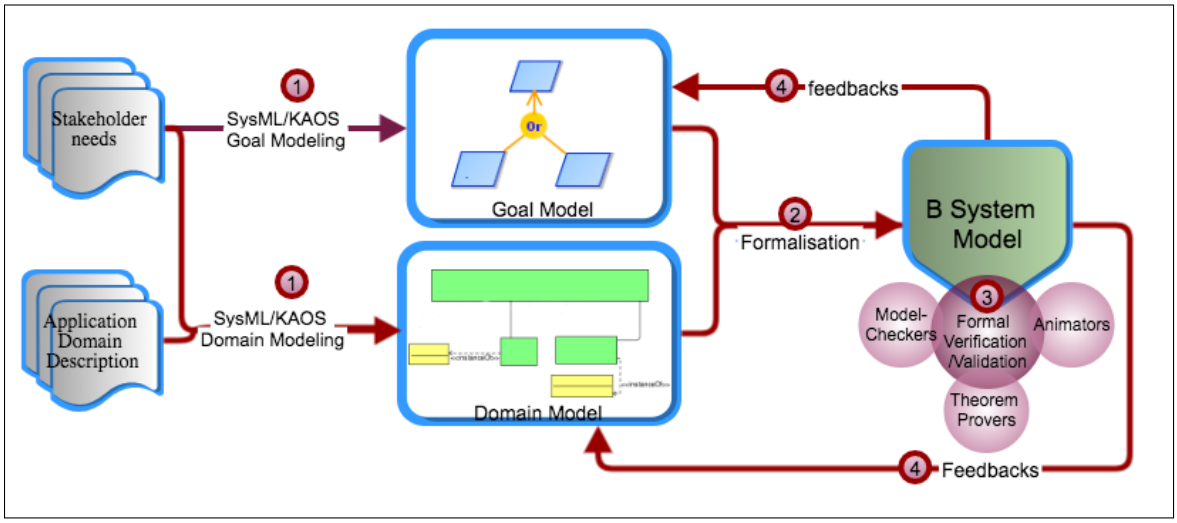


Figure 4.7 – Overview of the specification process

often updated during the specification of events or to discharge proof obligations. This paper provides formally verified rules to back propagate update operations within domain models (step *4* of Fig. 4.7). The back propagation operations are feedbacks from *B System* specifiers to domain modelers provided, in addition to translation mechanisms provided in [65], to ensure consistency between models and their *B System* specifications. When updates are manually made so as to meet the preconditions of a rule, the rule is automatically triggered and propagates updates to the corresponding domain model. This paper also discuss how and why the back propagation rules have been formally verified.

To manage the complexity of the system, SysML/KAOS considers its decomposition into subsystems. Therefore, the goal modeling language allows the assignment of system requirements to subsystems responsible of their achievement. However, Matoussi *et al.* [102] do not provide mechanisms to ensure that each subsystem specification is consistent with the specification of the high-level system and especially with requirements assigned to the subsystem. These mechanisms are defined in [59] and illustrated on the steam-boiler control specification problem.

4.4.3 Related Work

The review summarised here completes the state of the art that Tueno *et al.* have introduced in [65]. In [43], a platform called *ASSERT* is proposed for requirements capture and analysis. Domain models are captured using a controlled-English modeling language called SADL (Semantic Application Design Language) based on *OWL*. A domain model is represented by its classes, class individuals, properties and axioms. However, the domain modeling approach does not allow the distinction between a static element (whose state is independent of the system behavior) and a dynamic element. The steam-boiler domain is modeled with two ontologies: the abstract ontology defines some basic entities such as *System*, *component* and *Equipment*; the concrete ontology defines entities related to the steam-boiler such as *Pump* and *PumpController*. Equipments are components of systems. It distinguishes the entity *SteamBoiler* which is a type of *Equipment* from the entity *SteamBoilerSystem* which is a type of *System*. The operating mode is a property of *SteamBoilerSystem* and the water level is a property of *SteamBoiler*. However, it only considers the maximum of the water level and not its minimum. Requirements are expressed with elements defined in domain models, using a controlled-English modeling language called SRL (SADL Requirements Language). For requirements analysis, [43] proposes the translation of SRL definitions into a first-order-logic if-then form, independent of the target analysis tool. Another step allows the conversion of requirements, in the intermediate form, to the target analysis tool representation. Some rules are provided, in an informal way, for the translation of requirements.

In [37], Carreira *et al.* uses a UML object diagram to represent entities and associations related to the domain of a system. The representation includes a class *Controller* linked to a class *Boiler* which is composed of classes *Pump*, *Valve*, *SteamMeasurer* and *WaterMeasurer*. An object-oriented specification language named *OBLOG* [119] is used to represent requirements. The OBLOG specifications are then translated into a LOTOS model [28] for formal verifications. As in [43], rules are provided, in an informal way, for the translation of requirements.

While in [37, 43], domain models are used to set a domain specific language for the expression of requirements, the SysML/KAOS method allows the models (goal models and domain models) to evolve independently. The models have only to follow the same refinement logic. A one step translation allows the generation of a *B System* specification from SysML/KAOS goal models. The specification is then completed with the result of the translation of SysML/KAOS domain models. Thus, we are able to precisely propagate an update on the *B System* specification within the corresponding SysML/KAOS model. This allows each expert to update/enrich the system specification using the formalism that suits him while ensuring the overall consistency of system models. The approach will be supported by a platform called

Openflexo [109], allowing the federation of these goal, domain and formal models. The platform currently supports goal and domain modeling. The SysML/KAOS method also differs from that of [37,43] on the formal method used for verifications and validations. Indeed, the SysML/KAOS method allows the generation of *B System* specifications, which make it possible to take advantage of the range of tools that support the *B* method [10], largely and positively assessed on industrial projects for more than 25 years [93]. Finally, the SysML/KAOS domain modeling language is certainly based on *OWL* [118], but it defines additional constraints and restrictions, based on *PLIB* [112], and enriched, allowing a more reliable representation of critical domains.

In [99], domain model elements are directly specified in *Event-B*. The SysML/KAOS method, on the other side, uses ontologies for the representation of domain entities, their relationships and their static and dynamic properties. They give the structure of the *B System* model. The use of high level graphical modeling languages, as stated in [58,65], has several advantages, such as a better reusability, maintainability and readability of models. They also facilitate validations with stakeholders.

4.5 Conclusion and Future Work

Work done on case studies [58,133], regarding the SysML/KAOS requirements engineering method, and industrial returns reveal that it often appears that new elements need to be added to the *B System* specification obtained from SysML/KAOS models. This paper describes how elements manually added into the *B System* specification can be automatically back propagated to SysML/KAOS domain models. The rules contribute to maintain a strong consistency between domain models and their associated formal specification. They have been formalised and verified using *Rodin* [35], an industrial-strength tool supporting the *Event-B* method. We have proved that they are consistent and structure preserving.

Work in progress is aimed at studying the back propagation of updates on links between elements and element typings. We are also working on integrating the rules within the open-source platform *Openflexo* [109] which federates the various contributions of *FORMOSE* project partners [17].

Chapitre 5

Ajustement du langage SysML/KAOS de modélisation du domaine

Résumé

Ce chapitre décrit les ajustements réalisés, sur le langage *SysML/KAOS* de modélisation du domaine, à la suite de l'évaluation de la méthode SysML/KAOS dans le cadre de la spécification formelle des exigences du protocole de communication ferroviaire *Saturn*. Le protocole *Saturn* est un protocole proposé par *ClearSy Systems Engineering* afin de garantir la robustesse des échanges entre agents au sein d'une infrastructure ferroviaire. Il a été développé et implémenté à partir d'exigences non structurées représentées par de volumineux documents textuels. Il a donc été question de spécifier, vérifier et valider les exigences de Saturn afin de garantir la cohérence de son comportement et faciliter la mise en oeuvre de variantes.

La méthode SysML/KAOS a permis de définir les cinq premiers niveaux de raffinement de la spécification *B System* du protocole. Toutefois, plusieurs manquements ont été identifiés. Ce chapitre décrit les ajustements réalisés au sein du langage de modélisation du domaine et des règles y afférentes afin de combler ces manquements. La méthode *Event-B* a permis de spécifier et vérifier la version ajustée du langage et des règles. Ces derniers ont par la suite été implémentés au sein de la plateforme *Openflexo* et au sein de l'outil *JetBrains MPS* de modélisation du domaine. L'implémentation *JetBrains MPS* permet de construire les modèles de domaine en utilisant

une syntaxe textuelle pour en générer une spécification *B System* tandis que l'implémentation *Openflexo* permet d'utiliser une syntaxe graphique et assure la fédération entre modèles de domaine, modèles de buts, et modèles *B System*.

Commentaires

La contribution ici réside dans la définition d'une version ajustée du langage SysML/KAOS de modélisation du domaine nécessaire pour permettre la spécification des exigences de systèmes d'ingénierie (systèmes implémentant des processus ou supportant des activités d'ingénierie) à l'exemple du protocole *Saturn*. L'ajustement s'étend également aux règles de traduction (Chapitre 3) et de propagation (Chapitre 4) ; la méthode *Event-B*, supportée par la plateforme *Rodin*, ayant permis de prouver que les règles restent cohérentes, complètes, convergentes et isomorphes. L'annexe B décrit entièrement les définitions informelles puis formelles du langage et des règles ajustés. La vérification formelle des règles a nécessité la démonstration de *1416* obligations de preuve parmis lesquelles seul *275* ont nécessité une intervention manuelle (confère Sect. B.1 de l'annexe B). Ainsi, vérifier la version ajustée du langage et des règles a nécessité moins d'effort manuel. Ceci est dû (1) à la simplification des évènements *Event-B* (représentant les règles), dont le nombre a également été réduit, introduite par les ajustements réalisés, et (2) à la définition d'une tactique de preuve automatique plus efficace.

Dans un contexte où le domaine d'application du système est constitué d'une part de concepts ou entités et des liens entre eux, et d'autre part de leurs caractéristiques ou attributs quantifiés ou qualifiés par des données, la version initiale du langage est nécessaire. Elle permet en effet d'expliciter la distinction entre ce qui est tangible (concepts, individus et relations) et ce qui ne l'est pas (types de données, données, attributs et valeurs d'attributs) et qui sert juste à caractériser des éléments de la première catégorie [25]. C'est le cas de la grande majorité des systèmes d'information à l'exemple d'un système de gestion de librairies. Par contre, (i) lorsque les données regroupées en ensembles, leurs liens et les contraintes qui régissent leurs mises en relation, constituent un aspect essentiel du domaine d'application du système, (ii) ou lorsque la distinction entre tangible et intangible n'est pas claire ou nécessaire, la version ajustée du langage est plus appropriée. C'est le cas de la grande majorité des systèmes d'ingénierie et des systèmes industriels.

Cette version ajustée du langage, évaluée sur l'étude de cas *Saturn* [129], a fait l'objet d'un article accepté et publié [60] dans le cadre de la 14^{e} édition de la conférence internationale *International Conference on Software Technologies (ICSOFT)*. Une réédition de cet article, rédigée en français, a fait l'objet d'un article accepté et publié [62] dans le cadre des 18^{e} journées *Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL)*. Elle propose et illustre l'utilisation des diagrammes d'états-transitions algébriques (ASTDs) [66] afin de représenter les changements d'états des variables du modèle de domaine au fur et à mesure que le système satisfait ses buts. Il est à noter que les ASTDs peuvent être traduits en spécification *B System* [105] afin de compléter la spécification formelle issue de la traduction des modèles de buts et de domaine SysML/KAOS.

Par ailleurs, dans [60], les modèles de domaine sont construits et traduits en spécifications *B System* en utilisant l'outil *JetBrains MPS* de modélisation du domaine [56] tandis que dans [62], c'est la plateforme *Openflexo* [109] qui est utilisée.

Ces contributions et les articles sus-cités ont été élaborés par mes soins en tenant compte des remarques et commentaires issus de mon équipe d'encadrement. En ce qui concerne l'étude de cas, le protocole *Saturn* a tout d'abord été spécifié par l'Ingénieur *Hector Ruiz Barradas* de *ClearSy System Engineering*. La spécification a ensuite été ajustée à l'issue de plusieurs séances de travail où j'étais impliqué ainsi que les Ingénieur *Hector Ruiz Barradas* et Professeure *Régine Laleau*. Les ajustements ont finalement été validés au cours d'une séance plénière de travail qui impliquait les partenaires du projet *FORMOSE*.

A Formal Requirements Modeling Approach: Application to Rail Communication

Steve J. Tueno Fotso
Université Paris Est Créteil, LACL, Créteil, France
Université de Sherbrooke, GRIL, Québec, Canada
steve.jeffrey.tueno.fotso@usherbrooke.ca

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

Hector Ruiz Barradas
ClearSy System Engineering, Aix-en-Provence
hector.ruiz-barradas@clearsy.com

Marc Frappier
Université de Sherbrooke, GRIL, Québec, Canada
Marc.Frappier@usherbrooke.ca

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

Keywords: *Saturn* Rail Communication Protocol, Requirements Engineering, Formal Models, Domain Modeling, Railway, *SysML/KAOS*, *B System*, *Event-B*

Abstract

This paper is about the formal specification of requirements of a rail communication protocol called *Saturn*, proposed by ClearSy systems engineering, a French company specialised in safety critical systems. The protocol was developed and implemented within a rail product, widely used, without modeling, verifying and even documenting its requirements. This paper outlines the formal specification, verification and validation of Saturn's requirements in order to guarantee its correct behavior and to allow the definition of slightly different product lines. The specification is performed according to SysML/KAOS, a formal requirements engineering method developed in the *ANR FORMOSE* project for critical and complex systems. System requirements, captured with a goal modeling language, give rise to the behavioral part of a *B System* specification. In addition, an

ontology modeling language allows the specification of domain entities and properties. The domain models thus obtained are used to derive the structural part of the *B System* specification obtained from system requirements. The *B System* model, once completed with the body of events, can then be verified and validated using the whole range of tools that support the *B* method. Five refinement levels of the rail communication protocol were constructed. The method has proven useful. However, several missing features were identified. This paper also provides a formally defined extension of the modeling languages to fill the shortcomings.

5.1 Introduction

Refinement-based methods that support proof-by-construction help to progressively construct the specification of a complex system while ensuring its correctness. In recent years, these methods have been widely used on large scale projects in critical areas such as railway or aeronautics, thanks in particular to the emergence of industrial strength formal methods such as *B* [10] or *Event-B* [8]. A major difficulty identified in such projects lies in the construction of the initial formal model, based on needs identified by stakeholders [8, 94]. When poorly conducted, this step is responsible for the vast majority of critical failures [94].

Our work focuses on the formal requirements modeling of critical and complex systems. *SysML/KAOS* [92], as a requirements engineering method, is chosen because it includes an expressive and intuitive goal modeling language [92] to represent system requirements extracted from artifacts that describe stakeholder needs. In addition, SysML/KAOS includes a domain modeling language [61] to represent application domain entities and their properties using ontologies. Furthermore, the rules required to generate a *B System* specification [41] from goal and domain models are defined and the most relevant ones have been formally verified [65]. Goal models give the *behavioral part* (events) of the specification [102] while domain models provide its *structural part* (sets, constants and their properties, variables and their invariant) and the initialisation of state variables [65]. Once the event bodies are specified, the *B System* model can be formally verified and validated to assess the requirements. This can be done using the full range of tools that support the *B* method [10], largely and positively assessed on industrial projects for more than 25 years [93].

SysML/KAOS is supported by integrated development environments *Openflexo* [109] and *Atelier B* [41]. Openflexo supports goal modeling while Atelier B supports the specification, verification and validation of *B System* models based on the semantics of SysML/KAOS modeling languages. Domain models, on the other hand, are constructed and translated to *B System* using a tool [56] implemented on top of *Jetbrains MPS* [87] and PlantUML [116].

Saturn [129] is a rail communication protocol, proposed by *ClearSy*, which deals with exchanges of communication frames between rail agents through a bus. The protocol was developed and implemented within a rail product widely used, without modeling, verifying and even documenting its requirements. This paper outlines the formal specification, verification and validation of Saturn's requirements in order to guarantee the correctness of its behavior and to allow the definition of slightly different product lines. SysML/KAOS has provided a methodical and structured way for the formal specification of requirements. Furthermore, it ensures a better reusability and readability of models and a strong traceability between models. The specification is decomposed into five refinements. The use of SysML/KAOS on this case study led us to extend the domain modeling language and make it more suitable for use in system modeling: the language, described in [61, 64] 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. In addition, the translation rules [63, 65] have been updated to match the adjusted language and formally verified.

The remainder of this paper is structured as follows: Section 2 briefly describes the *B System* formal method, the SysML/KAOS requirements engineering method and its goal and domain modeling languages, and the translation of SysML/KAOS models. Follows a presentation, in Section 3, of the work done on the case study and of updates performed on the SysML/KAOS domain modeling language. Finally, Section 4 discuss work done and Section 5 reports our conclusion and future work.

5.2 Background

5.2.1 Event-B and B System

Event-B [8] is an industrial-strength formal method for *system modeling*. It allows the incremental construction of system specifications, using stepwise refinement, and the proof of useful properties. *B System* is an *Event-B* syntactic variant proposed by *ClearSy*, an industrial partner in the *FORMOSE* project [17], and supported by *Atelier B* [41]. It shares the same semantics with *Event-B*.

A *B System* specification consists of components. Each component can be either a system or a refinement and it may define static or dynamic elements. A refinement is a component which refines another one in order to concretise the system construction: addition of functionalities or specification of the achievement of some purposes. Constants, abstract and enumerated sets (user-defined types), and their properties, constitute the static part. The dynamic part includes the representation of system state using variables constrained through invariants (first-order predicates that constrain the possible values that the variables may hold) and updated through events.

5.2.2 SysML/KAOS Goal Modeling

SysML/KAOS [92] is a requirements engineering method which combines the traceability provided by *SysML* [78] with goal expressiveness provided by KAOS [138]. 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.

5.2.3 SysML/KAOS Domain Modeling

Domain models in SysML/KAOS are represented using ontologies. These ontologies are expressed using the SysML/KAOS domain modeling language [61, 64], which is built based on *OWL* [118] and *PLIB* [112].

Each domain model corresponds to a refinement level in the SysML/KAOS goal model. The *parent* association represents the hierarchy of domain models. A domain model can define multiple elements. *Concepts* (instances of Concept) denote collections of *individuals* (instances of Individual) with common properties. A concept can be declared *variable* when the set of its individuals can be updated by adding or deleting individuals. Otherwise, it is considered to be *constant*.

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). *Predicates* (instances of Predicate) are used to represent constraints between different elements of the domain model in the form of *Horn clauses*.

5.2.4 Translation of SysML/KAOS Models

The formalisation of SysML/KAOS goal models is detailed in [102]. The proposed rules allow the generation of a formal model whose structure reflects the hierarchy of the SysML/KAOS goal diagram: one component is associated with each level of the goal hierarchy; this component defines one event for each goal. As the semantics of the refinement between goals is different from that of the refinement between *B System* components, new proof obligations for goal refinement are defined in [102]. They depend on the goal refinement operator used and complete the *B System* proof obligations for invariant preservation and for event feasibility. For instance, the following proof obligations formalise the AND refinement of an abstract goal G into two concrete goals G_1 and G_2 (for an event `G`, `G_Guard` represents the guards of G and `G_Post` represents the post condition of its actions):

- $G_1\_Guard \Rightarrow G\_Guard$
- $G_2\_Guard \Rightarrow G\_Guard$
- $(G_1\_Post \wedge G_2\_Post) \Rightarrow G\_Post$

It should be noted that variables updated by subgoals must be distinct.

However, the *B System* specification generated from goal diagrams does not contain the static part and the state variables. These elements are provided by the translation of SysML/KAOS domain models. The corresponding rules are fully described in [63] and their formal verification is described in [65]. In short, domain models identify *B System* components. A concept without a parent gives a *B System* abstract set. Each concept C, with parent PC, gives a formal constant, subset of the correspondent of PC. Relations and attributes give formal relations.

5.3 Specification of the Saturn Communication Protocol

5.3.1 Main Characteristics of the Protocol

Saturn describes exchanges of communication frames between different agents connected via a bus so as to ensure high robustness and availability [129]. It deals with one active gateway (and possibly seven passive ones), several input/output agents (1-128) and an innovative ring network. Input/output agents can be secure or unsecure. Input agents provide boolean data. The data are periodically collected by the gateway, transformed and the result is made available to output agents through the ring network.

5.3.2 Goal Modeling

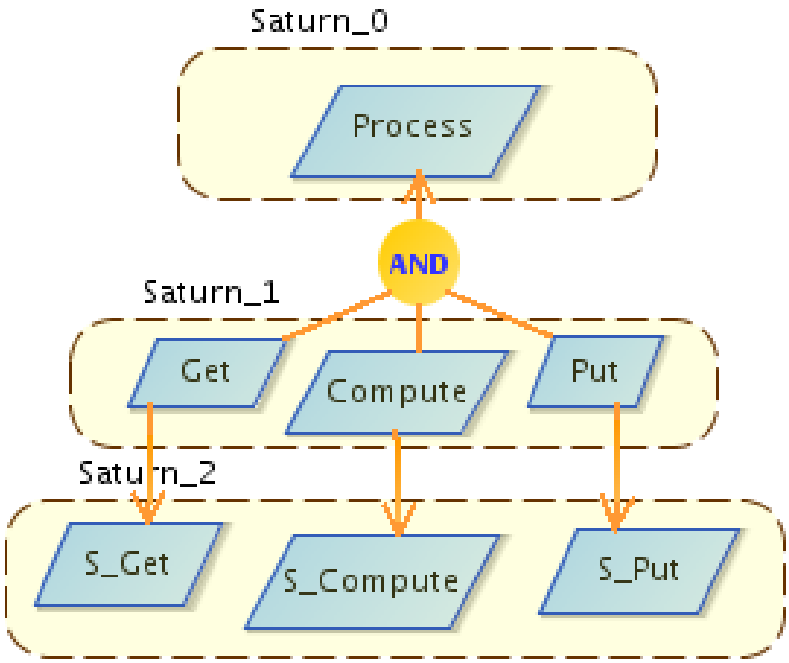


Figure 5.1 – Excerpt from the Saturn protocol goal diagram

Figure 5.1 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: `FB` is a boolean function that transforms inputs into outputs. The purpose gives the most abstract goal *Process* of the goal diagram. Goal *Process* is refined using the AND operator to introduce a goal *Get* for input data acquisition from input agents and a goal *Put* to make the result available to output agents. The second refinement level is a *data refinement* which refines goals defined within the first refinement level to take into account multiplicities of input and output agents. Thus, input data acquisition (goal *S_Get*) generates a boolean array, the computation (goal *S_Compute*) becomes a transformation between arrays and result delivery (goal *S_Put*) transfers the resulting array to output agents.

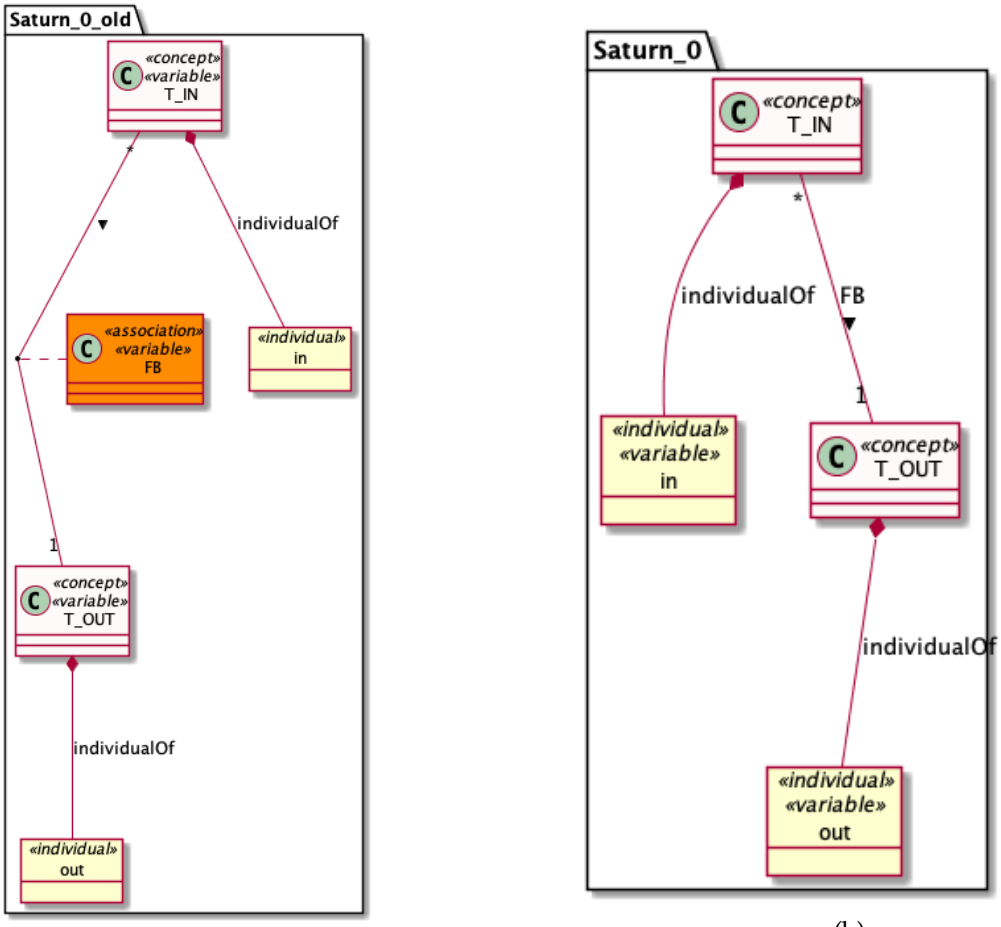


Figure 5.2 – ***Saturn_0***: root level ontology

5.3.3 Domain Modeling

First Attempt

Figure 5.2 (a) is an attempt to represent the domain model associated with the root level of the goal diagram of Fig. 5.1 using the SysML/KAOS domain modeling language previously described. It is illustrated using the syntax proposed by the *SysML/KAOS domain modeling tool* [56]. For readability purposes, we have decided to hide the representation of optional characteristics. 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, inputs and outputs change dynamically. In the domain model of Fig. 5.2 (a), a workaround consisted in considering that concepts T_IN and T_OUT and relation FB are variables. In the *B System* specification, the variability is reflected through the definition of a variable subset [63, 65]: for instance, the variable concept T_IN gives a *B System* abstract set T_IN and a variable x_T_IN⊆T_IN which contains, at any system state, the current value of the input. 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 ↦ out1 from x_FB; (2) withdrawal of individual in1 from x_T_IN; (3) withdrawal of individual out1 from x_T_OUT; (4) addition of individual in2 in x_T_IN; (5) addition of individual out2 in x_T_OUT; and (6) addition of maplet in2 ↦ out2 in x_FB. Too many actions are thus required and the modeling does not conform to Saturn's specification. Indeed, from a conceptual point of view: (1) the input data type must be constant (corresponds to the set of n-tuples of booleans, when considering n input agents); (2) the output data type must be constant; (3) the computation function FB is hard-coded and is therefore constant. What should change are individuals representing input and output states. 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. 5.1). 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 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;

- refine an individual with a maplet or a data value.

We have therefore identified the need to build a generalisation of the domain modeling language to enrich its expressiveness while preserving the fundamental constraints identified in [61, 64]. A major contribution of this new metamodel is that it merges 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 such as *OWL* [118]. 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 (annex B).

The Revised Domain Modeling Language

Figure 5.3 represents the revised SysML/KAOS domain metamodel. Major updates were made within the elements in pink. Classes Concept and DataSet have been merged into class Concept. In addition, classes Individual and DataValue have been merged into class Individual. A concept can now be an enumeration (*isEnumeration=TRUE*) if all its individuals are defined within the domain model. 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. Class Association is used to merge classes Relation and Attribute. *Maplet individuals* (instances of MapletIndividual) capture associations between individuals. 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. Class LogicalFormula replaces class Predicate to represent constraints between domain model elements. A defined concept (instance of class DefinedConcept) is a concept for which the type is provided by a logical formula.

Additional constraints 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 fully defined in annex B using the *B* syntax [8]. For instance:

- $x \in \mathsf{Concept} \setminus \mathsf{Association} \Rightarrow individualOf^{-1}[\{x\}] \cap MapletIndividual = \emptyset$:
  if a concept **x** is not an association, then no individual of **x** can be a maplet individual.
- $x \in MapletIndividual \cap dom(antecedent) \Rightarrow antecedent(x) \in domain(individualOf(x))$:
  antecedents of a maplet individual must be individuals of the domain of its association.

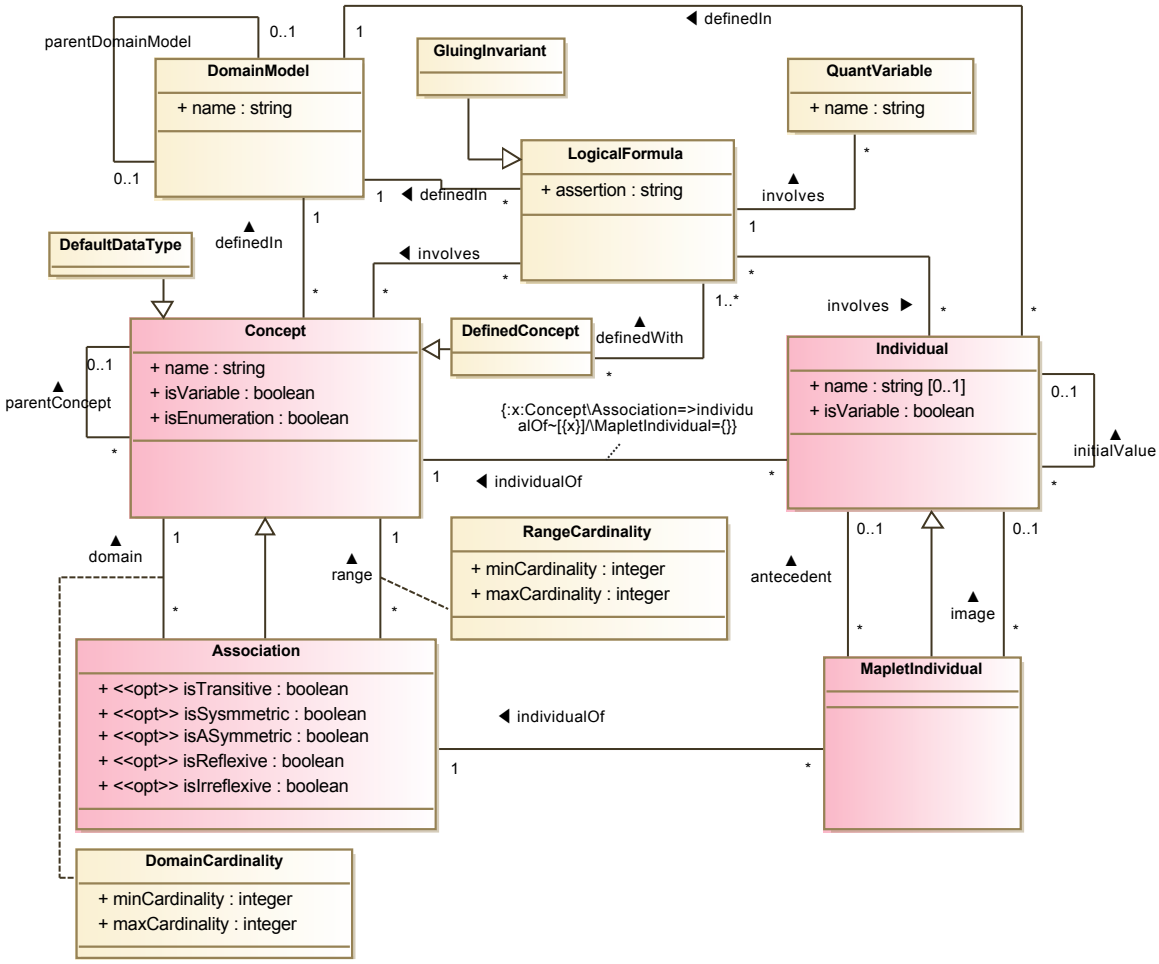


Figure 5.3 – The revised SysML/KAOS domain metamodel

- $x \in \text{Concept} \setminus (\text{Association} \cup dom(parentConcept)) \Rightarrow Concept\_isVariable(x) = FALSE$: every abstract concept (that has no parent concept) that is not an association must be constant. Abstract concepts that are associations can be variable.
- $x \in \text{Concept} \wedge Concept\_isEnumeration(x) = TRUE \Rightarrow Concept\_isVariable(x) = FALSE$: every concept that is an enumeration must be constant.
- $(ind \in \text{MapletIndividual} \cap dom(antecedent)$
  $\cap\, dom(image) \wedge Individual\_isVariable(ind) = FALSE)$
  $\Rightarrow (Individual\_isVariable(antecedent(ind)) = FALSE$
  $\wedge\, Individual\_isVariable(image(ind)) = FALSE)$:

antecedents and images of constant maplet individuals must be constant. A change in state of the antecedent or of the image of a maplet individual **x** leads to a change in state of **x**.

- $(x \in \text{Association} \wedge Concept\_isVariable(x) = FALSE)$
  $\Rightarrow (Concept\_isVariable(domain(x)) = FALSE$
  $\wedge\ Concept\_isVariable(range(x)) = FALSE)$:
  domains and ranges of constant associations must be constant.

The Saturn Protocol Domain Model

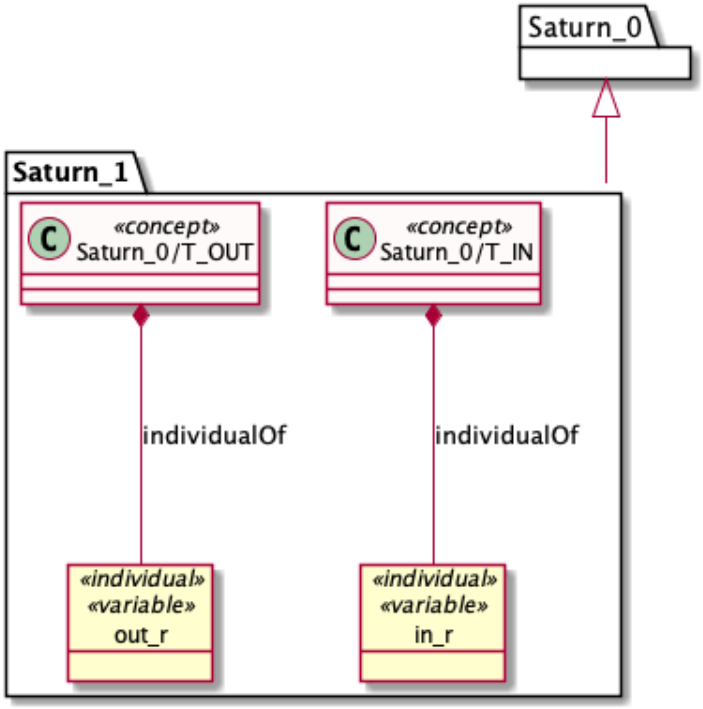


Figure 5.4 – ***Saturn_1***: ontology associated with the first refinement level

Figures 5.2 (b), 5.4 and 5.5 represent domain models associated with the refinement levels of the goal diagram of Fig. 5.1 using the updated SysML/KAOS domain modeling language.

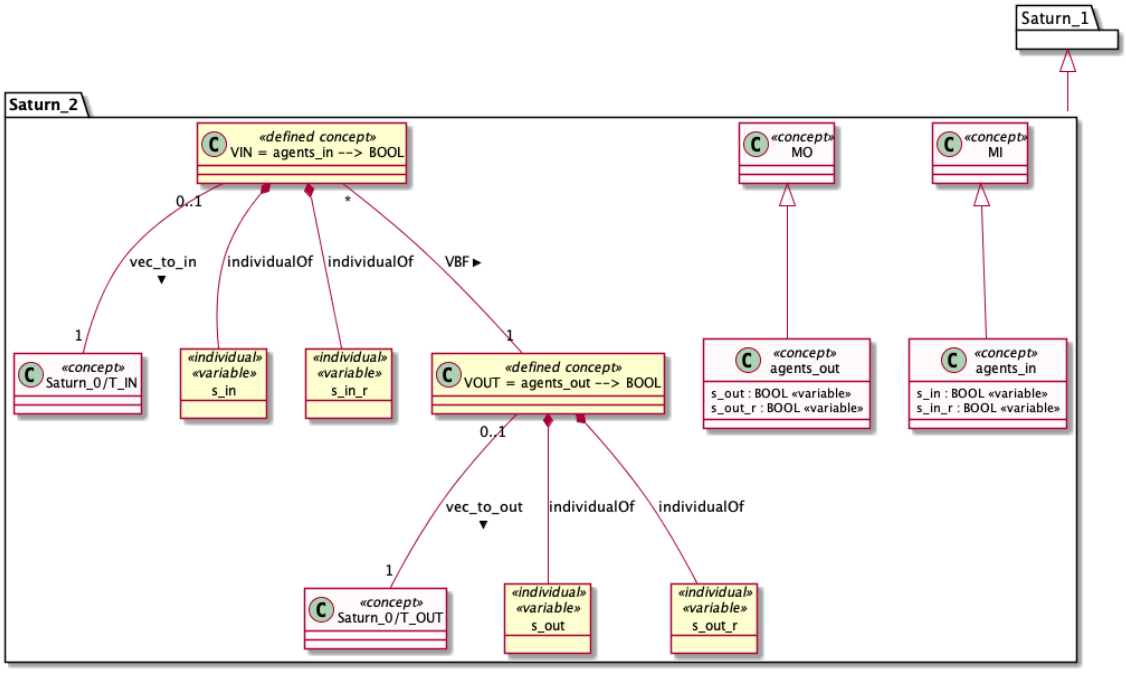


Figure 5.5 – ***Saturn_2***: ontology associated with the second refinement level

In domain model `Saturn_0` (Fig. 5.2 (b)), the type of input data is modeled as a constant concept `T_IN` (instance of class Concept of Fig. 5.3) defining a variable individual `in` (instance of class Individual of Fig. 5.3) which represents the input data. Similarly, the type of output data is modeled as a constant concept `T_OUT` defining a variable individual `out`. Finally, the computation function `FB` is modeled as a functional association (instance of class Association of Fig. 5.3) from `T_IN` to `T_OUT`. In domain model `Saturn_1` (Fig. 5.4) which refines `Saturn_0`, a new variable individual named `in_r` is defined to represent the acquired input data and another one, `out_r`, is defined to represent the computed result.

In domain model `Saturn_2` (Fig. 5.5) which refines `Saturn_1`, 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`, `in_r`, `out` and `out_r` are refined respectively by individuals `s_in`, `s_in_r`, `s_out` 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: $in = vec\_to\_in(s\_in), in\_r = vec\_to\_in(s\_in\_r), out = vec\_to\_out(s\_out), out\_r = vec\_to\_out(s\_out\_r)$. 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).

5.3.4 The B System Specification

B System Specification Constructed from Domain Models

Updates performed on the SysML/KAOS domain modeling language have resulted in adjustments on translation rules defined in [65]. The adjusted rules are fully described in annex B. They have been formally verified with Event-B. The corresponding Event-B specification can be found in annex B. The rules are implemented within the *SysML/KAOS Domain Modeling tool* [56].

The specification below represents the *B System* specification obtained from the domain model of Fig. 5.2 (b).

SYSTEM Saturn_0
SETS b_T_IN b_T_OUT
CONSTANTS b_FB
PROPERTIES
axm1: $b\_FB \in b\_T\_IN \rightarrow b\_T\_OUT$
VARIABLES b_in b_out

INVARIANT
inv1: $b\_in \in b\_T\_IN$
inv2: $b\_out \in b\_T\_OUT$
Event *INITIALISATION* $\widehat{=}$ **then**
act1: $b\_in :: b\_T\_IN$
act2: $b\_out :: b\_T\_OUT$

Concepts T_IN and T_OUT give abstract sets b_T_IN and b_T_OUT. Variable individuals in and out give *B System* variables b_in and b_out typed (inv1 and inv2) and initialised (act1 and act2) respectively as items of b_T_IN and b_T_OUT (rule 11 of Table B.2 of annex B). Finally, association FB gives a *B System* constant named b_FB, typed (axm1) as a total function between b_T_IN and b_T_OUT (rule 10 of Table B.2 of annex B).

The *B System* specification obtained from the domain model of Fig. 5.4 defines a refinement named Saturn_1 which refines Saturn_0 and introduces variables b_out_r and b_in_r with invariant $b\_out\_r \in b\_T\_OUT \wedge b\_in\_r \in b\_T\_IN$. The specification obtained from the domain model of Fig. 5.5 defines a *B System* refinement named Saturn_2 which refines Saturn_1. In Saturn_2, concepts MI and MO are defined as abstract sets. In addition, Saturn_2 defines agents_in, agents_out, VIN, VOUT, vec_to_in, vec_to_out and VBF as constants. Properties

axm1: $b\_agents\_in \subseteq b\_MI$
axm2: $b\_agents\_out \subseteq b\_MO$
define b_agents_in and b_agents_out as subsets of b_MI and b_MO (rule 5 of Table B.2 of annex B).
axm3: $b\_VIN = b\_agents\_in \rightarrow BOOL$
axm4: $b\_VOUT \in b\_agents\_out \rightarrow BOOL$
axm5: $b\_vec\_to\_in \in b\_VIN \rightarrowtail b\_T\_IN$
axm6: $b\_vec\_to\_out \in b\_VOUT \rightarrowtail b\_T\_OUT$
axm7: $b\_VBF \in b\_VIN \rightarrow b\_VOUT$
axm8: $b\_VBF = (b\_vec\_to\_in; b\_FB; b\_vec\_to\_out^{-1})$

Following rule 10 of Table B.2 of annex B, property axm5 defines b_vec_to_in as a total injection from b_VIN to b_T_IN. Constant b_vec_to_out is typed in a similar manner by property axm6. Finally, the total function b_VBF (axm7) is defined by property axm8 as the composition of functions b_vec_to_in, b_FB and $b\_vec\_to\_out^{-1}$ (property axm8 results from the translation of a logical formula defined in domain model ***Saturn_2***). The *B System* refinement Saturn_2 also defines variables such as b_s_in and b_s_out, along with their invariants and initialisations.

B System Specification Constructed from the Goal Model

The root level of the goal diagram of Fig. 5.1 gives the *B System* event *Process* specified as:

Event *Process* $\widehat{=}$ **then** $b\_out := b\_FB(b\_in)$ **END**

Furthermore, the first refinement level of the goal diagram gives the following *B System* specification:

Event *Get* ***ref_and*** *Process* $\widehat{=}$ **then** $b\_in\_r := b\_in$ **END**;
Event *Compute* ***ref_and*** *Process* $\widehat{=}$ **then** $b\_out\_r := b\_FB(b\_in\_r)$ **END**;
Event *Put* ***ref_and*** *Process* $\widehat{=}$ **then** $b\_out := b\_out\_r$ **END**

Each refinement level goal is translated into an event for which the body has been manually specified: event *Get* reads the input data, event *Compute* computes the output data and event *Put* outputs the result. The keyword ***ref_and*** is used to specify that concrete events *Get*, *Compute* and *Put* refine abstract event *Process*, in accordance with SysML/KAOS goal refinements, through the *AND* operator. This allows the automatic generation of proof obligations related to usage of the *AND* operator between abstract and concrete refinement levels [102] by the *Atelier B* tool:

(po1) $Get\_Guard \Rightarrow Process\_Guard$
(po2) $Compute\_Guard \Rightarrow Process\_Guard$
(po3) $Put\_Guard \Rightarrow Process\_Guard$

(po4) (*Get_Post* ∧ *Compute_Post* ∧ *Put_Post*) ⇒ *Process_Post*

For instance, the full specification of proof obligation (po4) is:
(*b_in_r* = *b_in* ∧ *b_out_r* = *b_FB*(*b_in_r*) ∧ *b_out* = *b_out_r*)
⇒ *b_out* = *b_FB*(*b_in*).

It expresses that when the input data is read and the output data is computed, the output data is the result of applying b_FB to the input data.

The keyword ***ref_or*** is used when the *OR* operator appears between an abstract and a concrete refinement levels.

5.4 Discussion

The specification of the Saturn protocol includes five refinement levels. It has been built, in a methodical and structured way, thanks to SysML/KAOS. Table 5.1 summarises the key characteristics related to proof obligations. Discharged using *Atelier B*, they allow the detection of omissions, ambiguities, redundancies and contradictions within requirements, while considering domain constraints. Furthermore, the domain modeling language has been extended in order to be more suitable for use in system modeling.

Table 5.1 – Key characteristics related to the formal specification

Refinement level	L0	L1	L2	L3	L4	*Summary*
Invariants	**2**	**6**	**8**	**1**	**1**	*18*
Events	**1**	**4**	**3**	**5**	**9**	*22*
Proof Obligations (PO)	**1**	**3**	**10**	**9**	**25**	*48*

5.5 Conclusion and Future Work

This paper focusses on assessment of the SysML/KAOS method through the formal modeling of requirements related to *Saturn*, a rail communication protocol proposed by *ClearSy* [129]. SysML/KAOS goal and domain modeling languages have been used to specify Saturn's requirements and application domain entities and properties. Translation rules, supported by tools [56, 102], have then been used to obtain a *B System* specification. The SysML/KAOS method has proven its usefulness for the specification of the protocol and has been enhanced, especially the domain modeling language, to make it more suitable for use in system modeling.

Work in progress is aimed at (i) integrating the updates within the open-source platform *Openflexo* [109], which federates the various contributions of *FORMOSE* project partners [17].

Part II

Gestion de la complexité au sein de SysML/KAOS

Chapitre 6

Formalisation des assignations d'exigences SysML/KAOS au travers des décompositions de composants B System

Résumé

L'utilisation des méthodes formelles pour la vérification et la validation des spécifications de systèmes critiques et complexes est importante, mais peut s'avérer extrêmement fastidieuse en l'absence de mécanismes de modularisation. *SysML/KAOS* est une méthode formelle d'ingénierie des exigences qui comprend un langage de modélisation des exigences à partir des besoins exprimés par les parties prenantes. Elle comprend également un langage de modélisation du domaine, pour la représentation des éléments caractéristiques du domaine d'application sous forme d'ontologies, et des règles permettant la mise en correspondance entre modèles SysML/KAOS et spécifications *B System*. Par ailleurs, pour gérer la complexité des systèmes, SysML/KAOS rend possible leur décomposition en sous-systèmes. En effet, le langage SysML/KAOS de modélisation des buts permet de représenter les assignations d'exigences aux sous-systèmes (ou agents) responsables de leur satisfaction. La contribution de ce chapitre est une approche permettant de garantir formellement que chaque exigence affectée à un sous-système sera correctement satisfaite par ce dernier, conformément à

la spécification du système de plus haut niveau. Cette approche repose sur les mécanismes de décomposition des composants *B System*. Un accent particulier est mis sur la préservation des invariants du système de plus haut niveau au sein des spécifications des sous-systèmes.

Commentaires

La contribution ici réside dans la définition d'une approche permettant de garantir formellement que chaque exigence SysML/KAOS affectée à un sous-système sera correctement satisfaite par ce dernier, dans la limite définie par la spécification formelle du système et des sous-systèmes.

La proposition, évaluée sur l'étude de cas *steam-boiler control specification problem* (problème de spécification du contrôleur d'une chaudière à vapeur) [22], a fait l'objet d'un article accepté et publié [59] dans le cadre de la 14^{e} édition de la conférence internationale *iFM (integrated Formal Methods)* qui s'est déroulée à Maynooth, Ireland en septembre 2018.

Cette contribution et l'article sus-cités ont été élaborés par mes soins en tenant compte des remarques et commentaires issus tant du Professeur *Michael Leuschel* de l'*Université de Düsseldorf* que de mon équipe d'encadrement.

Formalisation of SysML/KAOS Goal Assignments with *B System* Component Decompositions

Steve Jeffrey Tueno Fotso
Université de Sherbrooke, GRIL, Québec, Canada
Université Paris Est Créteil, LACL, Créteil, France
Steve.Jeffrey.Tueno.Fotso@USherbrooke.ca

Marc Frappier
Université de Sherbrooke, GRIL, Québec, Canada
Marc.Frappier@usherbrooke.ca

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

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

Michael Leuschel
University of Düsseldorf, Germany
leuschel@cs.uni-duesseldorf.de

Keywords: Requirements Engineering, Formal Models, Goal Modeling, Modularisation, *SysML/KAOS*, *B System*, *Event-B*, Steam Boiler

Abstract

The use of formal methods for verification and validation of critical and complex systems is important, but can be extremely tedious without modularisation mechanisms. *SysML/KAOS* is a requirements engineering method. It includes a goal modeling language to model requirements from stakeholder's needs. It also contains a domain modeling language for the representation of system application domain using ontologies. Translation rules have been defined to automatically map SysML/KAOS models into *B System* specifications. Moreover, since the systems we are interested in naturally break down into subsystems (enabling the distribution of work between several agents: hardware, software and human), *SysML/KAOS* goal models allow the capture of assignments of requirements to agents responsible of their achievement. Each agent is associated with a subsystem.

The contribution of this paper is an approach to ensure that a requirement assigned to a subsystem is well achieved by the subsystem. A particular emphasis is placed on ensuring that system invariants persist in subsystems specifications.

6.1 Introduction

The research work presented in this paper is part of the *FORMOSE* project [17] and focuses on the formal requirements modeling of systems in critical areas such as railway or aeronautics. System requirements are modeled using the SysML/KAOS goal modeling language [92]. Translation rules from goal models to *B System* specifications are defined in [102]. They allow the automatic generation of the skeleton of the formal specification of the system [102]. In addition, a language has been defined to express the domain model associated to the goal model, using ontologies [61, 64]. Its translation gives the structural part of the *B System* specification [65]. Finally, it remains to specify the body of events [1]. Once done, the *B System* specification can be verified, animated and validated using the whole range of tools that support the *B* method [10], largely and positively assessed on industrial projects for more than 25 years [93].

To ensure the distribution of work between several agents and a better maintenability, reusability and scalability of the system, SysML/KAOS allows its partitionning into subsystems: a goal diagram models the main system and further goal diagrams are built for subsystems. Actually, each subsystem is associated with an agent that is responsible for achieving its requirements. The contribution of this paper is an approach to ensure that a requirement assigned to a subsystem is well achieved. The approach uses formal decomposition mechanisms [11] to construct, from the formal specification of a high-level system, the interface of each of its subsystems. The interface of a subsystem describes the requirements that the high-level system expects from the subsystem. Proof obligations are defined to ensure that the invariants of each subsystem is consistent with that of the high-level system. The approach thus ensures that each subsystem achieves its expected goals with respect to constraints set by the high-level system. The proposed approach is illustrated on the *steam-boiler* control specification problem, proposed by *J. C. Bauer* in [22].

1. See [58, 133] for assessment case studies.

The remainder of this paper is structured as follows: Section 2 briefly describes the SysML/KAOS requirements engineering method and its goal and domain modeling languages, the *B System* formal method, and the translation of SysML/KAOS models. Section 3 presents existing techniques interested in the achievement of system requirements by subsystems, and existing formal decomposition approaches. Finally, Section 4 presents our approach illustrated on the steam-boiler control specification problem and Section 5 reports our conclusions and discusses future work.

6.2 Context

6.2.1 SysML/KAOS Goal Modeling

Presentation

SysML/KAOS [92,98] is a requirements engineering method based on *SysML* [78] and KAOS [138]. *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. *KAOS* is a requirements engineering method which allows the representation of requirements to be satisfied by a system and of expectations with regards to the environment through a hierarchy of 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 [17], 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*. In addition, *SysML/KAOS* includes a domain modeling language which combines the expressiveness of *OWL* [118] and the constraints of *PLIB* [112].

Figure 6.1 represents the metamodel associated with the modeling of SysML/KAOS functional goals [101]. A goal model consists of goals in relation through operators of which the main ones are: ***AND*** and ***OR***. An ***AND*** *operator* decomposes a goal into subgoals, and all of them must be achieved to realise the parent goal. An ***OR*** *operator* decomposes a goal into subgoals such that the achievement of only one

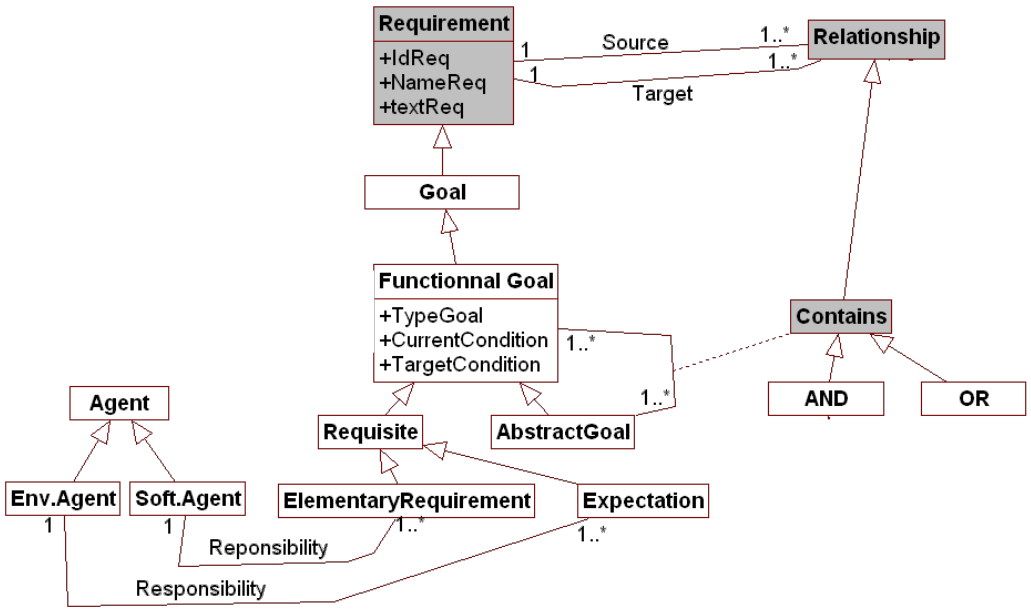


Figure 6.1 – The SysML/KAOS functional goal metamodel [101]

of them is sufficient for the accomplishment of the parent goal. An abstract goal is a functional goal which must be refined. A requisite is a functional goal sufficiently refined to be assigned to an operational agent. Environment agents are responsible of expectations and software agents are responsible of requirements.

Illustration

The challenge of the steam-boiler control specification problem [22] is to specify a system controlling the level of water in a steam-boiler. The system deals with a steam-boiler (SB), a water unit to measure the quantity of water in SB, a pump to provide SB with water, a pump controller and a steam unit to measure the quantity of steam flowing out of SB. In order to be concise, we limit the system operating modes to the three main ones: *normal*, *degraded* and *rescue*. We also consider two different minimum and maximum water quantities: (*Min1* and *Max1*) for the *normal* mode and (*Min2* and *Max2*), satisfactory levels for the abnormal modes (*degraded* and *rescue*). Figure 6.3 is a state diagram representing the steam boiler controller operating modes:

- In the *normal* mode, the controller tries to maintain the quantity of water within *Min1* and *Max1*, with all the units behaving correctly. When a failure occurs on the water unit, the mode is set to *rescue*. In case of any other failure, the mode is set to *degraded*.
- In the *degraded* mode, the controller tries to maintain the quantity of water within *Min2* and *Max2*, despite a possible failure other than a failure of the water unit. If a failure occurs on the water unit, the mode is set to *rescue*. When all failures are repaired, the mode is set to *normal*.
- In the *rescue* mode, the controller tries to maintain the quantity of water within *Min2* and *Max2*, despite a possible failure of the water unit. It estimates the water quantity, using the measurement of the pump controller and that of the steam unit. When all failures are repaired, the mode is set to *normal*. If the water unit is repaired and there is another failure, the mode is set to *degraded*.

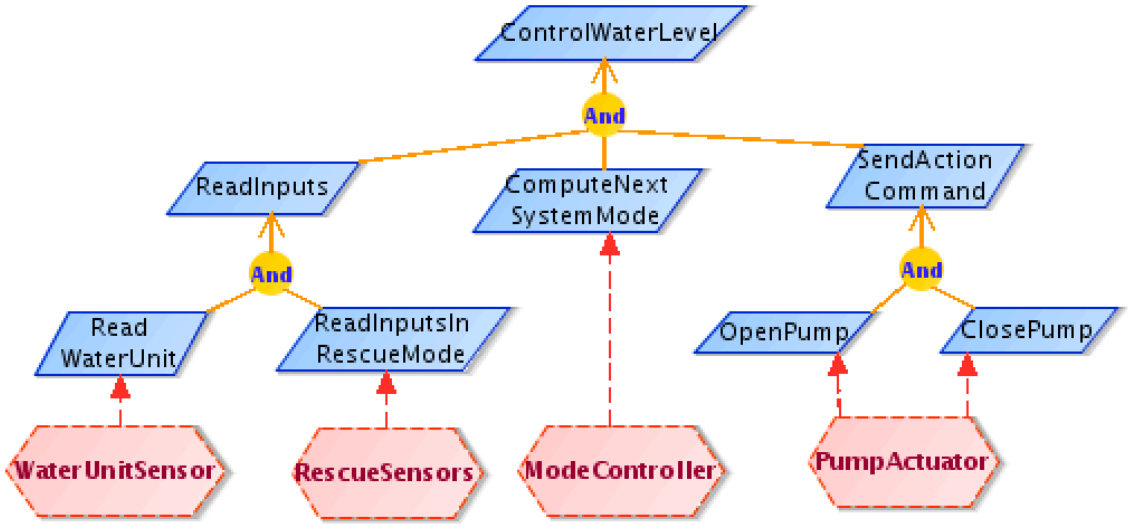


Figure 6.2 – Excerpt from the steam-boiler control system goal diagram

Figure 6.2 is an excerpt from the SysML/KAOS goal diagram representing the functional goals of the steam-boiler control system. The main purpose of the system is to control the level of water in the boiler (abstract goal **ControlWaterLevel**). To achieve it, the system must read inputs from the sensors (abstract goal **ReadInputs**), compute the next operating mode using available data (requisite **ComputeNextSystemMode**) and send an action command to the pump (abstract

goal **SendActionCommand**). The action may be the opening (requisite **OpenPump**) or the closing (requisite **ClosePump**) of the water pump. To ensure the achievement of goal **ReadInputs**, the system must be able to obtain *water unit* measurements, in case the *water unit* is behaving correctly (requisite **ReadWaterUnit**). However, since the *water unit* may become defective, the system must also be able to obtain measurements from the *steam unit* and *pump controller*, in order to estimate the quantity of water in the boiler (requisite **ReadInputsInRescueMode**). Four agents are defined for the achievement of requisites: `WaterUnitSensor` responsible of **ReadWaterUnit**, `RescueSensors` responsible of **ReadInputsInRescueMode**, `ModeController` responsible of **ComputeNextSystemMode** and `PumpActuator` responsible of **OpenPump** and **ClosePump**.

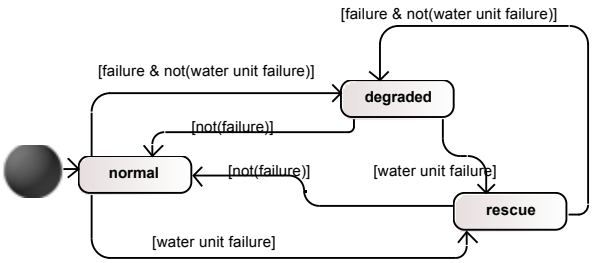


Figure 6.3 – State diagram of the steam boiler controller operating modes

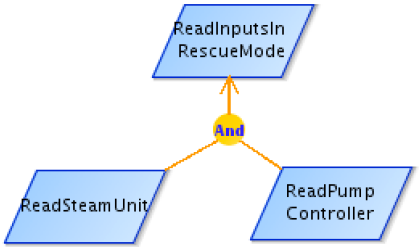


Figure 6.4 – Excerpt from the goal diagram of the subsystem associated with agent **RescueSensors**

Figure 6.4 is an excerpt from the SysML/KAOS goal diagram representing the functional goals of the subsystem associated with agent **RescueSensors**. Its main purpose is an abstract goal **ReadInputsInRescueMode** representing the requisite **ReadInputsInRescueMode** of the main system goal diagram (Fig. 6.2). To achieve it, the system must read values from the steam unit (**ReadSteamUnit**) and pump controller (**ReadPumpController**), in order to estimate the quantity of water in the boiler, in case of a failure of the water unit.

6.2.2 SysML/KAOS Domain Modeling

Presentation

The SysML/KAOS domain modeling language [61, 64] uses ontologies to represent domain models. It is based on *OWL* [118] and *PLIB* [112], two well-known ontology modeling languages. Each domain model corresponds to a refinement level in the SysML/KAOS goal model. The *parent* association represents the hierarchy of domain models. A domain model can define multiple elements. For our purposes, a domain model can define concepts and their individuals, relations, attributes, datasets and predicates [61, 64]. A concept represents a collection of individuals with common properties. It can be declared variable (*isVariable* = *TRUE*) when the set of its individuals can be dynamically updated by adding or deleting individuals. Otherwise, it is constant (*isVariable* = *FALSE*). A data set represents a collection of data values. A relation captures links between concepts, and an attribute, links between concepts and data sets. They can be variable or constant. Cardinalities are defined to represent restrictions on relations. A predicate expresses constraints between domain model elements, using the first order logic.

Illustration

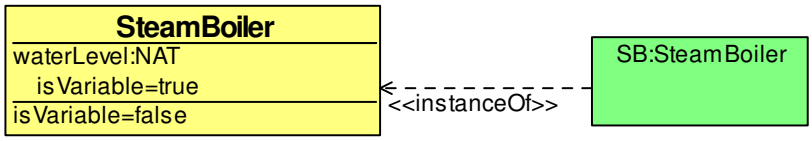


Figure 6.5 – **steam_boiler_controller_domain_model**: ontology associated with the root level of the goal diagram of Fig. 6.2

Figure 6.5 represents the SysML/KAOS domain model associated with the root level of the goal diagram of Fig. 6.2. The steam-boiler entity is modeled as a concept named `SteamBoiler`. As in the case study, adding or deleting a steam-boiler is not considered, property isVariable of `SteamBoiler` is set to `false`. Concept `SteamBoiler` has one individual named `SB`, representing the steam-boiler under the supervision of the system. The attribute `waterLevel` defined in `SteamBoiler` represents the water level in the boiler. It is variable, since it is possible to dynamically change the level of water in the boiler. Refinements of domain model **steam_boiler_controller_domain_model** define the operating mode of the controller using a variable attribute named `operatingMode`, having `SteamBoiler` as domain, and as range, an instance of EnumeratedDataSet containing three data values

(`normal`, `degraded` and `rescue`), representing the possible operating modes. For individual `SB`, `operatingMode` is initialised to `normal`, since we consider that the system starts in the normal mode. The associations between a steam-boiler and its sensors and actuators are modeled as relations: a relation named `SteamBoilerSensors` which links the steam-boiler to its sensors and a relation named `SteamBoilerActuators` which links the steam-boiler to its actuators. We have defined three sensors (a steam unit named `SU`, a pump controller named `PC` and a water unit named `WU`) and one actuator (a pump named `P`).

The specification below expresses, using predicates, some domain constraints that have been captured. It should be noted that the variables represent internal states of the controller [111].

```
p2.1: sensorState(WU)= "defective" => operatingMode(SB) ="rescue"
p2.2: (sensorState(WU)="nondefective" & sensorState(SU)= "defective")=> operatingMode(SB) ="degraded"
p2.3: (sensorState(WU)="nondefective" & sensorState(PC)= "defective")=> operatingMode(SB) ="degraded"
p2.4: (sensorState(WU)="nondefective" & actuatorState(P)= "defective")=> operatingMode(SB) ="degraded"
```

Predicate `p2.1` asserts that the operating mode must be *rescue* if the water unit is known to be *defective* and predicates `p2.2` .. `p2.4` assert that the operating mode must be *degraded* if a device is known to be *defective*, except for the water unit.

6.2.3 B System

Event-B is an industrial-strength formal method for *system modeling* [8]. It is used to incrementally construct a system specification, using refinement, and to prove properties. An *Event-B* model includes a static part called context and a dynamic part called machine. Constants, abstract and enumerated sets, and their properties, constitute the static part. The dynamic part includes the representation of the system state using variables constrained through invariants and updated through events. 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 the update of state variables. A machine can refine another machine, a context can extend other contexts and a machine can see contexts. 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 [8]. *B System* is an *Event-B* syntactic variant proposed by *ClearSy*, an industrial partner in the *FORMOSE* project [17], and supported by *Atelier B* [41]. A *B System* specification consists of components. Each component can be either a system or a refinement and it may define static or

dynamic elements. Although it is advisable to always isolate the static and dynamic parts of the *B System* formal model, it is possible to define the two parts within the same component. In the following sections, our *B System* models will be presented using this facility.

6.2.4 Translation of SysML/KAOS Models

Presentation

The formalisation of SysML/KAOS goal models is detailed in [102]. The proposed rules allow the generation of a formal model whose structure reflects the hierarchy of the SysML/KAOS goal diagram: one component is associated with each level of the goal hierarchy; this component defines one event for each goal. As the semantics of the refinement between goals is different from that of the refinement between *B System* components, new proof obligations for goal refinement are defined in [102]. They complete the classic proof obligations for invariant preservation and for event feasibility. Nevertheless, the generated *B System* specification does not contain the system structure, that are variables, constrained by an invariant, and constants, constrained by properties. This structure is provided by the translation of SysML/KAOS domain models. The corresponding rules are fully described in annex A and their formal verification is described in [65]. In short, domain models identify formal components. A concept without a parent gives a *B System* abstract set. Each concept *C*, with parent *PC*, gives a formal constant, subset of the correspondent of *PC*. Relations and attributes give formal relations. The rules also allow the extraction of the initialisation of state variables.

Illustration

Each refinement level, of the *B System* specification of the steam-boiler control system, is the result of the translation of goal and domain models, except the body of events that are provided manually. The full specification, verified using the *Rodin* tool [35], can be found in [133]. Its consistency is ensured with the discharge of *60* proof obligations, *20 %* manually and the rest automatically. Interactive proofs were mostly required because of enumerated set definitions that involve partitions: several proof rules require partition rewrites. The generated specification includes three refinement levels.

SYSTEM steam_boiler_controller
SETS SteamBoiler
CONSTANTS SB
PROPERTIES
axm: $SB \in SteamBoiler$
VARIABLES waterLevel
INVARIANT
inv: $waterLevel \in SteamBoiler \rightarrow \mathbb{N}$

Event INITIALISATION $\widehat{=}$ **then**
act: $waterLevel : \in \{SB\} \rightarrow \mathbb{N}$
END
Event ControlWaterLevel $\widehat{=}$ **any** wlvl
where
grd: $wlvl \in \mathbb{N}$
then
act: $waterLevel(SB) := wlvl$
END

Figure 6.6 – Root level of the *B System* specification of the steam-boiler control system

Figure 6.6 represents the root level of the *B System* specification of the steam-boiler control system. Concept `SteamBoiler` gives a set and its individual `SB` gives a constant typed with `axm` as an element of set *SteamBoiler*. Attribute `waterLevel` gives a total function from *SteamBoiler* to $\mathbb{N}$ initialised with the action `act` of event `INITIALISATION`. At this level, event `ControlWaterLevel` takes a parameter $wlv \in \mathbb{N}$ and defines it as the level of water in `SB`.

The first refinement level defines a component containing 6 variables, 7 invariants and 4 events (including the *INITIALISATION* event). The second refinement level (`steam_boiler_controller3`) defines a component containing the same set of variables (*waterLevel*, *operatingMode*, *sensorState*, *sensorInput*, *actuatorState*, and *actuatorOutput*), 4 invariants (`p2.1`..`p2.4`) and 6 events: *INITIALISATION*, *ReadWaterUnit*, *ReadInputsInRescueMode*, *ComputeNextSystemMode*, *OpenPump* and *ClosePump*.

The translation rules make it possible to obtain a *B System* specification which becomes complete after the definition of the body of events. The main system is associated with a *B System* model, and each subsystem is associated with another one. However, there are no mechanisms to ensure that subsystem goals are consistent with goals assigned to the high-level system. In the rest of this paper, we are interested in providing these mechanisms.

6.3 Existing Work

Section 6.3.1 presents relevant work related to the assignment of system goals to subsystems, with regard to mechanisms to ensure that subsystem goals are consistent with goals assigned to the high-level system; and Section 6.3.2 presents relevant formal model decomposition approaches.

6.3.1 Related Work on Goal Assignments

In [39, 126, 128], approaches are proposed to model a system made of several subsystems. Each subsystem has its own goals (local goals) that are under the responsibility of an agent (local agent). Each local agent has a degree of freedom in taking local actions to satisfy its local goals. Furthermore, it can negotiate with other agents in attempting to satisfy their local goals. However, to ensure the consistency between subsystem goals and system requirements, a specific subsystem is introduced, under the supervision of a global agent. The global agent focus on the satisfaction of global goals [39]: it can suspend, reschedule or require the execution of an action by a local agent in order to ensure a satisfactorily achievement of system requirements. Although local agents are unaware of objectives of the overall system, they act, under the supervision of the global agent, to ensure the achievement of these overall goals. This approach guarantees a certain degree of freedom in updating the overall goals. However, it requires to implement replanning primitives within local agents and replanning strategies within the global agent.

In [14, 74, 137], strategies are presented, for a system made of subsystems under the responsibility of agents, to ensure that system requirements are achieved. Each agent evaluates its state and behaviours of other agents, and takes actions that enforce the achievement of system requirements. The decision tree that drives the evaluations made by agents can be internally encoded in each agent, or externally via shared data structures. Algebras are proposed for the representation of desired states. However, relevant changes in the internal structure of an agent require a complete review of established strategies. Furthermore, either the agents must have access to the full behavioral history of other agents, or the strategies must include what agents currently know and what they learn from their actions.

In [45], Wayne only considers subsystems. Each subsystem has its own internal goals and can assign goals to other subsystems. A subsystem can accept or refuse to achieve a goal. Whenever a goal is accepted by a subsystem, the subsystem is responsible to provide feedbacks related to its achievement. The main system can be viewed as a subsystem which does not accept goals while a process can be viewed as a subsystem which does not assign goals. As in [39, 128], feedbacks allow one subsystem to monitor the achievement of the goal assigned to another subsystem and to ensure that it is satisfactorily achieved. However, this approach does not take into account the constraints common to goals assigned to different subsystems.

In our approach, formal subcomponents, called interfaces, are extracted from the specification of the high-level system to constrain the specification of subsystem goals. Interface definitions are automatically extracted using a formal model decomposition technique.

6.3.2 Formal Model Decomposition

Definition

Model decomposition here consists in obtaining, from an initial model, a certain number of less complex models, which can be refined independently and such that the recomposition of subsequent refinement levels produces a model which conforms to the definition of the initial model [11]. We focuss on distribution of elements of the dynamic part of the high-level system formal specification because the fundamental difference, between two SysML/KAOS agents of the same goal diagram, lies in their behaviors. Recall that system behaviour is formally represented with a set of events and by all the variables that can be updated by these events along with their invariants. The decomposition with respect to the *INITIALISATION* event is trivial and will not be considered. Indeed, whatever the chosen decomposition strategy, a variable xx assigned to a subcomponent will be initialised within the subcomponent with the same action of the parent component, that initialises xx, since initialisation actions are independent. Similarly, if all events involve only disjoint sets of variables and each invariant involves only the variables appearing in events corresponding to goals of the same agent, the decomposition is trivial: each agent may be assigned a subcomponent defining the events corresponding to the goals assigned to the agent, as well as the associated variables and invariants. The difficulty lies in taking into account variables appearing in events corresponding to goals assigned to different agents (shared variables) and invariants involving variables that are assigned to different subcomponents (shared invariants).

Existing Approaches for the Decomposition of Formal Models

Abrial *et al.* [11] are interested in mechanisms allowing the decomposition of *Event-B* models, and specifically of *Event-B* machines. Indeed, at some point of the refinement process, an *Event-B* machine may have so many events and so many state variables that a further refinement may become difficult or even impossible to manage. Abrial *et al.* consider the decomposition as the distribution of the events of the machine to be split, between several sub-machines. An approach is proposed to handle the variables shared between several events, using external variables and events. Events assigned to a sub-machine are its *internal events*. A variable that is only involved in internal events of a sub-machine is an *internal variable* of the sub-machine. If a variable is involved in internal events of different sub-machines, then it is defined in each of them as an *external variable*. In a sub-machine, an *external variable* can be seen as the input and output channel, allowing the sub-machine to synchronise its activities with other sub-machines defining the same variable. An *external variable* cannot be data-refined. In a sub-machine A, an *external event*

is an event introduced to simulate the way an external variable is handled, in another sub-machine B, by an *internal event* of B. External events simulate how external variables are handled in other sub-machines. They do so by abstracting the behaviour of events of the initial machine that involve the external variables. They cannot be refined. Iliasov *et al.* describe in [81] another method for decomposition in *Event-B*. The approach is a special case of the one proposed by Abrial *et al.* restricted to sequential systems for which functionalities can be distributed among several modules.

Iliasov *et al.* describe in [81] another method for decomposition in *Event-B*. The main goal of the approach is to enable parallel development of several independent parts of a system as well as formal reuse of developed modules in other developments. The proposed approach consists in completing the definition of the Event-B method with notions of *operations*, *interfaces* and *modules*. A module is a collection of operations that can be invoked by other operations. A module defines an interface that lists its operations as well as their pre and post conditions. Proof obligations are provided to ensure that the definition of the operations of a module respects the specification of its interface. Iliasov *et al.* present their approach as a special case of the approach proposed by Abrial *et al.*. Indeed, their approach targets sequential systems, even though their functionality is distributed among several modules, while the approach proposed by Abrial *et al.* targets distributed systems.

A decomposition approach, using shared events, is proposed in [33]. It enables the variables of the initial machine to be distributed between sub-machines. When the variables of a global event are distributed between separate sub-machines, each sub-machine defines an event which is a partial version of the global event, and which simulates the action of the global event on the considered variables. The partial version of an event, defined within a sub-machine, consists in a copy of the original event, restricted to the considered variables (variables of the global event that are allocated to the sub-machine): only parameters, guards and actions referring to the specified variables are preserved from the global event [123].

Silva *et al.* in [123] have identified two methods for decomposition in *Event-B*: the first one considering shared variables and the second one considering shared events. The shared variable decomposition is the decomposition approach introduced in [11] and the shared event decomposition is the one introduced in [33]. A tool is proposed to support the decomposition approaches. For Butler *et al.* [34], the shared event approach is suitable for developing message-passing distributed systems while the shared variable approach is suitable for designing parallel computing programs. Furthermore, it is easier to implement the shared variable approach compared to the shared event approach. Indeed, regarding the shared variable approach, once the events are assigned, the distribution of variables can be done automatically. The decomposition approach is implemented as a plug-in for the *Rodin* platform.

The real difficulty lies in the determination of the refinement level from which to introduce the decomposition. Regarding the shared event approach, it may be difficult, once the distribution of variables has been done, to separate the guards and actions of events in order to construct the partial events (a variable cannot appear in two different sub-machines). Regarding invariants, actually, [122, 123] let the user select which invariant predicate should be assigned to which subcomponent.

In [121], an approach is proposed for the construction of the specification of an *Event-B* machine from the combination of specifications of several other machines (basic machines). It assumes the partitioning of variables of basic machines, however events can be shared. The machine thus constructed is a composition of basic machines. Proof obligations are proposed in order to verify the composition of machines. The invariant of the composition of machines M_1 to M_n with variables x_1 to x_n respectively is defined as the conjunction of the individual invariants and the composition invariant $I_{CM}(x_1, ..., x_n)$: $I(M_1||...||M_n) \widehat{=} I_1(x_1) \wedge ... \wedge I_n(x_n) \wedge I_{CM}(x_1, ..., x_n)$. We reuse this definition for the determination of proof obligations associated with the verification of the decomposition of the system specification (Sect. 6.4.1): the system is seen as a composition of its subsystems.

6.4 Mechanisms to Ensure the Consistency between Subsystems and System Requirements

With translation rules, each SysML/KAOS model, whether for the main system or for a subsystem, gives a *B System* specification. To ensure that subsystem goals conform to system requirements, we propose the definition of *B System* components called *interfaces* that will bridge the gap between system and subsystem specifications. An interface of a subsystem defines events that correspond to goals that the system assigns to the subsystem. It also defines variables involved in these events as well as their constraints. The most abstract level of the formal specification of a subsystem is defined as a refinement of the subsystem interface; this ensures that the subsystem specification conforms to the interface specification. We propose the use of a formal decomposition strategy, applied at the most concrete level of the *B System* specification of the high-level system (parent component), to build subsystem interfaces.

Figure 6.7 represents an illustration of our approach for a main system S and two subsystems $S1$ and $S2$. The specification of S defines three components: ***M*** which corresponds to the root level and ***M_ref1*** and ***M_ref2*** which correspond to the first and second refinement levels. The component ***M_ref2*** defines variables *x1*, *x2* and *x3*, invariant *I(x1, x2, x3)* and events *E1(x1, x3)* and *E2(x2, x3)*. Variable *x3* is

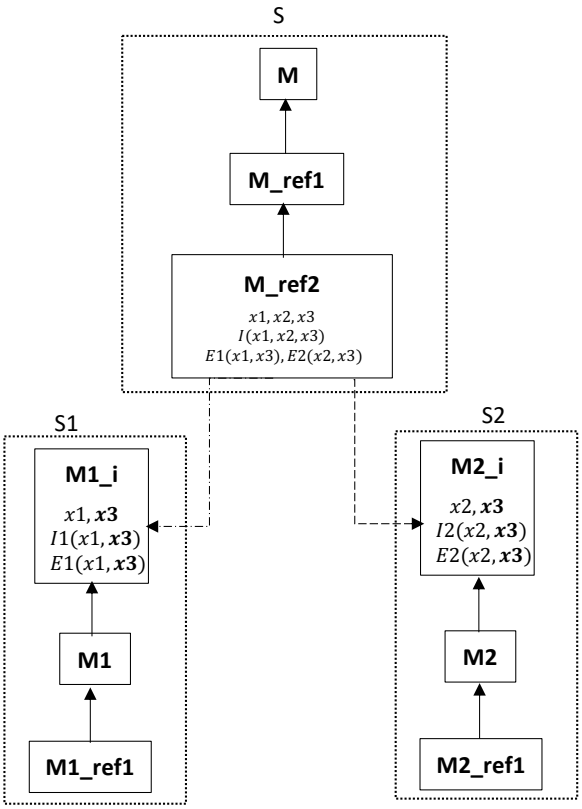


Figure 6.7 – Illustration of our approach

shared between the two events. We omitted the corresponding SysML/KAOS goal diagrams; however, the responsibility of *E1* is assigned to *S1* and the responsibility of *E2* is assigned to *S2*. Thus, the decomposition strategy is used to define interfaces ***M1_i*** for *S1* and ***M2_i*** for *S2*. The component representing the most abstract level of the specification of each subsystem (***M1*** for *S1* and ***M2*** for *S2*) is then defined as a refinement of the corresponding interface.

6.4.1 Construction of Interfaces

Interfaces, Variables and Events

In the SysML/KAOS methodology, goals are assigned to agents. A decomposition of the parent component, based on these assignments, may therefore use the shared variable decomposition approach: each agent gives a formal subcomponent, representing the subsystem interface, and for which the internal events are the correspondences of goals assigned to the agent. For an interface M_i corresponding to agent a_i, internal events of M_i are correspondences of goals assigned to a_i. The variables of M_i are the ones involved in internal events of M_i. If a variable of M_i appears in another interface, then it is an external variable; otherwise, it is an internal variable. Finally, external events are defined in M_i, to emulate how external variables of M_i are handled in other interfaces. Each external event of M_i is an abstraction of an internal event defined in another interface.

Regarding the illustration of Fig. 6.7, each interface contains the event assigned to the corresponding subsystem (we omitted external events for a sake of clarity). For instance, event *E1(x1, x3)* appears in ***M1_i***. Variables *x1* and *x3* also appears in ***M1_i*** because they are involved in *E1(x1, x3)*. Variable *x3* is defined as an external variable in ***M1_i*** and ***M2_i***.

Invariants

It remains necessary to decompose the invariants involving variables assigned to different interfaces. Let a component M, containing the variables x_1 and x_2 and the invariant $I(x_1, x_2)$, that is decomposed into subcomponents M_1 containing x_1 without x_2, and M_2 containing x_2 without x_1. Based on the composed invariant defined in [121] (see Sect. 6.3.2), we advocate that the following conditions are necessary and sufficient, regarding shared invariants (we disregard here properties defined in contexts), in addition to classical requirements of the *Event-B* method [8], to verify the decomposition of M into M_1 and M_2:

- **Subcomponent invariants do not contradict $\mathbf{I(x_1, x_2)}$:** If $I_1(x_1)$ is the invariant introduced in M_1 and $I_2(x_2)$ is the invariant introduced in M_2, then we must prove that: $\exists(x_1, x_2).(I(x_1, x_2) \wedge I_1(x_1) \wedge I_2(x_2))$. Thus, if we consider that x_1 is initialised to $x0_1$ in M_1 and that x_2 is initialised to $x0_2$ in M_2 (classical *Event-B* proof obligations ensure that predicate $I_1(x0_1) \wedge I_2(x0_2)$ evaluates to *TRUE*), we must prove that $I(x0_1, x0_2)$ evaluates to *TRUE*. This, in addition, ensures that initialisations in subcomponents preserve the composed invariant.

- Any subsystem event, that can update the value of a variable x introduced in the high-level system, must access x within a mutual exclusion context whenever it is triggered, so that no other event accessing the value of x can be triggered until its termination. Otherwise, it will not be possible to guarantee the accuracy of the value of a variable when an event is triggered within a component. Indeed, within the same *Event-B* component, events are triggered sequentially (to avoid possible inaccuracies in the state of the system), while subcomponents may have parallel behaviors. The constraint thus ensures the preservation of the sequentiality in the triggering of events coming from the high-level component, with regard to shared variables (the constraint is not necessary for events involving internal variables).
- **Subcomponent events simultaneously preserve global and local invariants:** If an event E_1, introduced in M_1, updates x_1, then we must prove that: $(I(x_1, x_2) \wedge I_1(x_1) \wedge I_2(x_2) \wedge E_1\_Guard(x_1) \wedge BA_{E_1}(x_1, x_1')) \Rightarrow (I(x_1', x_2) \wedge I_1(x_1') \wedge I_2(x_2))$. $E_1\_Guard(x_1)$ is a predicate denoting that the guard of E_1 is true for the current value of the state variable x_1. BA_{E_1} is the before-after predicate corresponding to E_1 [2]. For an event E_2, introduced in M_2, which updates x_2, the proof obligation is: $(I(x_1, x_2) \wedge I_1(x_1) \wedge I_2(x_2) \wedge E_2\_Guard(x_2) \wedge BA_{E_2}(x_2, x_2')) \Rightarrow (I(x_1, x_2') \wedge I_1(x_1) \wedge I_2(x_2'))$.

Thus, shared invariants (see Sect. 6.3.2) can remain in the parent component. It is just necessary to maintain the link between the parent component and the interfaces, through the introduction of a new clause within each interface allowing the referencing of the parent component or through the definition of an external record, and to include the above mentioned proof obligations. The most abstract level of the formal specification of a subsystem is then defined as a refinement of the subsystem interface. It is even possible to add variables, invariants or events in an interface to further constrain the specification of the subsystem or to assign specific goals.

It is also possible to define each variable of an interface as a constant within the others interfaces, where the variable do not appear, and to define shared invariants in each interface. However, this approach carries several difficulties: the update of a shared invariant will have to be done not only within the system specification but also within the specification of each subsystem; and it will be difficult to animate/model-check the formal model, since some variables will be seen as constants. In addition, it will be difficult to ensure that subsystem invariants are always simultaneously preserved, when considering shared variables.

2. The before-after predicate of E_1 denotes the relationship holding between the state variable of machine M_1 just before (denoted by x_1) and after (denoted by x_1') the triggering of E_1 [8]

Regarding the illustration of Fig. 6.7, each interface contains the definition of an invariant. Invariant *I(x1, x2, x3)* remains in ***M_ref2*** and the generated proof obligations are: (1) The invariants defined in ***M1_i*** and ***M2_i*** do not contradict the one defined in ***M_ref2***: $\exists(x1, x2, x3).(I(x1, x2, x3) \wedge I1(x1, x3) \wedge I2(x2, x3))$ (to be satisfied by the initialisation of variables); (2) actions of events *E1(x1, x3)* and *E2(x2, x3)* simultaneously preserve invariants defined in ***M1_i***, ***M2_i*** and the global invariant defined in ***M_ref2***:
(2a) $(I(x1, x2, x3) \wedge I1(x1, x3) \wedge I2(x2, x3) \wedge E1\_Guard(x1, x3)$
$\wedge\ BA_{E1}(x1, x3, x1', x3')) \Rightarrow (I(x1', x2, x3') \wedge I1(x1', x3') \wedge I2(x2, x3'))$;
(2b) $(I(x1, x2, x3) \wedge I1(x1, x3) \wedge I2(x2, x3) \wedge E2\_Guard(x2, x3)$
$\wedge\ BA_{E2}(x2, x3, x2', x3')) \Rightarrow (I(x1, x2', x3') \wedge I1(x1, x3') \wedge I2(x2', x3'))$.

Example:

Let M be a component defining invariant $\{x1, x2, x3\} \subset \mathbb{N} \wedge x1 + x2 = x3$ and events $E1 \mathrel{\widehat{=}}$ *then* $x1 := x1 + 1 || x3 := x3 + 1$ and $E2 \mathrel{\widehat{=}}$ *then* $x2 := x2 + 1 || x3 := x3 + 1$. If we consider the decomposition of M into subcomponents *M1* and *M2* with *M1* defining *E1* and invariant $x1 > 100$ and *M2* defining *E2* and invariant $x2 > 100$, the proof obligations are:

(1) $\exists(x1, x2, x3).(\{x1, x2, x3\} \subset \mathbb{N} \wedge x1 + x2 = x3 \wedge x1 > 100 \wedge x2 > 100)$;
(2a) $(\{x1, x2, x3\} \subset \mathbb{N} \wedge x1 + x2 = x3 \wedge x1 > 100 \wedge x2 > 100 \wedge x1' = x1 + 1 \wedge x3' = x3 + 1) \Rightarrow (\{x1', x2, x3'\} \subset \mathbb{N} \wedge x1' + x2 = x3' \wedge x1' > 100 \wedge x2 > 100)$;
(2b) $(\{x1, x2, x3\} \subset \mathbb{N} \wedge x1 + x2 = x3 \wedge x1 > 100 \wedge x2 > 100 \wedge x2' = x2 + 1 \wedge x3' = x3 + 1) \Rightarrow (\{x1, x2', x3'\} \subset \mathbb{N} \wedge x1 + x2' = x3' \wedge x1 > 100 \wedge x2' > 100)$.

They are dischargeable and guarantee that each action of a subsystem preserves not only its invariants, but also invariants of other subsystems and especially the invariant of the high-level system (the subsystems share a variable). They extend the classic proof obligation of invariant preservation [8], which just ensures that each subsystem preserves its own invariants, in the case of several subsystems operating simultaneously to achieve high-level goals and sharing data.

External Events

External events are introduced in interfaces to simulate, in a subsystem, how its external variables are handled in other subsystems. They are proposed by Abrial *et al.* in [11], because no link is maintained between a high-level formal component and its subcomponents after a shared variable decomposition operation. By defining a link between subsystem interfaces and the most concrete component of the high-level system specification, as proposed in Sect. 6.4.1, it becomes redundant to define external events within interfaces. Through the link between an interface and the parent component, for an external variable x, it would suffice to evaluate the

events of the parent component involving x and which are not defined within the interface, to "observe" how x is handled in other subsystems. This approach avoids the difficulties lying in the definition of external events: (1) redundance of the same behavior, associated with an external variable, in each interface where the external variable appears; (2) partitioning of guards and actions of an event to consider only the variables of the interface where the external event must be defined.

6.4.2 Illustration on the Steam-Boiler Case Study

Table 6.1 – Repartition of variables between events and invariants in `steam_boiler_controller3`

Variables	Invariants	Events
waterLevel		*INITIALISATION, ReadWaterUnit, ReadInputsInRescueMode, ComputeNextSystemMode, OpenPump, ClosePump*
operatingMode	`p2.1..p2.4`	*INITIALISATION, ReadWaterUnit, ComputeNextSystemMode, OpenPump, ClosePump*
sensorState	`p2.1..p2.4`	*INITIALISATION, ReadWaterUnit, ReadInputsInRescueMode, ComputeNextSystemMode*
sensorInput		*INITIALISATION, ReadWaterUnit, ReadInputsInRescueMode*
actuatorState	`p2.4`	*INITIALISATION, ComputeNextSystemMode, OpenPump, ClosePump*
actuatorOutput		*INITIALISATION, OpenPump, ClosePump*

For the steam-boiler control system, the decomposition must be introduced in the second refinement level (`steam_boiler_controller3`), because it is the most concrete level of the *B System* specification of the main system. Table 6.1 presents the sharing of state variables between invariants and events of `steam_boiler_controller3`: variable *waterLevel* is shared between all agents, when considering events where it is involved; variable *sensorState* is shared between agents `WaterUnitSensor`, `RescueSensors` and `ModeController`; and variable *actuatorOutput* is owned by agent `PumpActuator`.

We have defined interfaces of the subsystems: each SysML/KAOS agent gives an interface.

Table 6.2 presents an overview of interfaces obtained from the decomposition of `steam_boiler_controller3`, along with their variables and events. For an interface I, elements in bold are those that are internal to I and the other elements are those that are external (shared). For instance, event ***ReadWaterUnit*** is an internal event in interface *WaterUnitSensor_i*, while event *ReadInputsInRescueMode* is an external event that simulates, in *WaterUnitSensor_i*, the behaviour of internal event ***ReadInputsInRescueMode***, defined in interface *RescueSensors_i*; variable ***actuatorOutput***

Table 6.2 – Overview of interfaces obtained from the decomposition of `steam_boiler_controller3`

Interfaces	Events	Variables
WaterUnitSensor_i	***ReadWaterUnit***, *ReadInputsInRescueMode*, *ComputeNextSystemMode*, *OpenPump*, *ClosePump*	*waterLevel*, *operatingMode*, *sensorState*, *sensorInput*
RescueSensors_i	***ReadInputsInRescueMode***, *ReadWaterUnit*, *ComputeNextSystemMode*, *OpenPump*, *ClosePump*	*waterLevel*, *sensorState*, *sensorInput*
ModeController_i	***ComputeNextSystemMode***, *ReadWaterUnit*, *ReadInputsInRescueMode*, *OpenPump*, *ClosePump*	*waterLevel*, *operatingMode*, *sensorState*, *actuatorState*
PumpActuator_i	***OpenPump***, ***ClosePump***, *ReadWaterUnit*, *ReadInputsInRescueMode*, *ComputeNextSystemMode*	*waterLevel*, *operatingMode*, *actuatorState*, ***actuatorOutput***

is an internal variable in interface *PumpActuator_i*, while variable *waterLevel* is an external variable. Variable *waterLevel* is defined as an external variable in all interfaces because it is involved in internal events of the four interfaces. In addition, since variable *waterLevel* is involved in all events, each interface defines an external event that simulates the behaviour of each event not internal to the interface. Once it will be possible to define a link between each interface and its parent component, we believe that it will no longer be necessary to define these external events. Invariants `p2.1`..`p2.4` remain in `steam_boiler_controller3`; however, if needed, invariants `p2.1`..`p2.3` can be defined in interfaces *WaterUnitSensor_i*, *ModeController_i* and *PumpActuator_i*, and invariant `p2.4` can be defined in *ModeController_i*.

Figure 6.8 is an overview of the root level of the *B System* specification of the subsystem associated to agent `RescueSensors`. It is a refinement of interface *RescueSensors_i*. We provide the specification of the event corresponding to goal **ReadSteamUnit** of the goal diagram of Fig. 6.4: when water unit `WU` is defective and steam unit `SU` and pump controller `PC` are non-defective (`grd1`), then a natural integer `val1` is set as the input obtained from sensor `SU` (`act2`). Controller variable *measures* is used to take into account the *non-simultaneity* and the *non scheduling* of the measurement of values of sensors `SU` and `PC`, introduced in the goal diagram with the use of the *AND* operator between the root and first refinement levels. Within event **ReadSteamUnit**, variable *measures* allows the controller to consider the following cases: (1) when the measurement of values of `SU` and `PC` has not yet been achieved ($SU \notin dom(measures) \wedge PC \notin dom(measures)$), the value of `SU` is measured (`grd4`) and saved into variables *sensorInput* (`act2`) and *measures* (`act3`); (2) when the value of `PC` has already been measured, the value of `SU` is measured and used, together with the value of `PC`, to estimate the water level (`grd4` and `act1`).

REFINEMENT RescueSensors **REFINES** RescueSensors_i
VARIABLES waterLevel sensorState sensorInput measures
INVARIANT
inv: $measures \in \{SU, PC\} \nrightarrow \mathbb{N}$
theorem t1: $ReadSteamUnit\_Guard \Rightarrow ReadInputsInRescueMode\_Guard$
theorem t2: $ReadPumpController\_Guard \Rightarrow ReadInputsInRescueMode\_Guard$
theorem t3: $ReadSteamUnit\_Post \wedge ReadPumpController\_Post \Rightarrow ReadInputsInRescueMode\_Post$
Event INITIALISATION $\widehat{=}$ **then**
act1: $waterLevel := \{SB \mapsto Min1\}$
act2: $sensorState := Sensor \times \{nondefective\}$
act3: $sensorInput :\in Sensor \rightarrow \mathbb{N}$
act4: $measures := \emptyset$
END
Event ReadSteamUnit $\widehat{=}$
any wlvl values val1 val2 **where**
grd1: $sensorState(WU) = defective \wedge sensorState[\{SU, PC\}] = \{nondefective\}$
grd2: $\{val1, val2\} \subseteq \mathbb{N}$ grd3: $SU \notin dom(measures)$ grd4: $values = \{SU \mapsto val1\}$
grd5: $wlvl \in \{TRUE \mapsto Min2 \,..\, Max2, FALSE \mapsto \{waterLevel(SB)\}\}(bool(PC \in dom(measures)))$
then
act1: $waterLevel(SB) := wlvl$ act2: $sensorInput := sensorInput \lhd\!\!\!- values$
act3: $measures := \{TRUE \mapsto \emptyset, FALSE \mapsto values\}(bool(PC \in dom(measures)))$
END

Figure 6.8 – Overview of the root level of the *B System* specification of the subsystem `RescueSensors`

Action `act3` allows, regarding the last case, to reset the content of variable *measures* for further measurements. The behavior of event **ReadPumpController** is identical to that of **ReadSteamUnit**, except that it performs the measurement of the value of PC.

Interface *RescueSensors_i* provides variables *waterLevel*, *sensorState* and *sensorInput* and event **ReadInputsInRescueMode** to component *RescueSensors*. Theorems `t1`..`t3` represent the SysML/KAOS proof obligations related to the use of the *AND* refinement operator [3] (Fig. 6.4) [102].

3. For an event `G`, `G_Guard` represents the guard of `G` and `G_Post` represents the post condition of its actions [102].

6.4.3 Discussion

The proposed approach uses the shared variable decomposition strategy and proof obligations to ensure that subsystems specifications conform to system requirements. The approach fits into the following process which is applicable for any system *S* made of subsystems *S1*..*Sn*, assuming that SysML/KAOS models of *S*, *S1*..*Sn* are already defined:

(1) Translate SysML/KAOS models of *S* into a *B System* specification made of components $C\_S_0, C\_S_1, ..., C\_S_p$, where C_S_r is a refinement of C_S_{r-1} (Sect. 6.2.4 and [63, 65, 102]).

(2) Complete the specification obtained from (1) by specifying the body of events (Sect. 6.2.4 and [58]).

(3) Use the formal decomposition strategy to construct, from C_S_p, the formal subcomponents *S1_i*..*Sn_i*, where *Sk_i* denotes the interface of subsystem *Sk*, containing the specification of goals that *S* assigns to *Sk* with their associated variables and constraints (Sect. 6.4.1).

(4) For each subsystem *Sk*:

(i) **IF** *Sk* is made of subsystems
THEN restart the whole process with *Sk* as the high-level system
ELSE apply steps (1) and (2) on *Sk*

(ii) Set component C_Sk_0 as a refinement of *Sk_i*.

The approach makes it possible to independently define, check and evolve the specifications of subsystems. It also allows centralised updates of constraints and goals assigned to subsystems: global update of the high-level system specification, which can be automatically propagated into interfaces, and/or local update of an interface, which is available for the whole subsystem specification.

6.5 Conclusion and Future Work

This paper focusses on an approach to ensure that a requirement assigned to a subsystem is well achieved by the subsystem. The approach uses a formal model decomposition strategy and proof obligations to guarantee that subsystem goals are consistent and meet system requirements expressed in SysML/KAOS models that are translated to *B System* specifications. The approach is appraised on the *steam-boiler* control specification problem [22], using *Rodin* [35], an industrial-strength tool supporting the *Event-B* method. Its advantages are discussed, with regard to some relevant related work.

Work in progress is aimed at studying the back propagation of updates on a *B System* specification within the associated SysML/KAOS model. We are also working on integrating the approach within the open-source platform *Openflexo* [109] which federates the various contributions of *FORMOSE* project partners [17].

Part III

Études de cas

Chapitre 7

Spécification formelle des exigences d'un protocole de transport ferroviaire : cas du protocole *hybrid ERTMS/ETCS level 3*

Résumé

Ce chapitre décrit une spécification des exigences du protocole de transport ferroviaire *hybrid ERTMS/ETCS level 3* dans le cadre de l'étude de cas proposée pour *ABZ2018*. Cette spécification est réalisée à partir de *SysML/-KAOS*. La spécification construite comprend sept niveaux de raffinement et sa cohérence a été vérifiée et validée à l'aide de la plateforme *Rodin*. Le langage SysML/KAOS de modélisation des buts est utilisé pour représenter les exigences du protocole et en extraire l'ossature d'une spécification *B System*. La partie structurelle de cette spécification *B System* est fournie par la traduction des modèles de domaine SysML/KAOS. La construction de la spécification est incrémentale, basée sur les mécanismes de raffinement existant au sein des méthodes SysML/KAOS et *B System*. La seule partie de la spécification qui doit être complétée manuellement est le corps des événements.

Commentaires

La contribution ici réside dans l'évaluation de la méthode SysML/KAOS sur une étude de cas d'envergure industrielle, en l'occurrence la spécification et la vérification formelle des exigences du protocole de transport ferroviaire *hybrid ERTMS/ETCS level 3* [79]. Cette évaluation permet (i) d'illustrer l'usage de SysML/KAOS, (ii) d'identifier ses forces et faiblesses au regard des méthodes de spécification existantes et (iii) d'éprouver l'adéquation entre le langage de modélisation des buts fonctionnels et le langage de modélisation du domaine défini dans le cadre de cette thèse.

L'évaluation décrite dans ce chapitre a fait l'objet d'un article accepté et publié [58] dans le cadre de la conférence ABZ2018. Sous invitation, une extension de l'article a fait l'objet d'une publication dans une édition du journal international *STTT (International Journal on Software Tools for Technology Transfer)* [135]. Le protocole *hybrid ERTMS/ETCS level 3* a également été spécifié, directement en Event-B, afin de mieux évaluer les avantages et limites inhérents à l'utilisation de la méthode SysML/KAOS. Cette approche classique de spécification a fait l'objet d'un article accepté et publié [96] dans le cadre de la conférence ABZ2018. Cet article a également fait l'objet d'une publication dans une édition du journal international *STTT*.

La spécification réalisée en utilisant la méthode SysML/KAOS et les articles afférents ont été élaborés par mes soins en tenant compte des remarques et commentaires issus de mon équipe d'encadrement. La spécification réalisée en utilisant l'approche classique a quant à elle été élaborée par la Professeure *Amel Mammar* de *Télécom SudParis* et les articles afférents ont été élaborés par le Professeur *Marc Frappier* de l'*Université de Sherbrooke*. Mon intervention dans ce dernier cas s'est limitée aux phases d'étude préliminaire (analyse et compréhension de la description de l'étude de cas, définition des abstractions de base) et de validation/comparaison.

Modeling the Hybrid ERTMS/ETCS Level 3 Standard Using a Formal Requirements Engineering Approach

Steve Jeffrey Tueno Fotso
Université Paris Est Créteil, LACL, Créteil, France
Université de Sherbrooke, GRIL, Québec, Canada
steve.tuenofotso@univ-paris-est.fr

Marc Frappier
Université de Sherbrooke, GRIL, Québec, Canada
Marc.Frappier@usherbrooke.ca

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

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

Keywords: Hybrid ERTMS/ETCS Level 3 Standard, Requirements Engineering, Formal Models, Domain Modeling, *SysML/KAOS*, *B System*, *Event-B*

Abstract

This paper presents a specification of the hybrid ERTMS/ETCS level 3 standard in the framework of the case study proposed for ABZ2018. The specification is based on methods and tools, developed in the *ANR FORMOSE* project, for the modeling and formal verification of critical and complex system requirements. The requirements are specified with *SysML/KAOS* goal diagrams and are automatically translated into *B System* specifications, in order to obtain the architecture of the formal specification. Domain properties are specified by ontologies with the SysML/KAOS domain modeling language, based on *OWL* and *PLIB*. Their automatic translation completes the structural part of the formal specification. The only part of the specification that must be manually completed is the body of events. The construction is incremental, based on refinement mechanisms that exist within the involved methods. Regarding the case study, the formal specification includes seven refinement levels and all proofs have been discharged under the Rodin platform.

7.1 Introduction

As highlighted by Lee *et al.* in [94], many designers leave system requirements implicit and start working on design solutions without a clear definition of the purpose of the system. Therefore, they measure their success by comparing their design with the implicit design goals that they have in mind, which may or may not meet stakeholder needs. As a consequence, they spend a great deal of time improving and iterating the design solution without, most often, reaching consensus with stakeholders. The approach proves to be expensive, inefficient and source of many failures [40], very often tragic in critical areas such as railway or aeronautics [94,104].

In this paper, we are interested in using the *SysML/KAOS* method, part of the *FORMOSE* project [17], on the case study proposed for ABZ2018 [79]. This case study deals with the specification of the *hybrid ERTMS/ETCS level 3* protocol (HEEL3) [54,107]. The case study is described in two main documents. The first one, [79], gives the general principles of HEEL3 and defines requirements to be considered. The second one, [54], offers a technical and detailed description of the protocol specification. It provides the safety requirements that the system must guarantee.

The SysML/KAOS method includes a requirements modeling language [92,98] to represent system requirements with goal diagrams. Domain entities and their related properties are represented with ontologies using a domain modeling language [61,64]. Once constructed, goal and domain models can be semi-automatically translated into a *B system* specification [41] following a set of translation rules [65,102], supported by tools [56,102]. The goal models give the set of *B System* components, each goal gives an event. As the refinement links defined between these components have to represent the SysML/KAOS refinements, which differs from *B System* refinement, new proof obligations are generated. The domain models, on the other hand, give the structural part of the *B System* specification. It consists of variables, constrained by an invariant, and constants, constrained by properties. Once completed with event bodies, the *B System* specification can be formally verified and validated to assess the requirements. This can be done using the full range of tools that support the *B* method [7], largely and positively assessed on industrial projects for more than 25 years [93].

Regarding the case study, the development team is composed of four members (the authors of this paper) which all have a good expertise in the formal specification of complex systems. Other members of the FORMOSE project have been involved in providing feedback on improvements related to the use of the SysML/KAOS method. The *Rodin* platform [35] has been used to support the verification and the validation of the *B System* specification, especially to prove the safety invariants and the refinement logic. The use of Rodin is made possible because Event-B and *B*

System share the same semantics. In addition, Rodin provides an intuitive interactive proof mechanism and allows not only to use *Atelier B* provers [115], but also other efficient proof tools such as *SMT* solvers [127]. The complete Rodin project can be found in [134]. Compared to direct specification approaches using only plain *Event-B* such as [97], SysML/KAOS provides a more structured and methodological process to the formal specification of the system. Furthermore, it allows a better reusability and readability of models, and a strong traceability between the formal specification and SysML/KAOS models, which capture system and domain textual descriptions.

In comparison to the paper on the same topic published at the ABZ2018 conference [58], this paper:

- is based on the revised version (version *1C*) of the technical document [54], which provides HEEL3's principles;
- uses the revised version of the SysML/KAOS domain modeling language [132] to represent domain entities and constraints. A graphical representation of domain models is provided;
- extends the discussion section with a comparison with the other case study specifications published in the ABZ2018 proceedings.

The remainder of this paper is structured as follows: Section 2 briefly describes the *B System* formal method, the SysML/KAOS goal and domain modeling languages and the rules for obtaining a *B System* specification from SysML/KAOS models. Follows a presentation, in Section 3, of the identified requirements and of the modeling strategy and, in Section 4, of some relevant details related to the specification. Section 5 reports a discussion on the use of SysML/KAOS and compares the work described in this paper with related studies. Finally, Section 6 reports our conclusion and outlines future work related to the SysML/KAOS method.

7.2 Background

7.2.1 Event-B and B System

Event-B [8] is an industrial-strength formal method for *system modeling*. It allows the incremental construction of system specifications and the proof of useful properties. Its main purpose is the modeling of closed systems: the modeling of the system is accompanied by that of its environment and of all interactions likely to occur between them.

B System is an *Event-B* syntactic variant proposed by *ClearSy*, an industrial partner in the *FORMOSE* project [17], and supported by *Atelier B* [41]. It shares the same semantics with *Event-B*. A *B System* specification consists of components. Each component can be either a system or a refinement and it may define static or dynamic elements. A refinement is a component which refines another one in order to concretise the system construction: addition of functionalities or specification of the achievement of some purposes. Constants, abstract and enumerated sets, and their properties, constitute the static part. The dynamic part includes the representation of system state using variables constrained through invariants and updated through events. 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 the update of state variables.

As with *Event-B*, proof obligations are defined to prove *invariant preservation* by events (invariant has to be true at any system state), *event feasibility* (existence of a state where event can be triggered), *convergence* (for events that need only be triggered a finite number of times) and *machine refinement* (the specification of a concrete machine conforms to that of the refined machine) [8]. This last proof obligation requires the guard and action of each concrete event to be stronger than that of the refined event (*guard strengthening* and *action simulation*), knowing that each concrete event either refines an abstract event or refines the *skip* event.

7.2.2 SysML/KAOS Goal Modeling

SysML/KAOS [92, 98] is a requirements engineering method which combines the traceability provided by *SysML* [78] with goal expressiveness provided by KAOS [138]. It allows the representation of requirements that must be satisfied by a system and of expectations with regards to the environment through a hierarchy of goals. The goal hierarchy is composed of 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.

7.2.3 SysML/KAOS Domain Modeling

Modeling the domain of a system consists in giving a representation of the set of entities that the system will be called upon to manipulate and the set of properties and constraints associated with them [19, 31, 46, 76]. A significant feature in our case, concerned with the specification of engineering systems, is that domain models

distinguish static entities from dynamic ones while not distinguishing tangible entities from intangible ones. It should be noted that a static entity is an entity whose state cannot be changed by system actions while a dynamic entity is the one whose state is dependent of system actions [86]. Furthermore, an intangible entity is an entity which cannot normally be touched or seen but can be objectively measured or conceived while a tangible entity is an entity which can normally be touched or seen or is an abstraction of such an entity [25]. Domain models in SysML/KAOS are represented as ontologies. These ontologies are expressed using the SysML/KAOS domain modeling language [61, 132], based on *OWL* [118] and *PLIB* [112], two well-known and complementary ontology modeling formalisms. Domain models are used to automatically generate the structural part of the *B System* formalisation of system requirements (sets, constants, properties, variables and invariants).

Each domain model corresponds to a refinement level in the SysML/KAOS goal model. They can be linked together to form a hierarchy. A domain model can define multiple elements. *Concepts* designate collections of *individuals* with common properties. A concept can be declared *variable* when the set of its individuals can be updated by adding or deleting individuals. Otherwise, it is considered to be *constant*. In addition, a concept can be an enumeration if all its individuals are defined within the domain model. It should be noted that an individual can be *variable* if it is introduced to represent a system state variable: it can represent different individuals at different system states. Otherwise, it is *constant*. *Associations* are concepts used to capture links between concepts. *Maplet individuals* capture associations between individuals through associations. The variability of an association is related to the ability to add or remove maplets. *Logical formulas* are used to represent constraints between different elements of the domain model in the form of *Horn clauses*. They are specified using the *B* syntax. *Gluing invariants* are logical formulas used to represent links between data defined within a domain model and those appearing in more abstract domain models. They capture relationships between abstract and concrete data during refinement and are used to discharge proof obligations.

7.2.4 Translation of SysML/KAOS Models

The formalisation of SysML/KAOS goal models is detailed in [102]. The proposed rules allow the generation of a formal model whose structure reflects the hierarchy of the SysML/KAOS goal diagram: one component is associated with each level of the goal hierarchy; this component defines one event for each goal. As the semantics of the refinement between goals is different from that of the refinement between *B System* components, new proof obligations for goal refinement are defined in [102].

They depend on the goal refinement operator used and complete the classic proof obligations for invariant preservation and for event feasibility. For an abstract goal G and two concrete goals G_1 and G_2 (for an event `G`, `G_Guard` represents the guards of G and `G_Post` represents the post condition of its actions):

— For the *AND* operator (variables involved in subgoals must be distinct), 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 the *OR* operator, they are
  - $G_1\_Guard \Rightarrow G\_Guard$
  - $G_2\_Guard \Rightarrow G\_Guard$
  - $G_1\_Post \Rightarrow G\_Post$
  - $G_2\_Post \Rightarrow G\_Post$
  - $(G_1\_Guard \wedge G_1\_Post) \Rightarrow \neg G_2\_Guard$
  - $(G_2\_Guard \wedge G_2\_Post) \Rightarrow \neg G_1\_Guard$

— For the *MILESTONE* operator, 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)

Nevertheless, the generated *B System* specification does not contain the system structure, that are variables with their associated invariant and constants with their associated properties. This structure is provided by the translation of SysML/KAOS domain models. The corresponding translation rules are fully described in annex B. In short, domain models identify formal components. Concepts give *B System* types while individuals give set elements. The rules also allow the extraction of the initialisation of state variables.

7.3 Requirements and Modeling Strategy

We have considered three reference documents throughout this work to define system requirements and characterise the application domain. The first one, [79], gives the general principles of the *hybrid ERTMS/ETCS level 3* protocol (HEEL3) and defines requirements to be considered. Readers can refer to this document for a full description of the case study and of its requirements. The second one, [54], offers a technical and detailed description of the protocol specification. It provides the safety requirements that the system must guarantee. The work described in this paper is

consistent with the revisions made in [54] after the ABZ2018 conference. The third one, [107], proposed by *Network Rail, UK* and *ProRail, Netherlands* describes HEEL3 while clarifying the specificities of the latter in comparison with the other protocols of the ERTMS/ETCS family. It concisely presents the high-level objectives related to the quality of rail transport and how each protocol of the ERTMS/ETCS family contributes to their achievements, along with their advantages and limitations.

As a reminder, HEEL3 has been proposed to optimize the use and occupation of railways [54, 79, 107]. It thus proposes the division of the track into separate entities, each named Trackside Train Detection (TTD). In addition, each TTD is subdivided into sub-entities called Virtual Sub-Sections (VSS). A TTD has two possible states: *free* and *occupied* with a safety invariant stating that if a train is located on a TTD, then the state of the TTD must be set to *occupied*. In addition to these two states, a VSS may have the *unknown* or the *ambiguous* state. The *ambiguous* state is used when the information available to the system suggest that two trains are potentially present on the VSS. The *unknown* state is used when the system can guarantee neither the presence nor the absence of a train on the VSS. For an optimal safety, Movement Authorities (MA) are evaluated and assigned to each connected train. The MA of a train designates a portion of the track on which it is guaranteed to move safely. ERTMS (European Rail Traffic Management System) designates a protocol and a set of tools that allow a train to know and report its position. Similarly, TIMS (Train Integrity Monitoring System) designates the component that allows a train to know and report its integrity and its size. HEEL3 considers three kinds of trains: (1) trains equipped with TIMS (TIMS trains), which can report themselves as *integer* or not; (2) trains equipped with ERTMS (ERTMS trains), which can report their position (connected trains) or not (unconnected trains); and finally, (3) trains that are equipped neither with a ERTMS nor with a TIMS (unconnected trains).

7.3.1 Modeling Strategy

The SysML/KAOS requirements engineering method allows the progressive construction of system requirements from the refinement of stakeholder needs. Thus, even if the management of VSSs is the purpose of the case study [79], an essential part of our work is devoted to putting it into perspective with more abstract objectives that will explain what VSSs are useful for. We have chosen to consider that the general objective that the system must fulfil is: *safely move trains on the track*. The most abstract level of the formal specification that has been built is a translation of this general objective. Its concrete refinement levels are representations of the choices allowing the achievement of the objective. The specification includes seven refinement levels explicitly related to stakeholder needs through SysML/KAOS models. The SysML/KAOS method makes it possible to

trace the source and justify the need for each formal component and its contents. Within the formal specification, the scheduling of events and the refinement strategy are enforced using proof obligations expressed as theorems. The specification is devoted to the formalisation of system functional goals and in their verification with regard to domain properties and safety invariants. The environment behavior is left nondeterministic with respect to domain constraints modeled in SysML/KAOS domain models.

7.3.2 Requirements Modeling

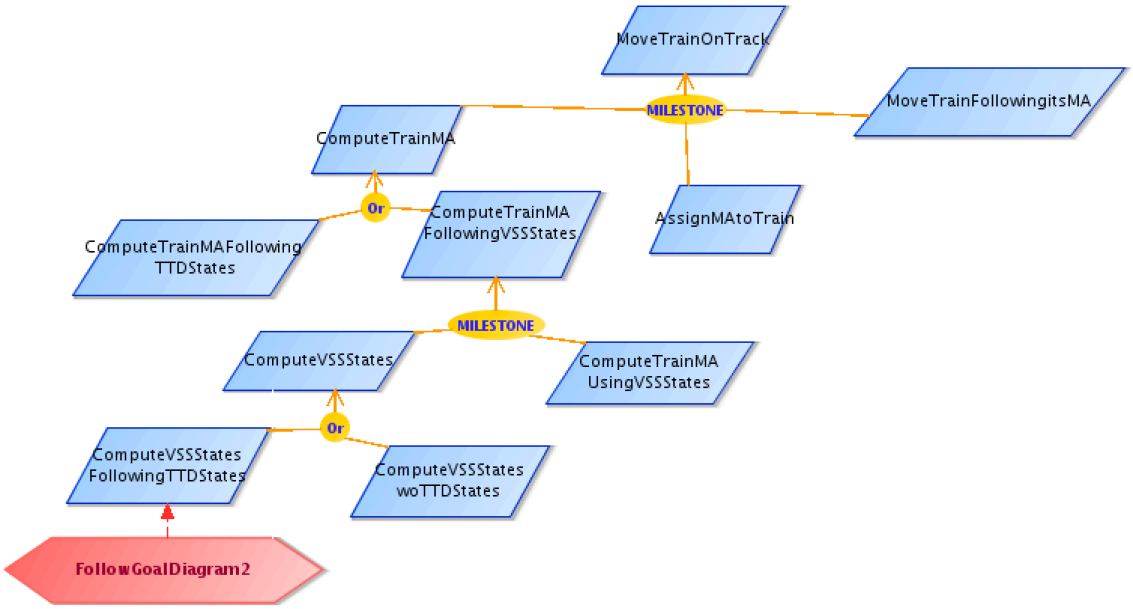


Figure 7.1 – The SysML/KAOS goal diagram

Figure 7.1 is an excerpt from the SysML/KAOS functional goal diagram focused on the main system purpose: *safely move trains on the track* (`MoveTrainOnTrack`). To achieve it, the system must ensure that the train has a valid MA (`ComputeTrainMA`). If the MA has been recomputed, then the system must assign the new MA to the train (`AssignMAtoTrain`). Finally, the train has to move following its assigned MA (`MoveTrainFollowingItsMA`). The second refinement level of the SysML/KAOS goal diagram focuses on data needed to determine the MA of a train : the MA computation can be based only on TTD states (`ComputeTrainMAFollowingTTDStates`) or following VSS states (`ComputeTrainMAFollowingVSSStates`) [54]. When the computation is only based on TTD states, it corresponds to the *ERTMS/ETCS Level 2*

protocol. When VSS states are involved, it corresponds to the *ERTMS/ETCS Level 3* protocol. The MA computation based on VSS states requires the update of the states of VSSs (`ComputeVSSStates`) and the computation of the MA (`ComputeTrainMAUsingVSSStates`). Finally, depending on the type of the ERTMS/ETCS level 3 implementation, it is possible to use or not the TTD states when computing the VSS states (Table 1 of [107]). Goal `ComputeVSSStateswoTTDStates` represents the case where TTD states are not required (*virtual (without train detection)* level 3 type), with the disadvantage of only allowing the circulation of trains equipped with TIMS. Goal `ComputeVSSStatesFollowingTTDStates`, on the other hand, represents the case where TTD states are used to compute VSS states (*hybrid* level 3 type).

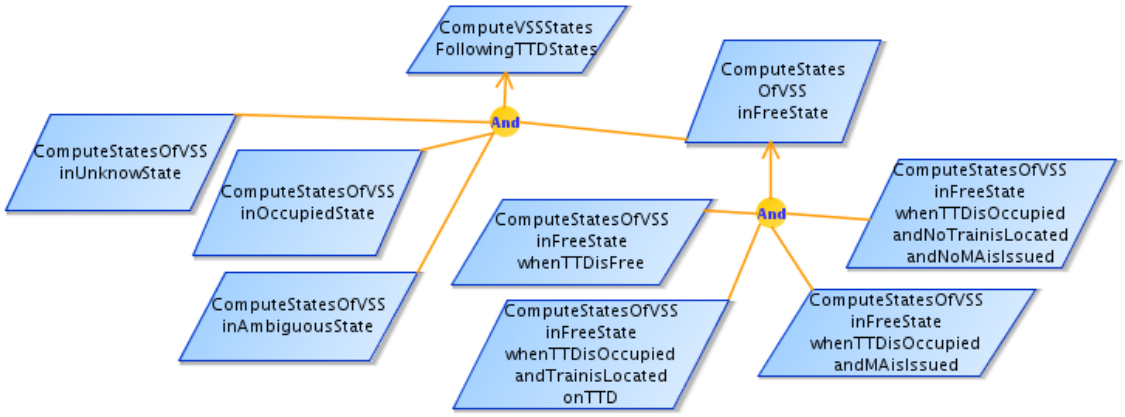


Figure 7.2 – SysML/KAOS goal diagram of the VSS state computation purposes

Figure 7.2 is an excerpt from the SysML/KAOS functional goal diagram focused on the purpose of VSS state computation with the use of TTD states (`ComputeVSSStatesFollowingTTDStates`). To compute the state of VSSs, it is necessary to take into account their previous status (Figure 7 of [54]). For instance, goal `ComputeStatesOfVSSinUnknownState` deals with VSSs that were previously in the unknown state while goal `ComputeStatesOfVSSinFreeState` deals with those that were previously in the free state. Compared to [58], Figure 7.2 has been updated to take into account the revisions made within [54]. The last refinement level is focused on VSSs previously in the free state. Its goals come from requirements of the updated transition *#1A* of Table 2 of [54]. When the TTD is free, then the VSSs remain free (`ComputeStatesOfVSSinFreeStateWhenTTDisFree`). When the TTD is occupied and no train is located on it while no MA is issued, then the VSSs move in the unknown state (`ComputeStatesOfVSSinFreeStatewhenTTDisOccupiedandNoTrainisLocatedandNoMAisIssued`). Goal `ComputeStates-`

OfVSSinFreeStatewhenTTDisOccupiedandTrainisLocatedonTTD deals with VSSs previously in the free state when TTDs are occupied and trains are located on them while goal ComputeStatesOfVSSinFreeStatewhenTTDisOccupiedandMAisIssued is triggered when a TTD is occupied and a MA is issued.

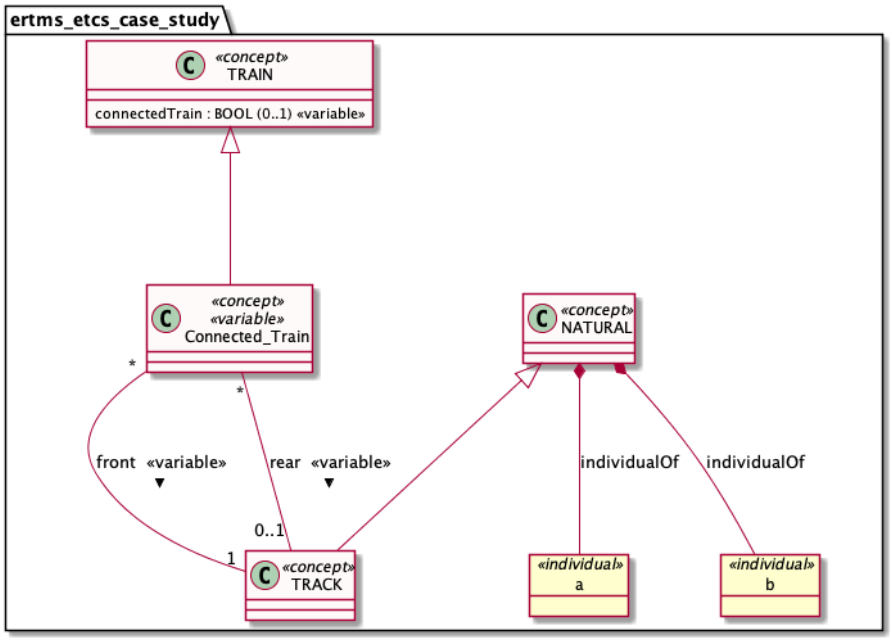
7.4 Model Details

From the goal model, we distinguish seven refinement levels which are translated into seven *B System* components. The rest of this section consists of a presentation of SysML/KAOS domain models associated with the first three refinement levels of the main goal diagram and of a description of the *B System* specification obtained. We also provide an overview of the specification of the fifth refinement level, which introduces the four events in charge of updating VSS states. Domain models are illustrated using the syntax proposed by the *SysML/KAOS domain modeling tool* [56], a tool implemented on top of *Jetbrains MPS* [87] and *PlantUML* [116] to provide a proof of concept of the SysML/KAOS domain modeling language.

7.4.1 The Root Level

Figure 7.3 represents the domain model associated with the root goal MoveTrainOnTrack of the diagram of Figure 7.1. It represents the entities needed for the specification of the movement of a train on the track and their characteristics. For instance, concept TRAIN models the set of trains. Association connectedTrain models the subset of TRAIN that broadcast their location at least once and for each, the current connection status. Concept Connected_Train is used to represent the set of trains for which the connection status is known. Association front models the estimated position of the front of each connected train. For each connected train equipped with a TIMS, association rear models the estimated position of its rear: the rear is deduced from the front and length of the train, since a train equipped with a TIMS broadcast its length and its integrity. Thus, $dom(front) \setminus dom(rear)$ represents the set of trains equipped with a ERTMS and not equipped with a TIMS.

Logical formulas represent constraints on domain model elements. Each logical formula is prefixed with an identifier p<i>.<j> where <i> designates the refinement level number and <j> identifies the formula in the refinement level. For example, logical formula p0.2 defines TRACK, a subset of the set of natural numbers, as the data range *a..b*. In addition, logical formula p0.4 defines concept Connected_Train as the domain of association connectedTrain (*dom(connectedTrain)*). This definition




```
Logical formulas:
 p0.1: a<b
 p0.2: TRACK=a..b
 p0.3: !tr.(tr : dom(rear) => rear(tr) < front(tr))
 p0.4: Connected_Train = dom(connectedTrain)
```

Figure 7.3 – SysML/KAOS domain modeling of the root level of the goal diagram of Figure 7.1

allows each reference to `Connected_Train` to be replaced by *dom(`connectedTrain`)* in the *B System* specification. This replacement has to be manually done in current release of the SysML/KAOS domain modeling tool, but will be done automatically in future releases.

The Openflexo platform [109] is used to build a tool that federates all the models involved in the SysML/KAOS requirements engineering method. The tool, available at [110], currently allows to build goal models as the ones of Figure 7.1 and 7.2 and to associate the related domain models. It was not used to build all domain models because it does not yet support rules to generate *B System* models. However, it has been used to represent the domain model associated with the root level of the diagram of Figure 7.1. An overview of the constructed domain model, similar to that of Figure 7.3, is provided by Figure 7.4. The upper black box contains the definition

of logical formulas that constrain domain model elements. Each blue rectangle represents a defined concept; each yellow rectangle represents an association; gray ovals represent default datatypes (like *NATURAL*) and red rectangles represent the other concepts. For instance, concepts `Connected_Train` and `TRACK` are defined concepts given by the first two logical formulas. Elements `a` and `b` are constant individuals (each represented by a green rectangle) while `front` and `rear` are associations.

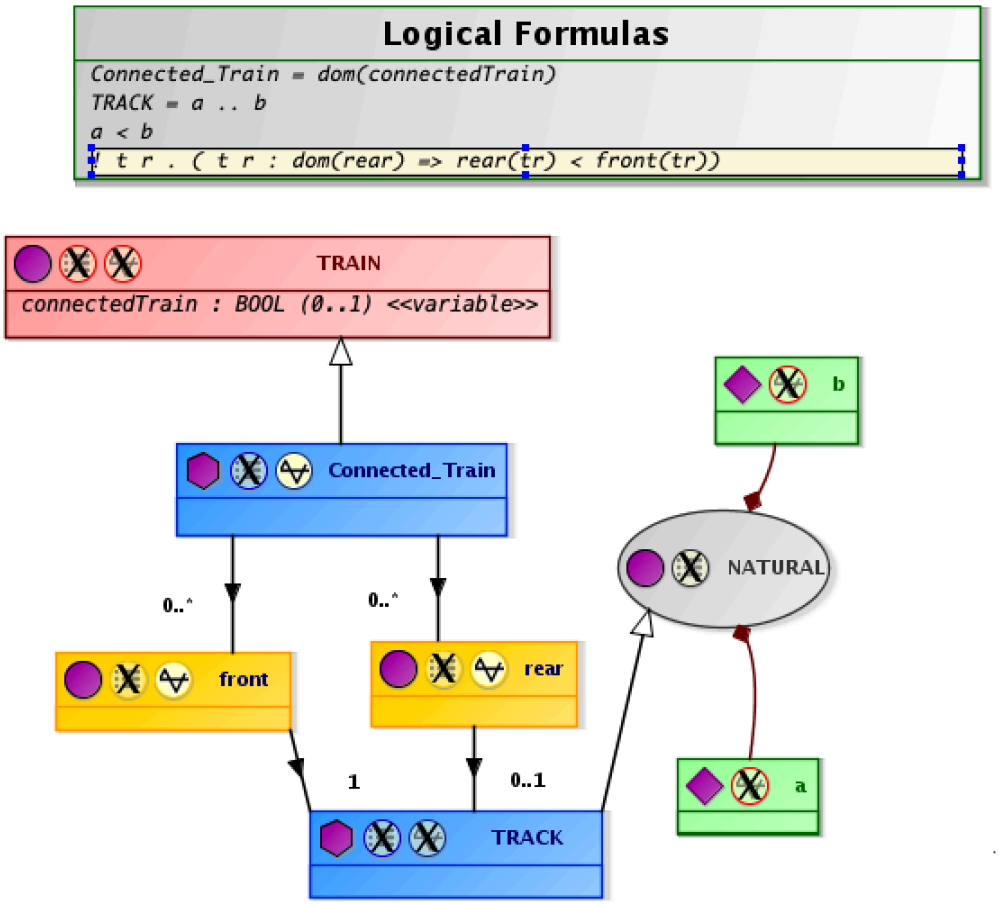


Figure 7.4 – Overview of the root domain model constructed with the Openflexo SysML/KAOS tool

SYSTEM ertms_etcs_case_study
SETS TRAIN
CONSTANTS a b TRACK
PROPERTIES
axm1: $a \in \mathbb{N}$ axm2: $b \in \mathbb{N}$ p0.1: $a < b$
p0.2: $TRACK = a \mathinner{..} b$
VARIABLES connectedTrain front rear
INVARIANT
inv1: $connectedTrain \in TRAIN \nrightarrow BOOL$
inv2: $front \in dom(connectedTrain) \rightarrow TRACK$
inv3: $rear \in dom(connectedTrain) \nrightarrow TRACK$
p0.3: $\forall tr \cdot (tr \in dom(rear) \Rightarrow rear(tr) < front(tr))$

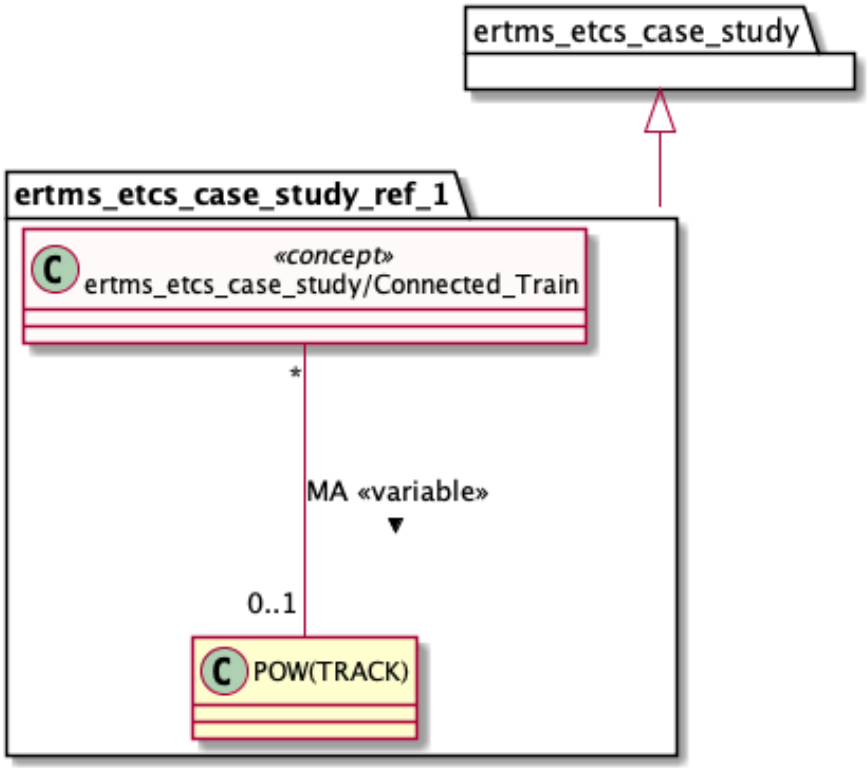
Event MoveTrainOnTrack $\widehat{=}$
any tr len n_rear
where
grd1: $tr \in connectedTrain^{-1}[\{TRUE\}]$
grd2: $len \in \mathbb{N}_1$
grd3: $front(tr) + len \in TRACK$
grd4: $tr \in dom(rear) \Rightarrow n\_rear = rear \lhd\!\!\!\!- \{tr \mapsto rear(tr) + len\}$
grd5: $tr \notin dom(rear) \Rightarrow n\_rear = rear$
then
act1: $front(tr) := front(tr) + len$
act2: $rear := n\_rear$
END
END

Figure 7.5 – *B System* specification of the root level of the goal diagram of Figure 7.1

Figure 7.5 represents the *B System* specification obtained from the translation of the root level of the goal diagram of Figure 7.1 and of the associated domain model of Figure 7.3. The domain model gives rise to sets, constants, properties, variables and invariants of the formal specification. Logical formulas involving variables give rise to invariants and the others to properties. No variable is defined to represent the variable concept `Connected_Train` because a logical formula, `p0.4`, defines it as equivalent to *dom(`connectedTrain`)*. Thus, any reference to `Connected_Train` is replaced by *dom(`connectedTrain`)* in the *B System* specification. The root goal is translated into an event for which the body has been manually specified: the movement of a connected train (`grd1`) results in the incrementation of the position of its front (`act1`) and its rear (`act2` in the case of an *integer* train: $tr \in dom(rear)$) of the value corresponding to the movement. Of course, the movement can only be done if the train stays on track (`grd3`). Another event is defined within the complete specification [134] to handle train exits.

7.4.2 The First Refinement Level

Figure 7.6 represents the domain model associated with the first refinement level of the SysML/KAOS goal diagram of Figure 7.1. It refines the one associated with the root level (Figure 7.3) and introduces an association named `MA` representing the MA assigned to a connected train. The MA of a train is modeled as a segment




```
Logical formulas:
 p1.1: !tr. (tr : dom(MA)
  => #p,q.(p..q<:TRACK & p<=q & MA(tr)=p..q)))
 p1.2: !tr. (tr : dom(MA) => (front(tr) : MA(tr)))
 p1.3: !tr. (tr : dom(rear) & tr : dom(MA) => rear(tr) : MA(tr))
 p1.4: !tr1,tr2. ((tr1 : dom(MA) & tr2 : dom(MA) & tr1 /= tr2)
  => MA(tr1) /\ MA(tr2)={})
```

Figure 7.6 – SysML/KAOS domain modeling of the first refinement level of the goal diagram of Figure 7.1

of the track (`p1.1`), containing the train (`p1.2` and `p1.3`). Finally, logical formula `p1.4` asserts that the MA assigned to two different trains must be disjoint. Logical formulas `p1.2` and `p1.3` are gluing invariants, linking the concrete variable `MA` with the abstract variables `front` and `rear`.

REFINEMENT ertms_etcs_case_study_ref_1
REFINES ertms_etcs_case_study
VARIABLES connectedTrain front rear MA MAtemp
INVARIANT
inv1: $MA \in dom(connectedTrain) \pfun \mathbb{P}(TRACK)$
p1.1: $\forall tr \cdot (tr \in dom(MA) \Rightarrow (\exists p, q \cdot (p \upto q \subseteq TRACK \land p \leq q \land MA(tr) = p \upto q)))$
p1.2: $\forall tr \cdot (tr \in dom(MA) \Rightarrow front(tr) \in MA(tr))$
p1.3: $\forall tr \cdot (tr \in dom(rear) \cap dom(MA) \Rightarrow rear(tr) \in MA(tr))$
p1.4: $\forall tr1, tr2 \cdot ((\{tr1, tr2\} \subseteq dom(MA) \land tr1 \neq tr2) \Rightarrow MA(tr1) \cap MA(tr2) = \emptyset)$
inv6: $MAtemp \in dom(connectedTrain) \pfun \mathbb{P}(TRACK)$
inv7: $\forall tr \cdot (tr \in dom(MAtemp) \Rightarrow (\exists p, q \cdot (p \upto q \subseteq TRACK \land p \leq q \land MAtemp(tr) = p \upto q)))$
theorem s1: *ComputeTrainMA_Guard* $\Rightarrow$ *MoveTrainOnTrack_Guard*
theorem s2: *ComputeTrainMA_Post* $\Rightarrow$ *AssignMAtoTrain_Guard*
theorem s3: *AssignMAtoTrain_Post* $\Rightarrow$ *MoveTrainFollowingItsMA_Guard*
theorem s4: *MoveTrainFollowingItsMA_Post* $\Rightarrow$ *MoveTrainOnTrack_Post*

Event
ComputeTrainMA $\widehat{=}$
any tr p q len
where
grd1: $tr \in connectedTrain^{-1}[\{TRUE\}]$
grd2: $p \upto q \subseteq TRACK \land p \leq q$
grd3: $front(tr) \in p \upto q$
grd4: $tr \in dom(rear) \Rightarrow rear(tr) \in p \upto q$
grd5: $p \upto q \cap union(ran(\{tr\} \domrest MA)) = \emptyset$
grd6: $len \in \mathbb{N}_1$
grd7: $front(tr) + len \in TRACK$
then
act1: $MAtemp(tr) := p \upto q$
END

AssignMAtoTrain $\widehat{=}$
any tr len
where
grd1: $tr \in connectedTrain^{-1}[\{TRUE\}] \cap dom(MAtemp)$
grd2: $front(tr) \in MAtemp(tr)$
grd3: $tr \in dom(rear) \Rightarrow rear(tr) \in MAtemp(tr)$
grd4: $MAtemp(tr) \cap union(ran(\{tr\} \domrest MA)) = \emptyset$
grd5: $len \in \mathbb{N}_1$
grd6: $front(tr) + len \in MAtemp(tr)$
then
act1: $MA(tr) := MAtemp(tr)$
END

MoveTrainFollowingItsMA $\widehat{=}$
any tr len n_rear
where
grd1: $tr \in connectedTrain^{-1}[\{TRUE\}] \cap dom(MA)$
grd2: $len \in \mathbb{N}_1$
grd3: $front(tr) + len \in MA(tr)$
grd4: $tr \in dom(rear) \Rightarrow n\_rear = rear \domrest \{tr \mapsto rear(tr) + len\}$
grd5: $tr \notin dom(rear) \Rightarrow n\_rear = rear$
then
act1: $front(tr) := front(tr) + len$
act2: $rear := n\_rear$
END
END

Figure 7.7 – *B System* specification of the first refinement level of the diagram of Figure 7.1

Figure 7.7 represents the *B System* model obtained from the translation of the first refinement level of the goal diagram of Figure 7.1 and of the associated domain model of Figure 7.6. Each refinement level goal is translated into an event for which the body has been manually specified: the current MA of the train is computed and stored into a variable named `MAtemp` (event `ComputeTrainMA`). Because the computation of the MA is out of the scope of the case study [79], the event simply nondeterministically chooses an MA, with respect to the safety invariants. This MA is then assigned to the train by updating the variable `MA` (event `AssignMAtoTrain`) and taken into account for the train displacement (event `MoveTrainFollowingItsMA`). Theorems `s1`, `s2`, `s3` and `s4` represent the proof obligations related to the use of the *MILESTONE* operator between the root and the first refinement levels. Since each proof obligation has been modeled as a *B System* theorem, it has been proved based on system properties and invariants. To deal with the fact that *B System* does not currently support the temporal logic, we have used the proof obligation $G1\_Post \Rightarrow G2\_Guard$ for invariants `s2` and `s3`, instead of $\Box(G1\_Post \Rightarrow \Diamond G2\_Guard)$ (Sect. 7.2.4), since:

$$(G1\_Post \Rightarrow G2\_Guard)$$
$$\Rightarrow (\Box(G1\_Post \Rightarrow \Diamond G2\_Guard))$$

By using this trick, we replace the proof obligation involving operators of the temporal logic with a more constraining proof obligation. The trick is only useful if it is possible and easier to discharge the newly introduced proof obligation.

To ease understanding of the represented *B System* specification, theorems related to *SysML/KAOS* refinements are symbolically represented. They are obviously fully represented in the Rodin project. For instance, the full specification of `s1` is given below:

theorem `s1`:

$$\begin{aligned}
&\forall tr, p, q, len \cdot (((tr \in connectedTrain^{-1}[\{TRUE\}]) \\
&\quad \wedge (p \mathinner{..} q \subseteq TRACK \wedge p \leq q) \ \wedge (front(tr) \in p \mathinner{..} q) \\
&\quad \wedge (tr \in dom(rear) \Rightarrow rear(tr) \in p \mathinner{..} q) \\
&\quad \wedge (p \mathinner{..} q \cap union(ran(\{tr\} \lhd MA)) = \emptyset) \\
&\quad \wedge (len \in \mathbb{N}_1) \ \wedge (front(tr) + len \in TRACK)\ ) \\
&\quad \Rightarrow \ (\ (tr \in connectedTrain^{-1}[\{TRUE\}]) \\
&\quad \wedge (len \in \mathbb{N}_1) \ \wedge (front(tr) + len \in TRACK)\ ))
\end{aligned}$$

It expresses the fact that the activation of the guard of `ComputeTrainMA` for certain parameters is sufficient for the activation of the guard of `MoveTrainOnTrack` for this same group of parameters.

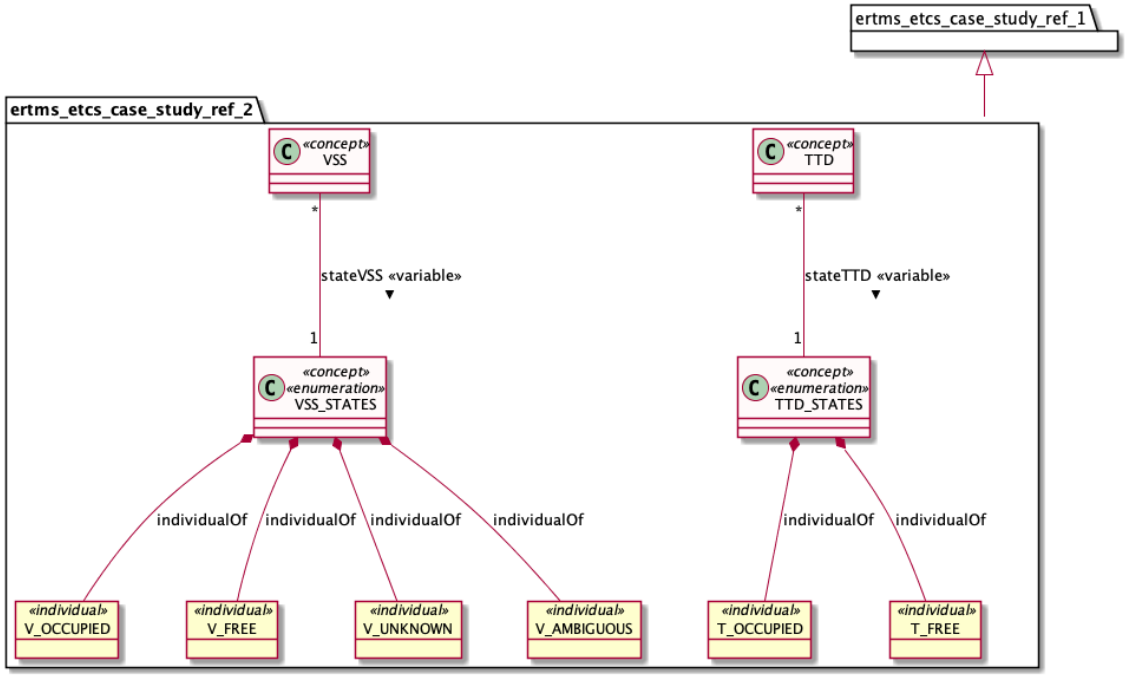
ClearSy is currently working on a release of *Atelier B* that directly supports the specification of SysML/KAOS refinement operators, when refining *B System* events, such as *ref_and* and *ref_or*. These keywords are used by the proof generator to automatically generate the right proof obligations.

7.4.3 The Second Refinement Level

Figure 7.8 represents the domain model associated with the second refinement level of the diagram of Figure 7.1. It refines the one associated with the first refinement level and introduces two concepts named `TTD` and `VSS`. Associations `stateTTD` and `stateVSS` represent the states of the corresponding concepts. Logical formulas `p2.1`..`p2.8` define each TTD as a segment of the track and each VSS as a segment of a TTD. Logical formulas `p2.9` and `p2.10` are used to state that if a train `tr` is located on a TTD, then its state must be occupied: a train $tr \in TRAIN$ is located on $ttd \in TTD$ if $front(tr) \in ttd$ (`p2.9`) or if `tr` is equipped with a TIMS ($tr \in dom(rear)$) and $(rear(tr)..front(tr)) \cap ttd \neq \emptyset$ (`p2.10`). Finally, logical formulas `p2.11`..`p2.13` state that two different trains must be on disjoint parts of the track: for two trains `tr1` and `tr2`, if they are equipped with TIMS, then the track segments that they occupy should just be disjoint (`p2.11`); if they are on the same TTD and one of them, `tr2`, is not equipped with a TIMS, then, the second, `tr1`, must be equipped with a TIMS and `tr2` must be in the rear of `tr1` (`p2.12`); if none of them is an integer train, then they must be in two distinct TTDs (`p2.13`). Logical formulas `p2.9` and `p2.10` are gluing invariants, linking the concrete variable `stateTTD` with the abstract variables `front` and `rear`. The *B System* specification obtained from the translation of the second refinement level includes the result of the translation of the domain model of Figure 7.8, two new events (`ComputeTrainMAFollowingTTDStates`, `ComputeTrainMAFollowingVSSStates`), an extension of event `MoveTrainFollowingItsMA` taking into account the new safety invariants and theorems representing the proof obligations related to the usage of the *OR* operator between the first and second refinement levels.

7.4.4 The Fifth Refinement Level

For the fifth refinement level, corresponding to the first refinement level of the goal diagram of Figure 7.2, the *B System* specification introduces four events obtained from the translation of goals and five theorems representing the proof obligations related to the use of the *AND* operator between the fourth and the fifth refinement levels. These theorems are :




```
Logical formulas:
 p2.1: TTD <: POW1(TRACK) p2.2: union(TTD) = TRACK p2.3: inter(TTD) = {}
 p2.4: !ttd. (ttd : TTD => #p,q.(p..q<:TRACK & p<q & ttd=p..q)))
 p2.5: VSS <: POW1(TRACK) p2.6: union(VSS) = TRACK p2.7: inter(VSS) = {}
 p2.8: !vss. (vss : VSS => #p,q,ttd.(ttd : TTD & p..q<:ttd & p<q & vss=p..q)))
 p2.9: !ttd,tr. ( tr : dom(front) \ dom(rear) & ttd : TTD
       & front(tr) : ttd) => (( ttd |-> T_OCCUPIED ) : stateTTD)
 p2.10: !ttd,tr. (tr : dom(rear) & ttd : TTD & (rear(tr)..front(tr))/\ttd /= {})
       => (( ttd |-> T_OCCUPIED ) : stateTTD)
 p2.11: !tr1,tr2. (tr1 : dom(rear) & tr2 : dom(rear) & tr1 /= tr2)
       => ( (rear(tr1)..front(tr1))/\(rear(tr2)..front(tr2))={})
 p2.12: !tr1,tr2,ttd.(tr1 : dom(rear) & tr2 : dom(front)\dom(rear) & tr1 /= tr2
       & ttd : TTD & front(tr2) : ttd & rear(tr1)..front(tr1))/\ttd /= {})
       => ( front(tr2)<rear(tr1))
 p2.13: !tr1,tr2,ttd. ( tr1 : dom(front)\dom(rear) & tr2 : dom(front)\dom(rear)
       &tr1 /= tr2 & ttd : TTD & front(tr1) : ttd) => ( front(tr2) /: ttd)
```

Figure 7.8 – SysML/KAOS domain modeling of the second refinement level of the goal diagram of Figure 7.1

theorem s1: *ComputeStatesOfVSSinUnknownState_Guard*
⇒ *ComputeVSSStatesFollowingTTDStates_Guard*
theorem s2: *ComputeStatesOfVSSinOccupiedState_Guard*
⇒ *ComputeVSSStatesFollowingTTDStates_Guard*
theorem s3: *ComputeStatesOfVSSinAmbiguousState_Guard*
⇒ *ComputeVSSStatesFollowingTTDStates_Guard*
theorem s4: *ComputeStatesOfVSSinFreeState_Guard*
⇒ *ComputeVSSStatesFollowingTTDStates_Guard*
theorem s5: (*ComputeStatesOfVSSinUnknownState_Post*
∧ *ComputeStatesOfVSSinOccupiedState_Post*
∧ *ComputeStatesOfVSSinAmbiguousState_Post*
∧ *ComputeStatesOfVSSinFreeState_Post*)
⇒ *ComputeVSSStatesFollowingTTDStates_Post*

The formal specification has been verified using *Rodin* [35]. We have in particular discharged all the proof obligations associated with safety invariants that were identified and with refinement operators used within goal diagrams. The full specification can be found in [134].

7.5 Discussion

7.5.1 Validation And Verification

The SysML/KAOS method not only makes it possible to verify the consistency of requirements and their refinement logic, but also to better present and validate the requirements with the various stakeholders. Indeed, SysML/KAOS includes semi-formal languages for a high-level representation of system goals and application domain properties. This ensures a better reusability and readability of models. Improved readability of models was assessed among members of the FORMOSE project. Indeed, of the fifteen or so surveyed members representing various academic [1] and industrial [2] partners, all found the readability of SysML/KAOS models much better than that of a *B System* specification. The improved readability was also confirmed by stakeholders of the *Municipality of Montreal (la Ville de Montréal -VdM)* [3]

1. *University Paris Est Créteil, France; University of Sherbrooke, Canada; Télécom SudParis, France; Institut Mines-Telecom Brest, France*
2. *THALES, France; ClearSy Systems Engineering, France; Openflexo, France*
3. *Montreal* is the second-largest city in *Canada* and the largest city in *Québec*

where the SysML/KAOS method was used to deal with requirements of a road transportation system [6]: four validation sessions were organised and allowed to introduce SysML/KAOS to VdM stakeholders and to obtain their feedbacks related to the constructed SysML/KAOS models.

SysML/KAOS also includes rules for obtaining a *B System* specification and the proof obligations required to guarantee consistency of goal refinements and accuracy of requirements with respect to environment constraints. This ensures a strong traceability between the *B System* specification and goal diagrams which are abstractions of needs identified within the reference documents [54, 107].

The *B System* specification, however, remains quite abstract and needs to be further refined in order to come up with an implementable model. Indeed, the specification is in the *problem space*, focused on the justification (with regard to stakeholders needs) and verification of system requirements.

Using the SysML/KAOS method, we have quickly built the refinement hierarchy and we have determined and formally expressed the safety invariants. The method bridges the gap between the system textual description and its *B System* specification. Table 7.1 summarises the key characteristics related to the formal specification and to proof obligations. The expression of theorems representing proof obligations associated to SysML/KAOS refinement operators was difficult because there is no way in Rodin to designate the guard and the post condition of an event within logical formulas. The proofs have been discharged using the Rodin tool extended with *Atelier B provers* [115] and *SMT solvers* [127]. Customised auto-tactic/post-tactic profiles, including the added provers, with extended timeouts, have been defined. In previous version of the formal specification [58], some proof obligations were difficult because of conditional actions such as
$rear := (\{TRUE \mapsto rear \lhd\!\!\!\!- \{tr \mapsto rear(tr) + len\}, FALSE \mapsto rear\})(bool(tr \in dom(rear)))$
defined in component `ertms_etcs_case_study` (Figure 7.5) to simulate an *if-then-else*. To simplify these proofs, we have introduced conditionally defined parameters such as parameter `n_rear` of event `MoveTrainOnTrack` (component `ertms_etcs_case_study`) which is defined with guards `grd4` or `grd5` following condition $tr \in dom(rear)$. This has significantly increased the number of automatically discharged proof obligations.

Table 7.1 – Key characteristics related to the formal specification

Refinement level	L0	L1	L2	L3	L4	L5	L6
Invariants	4	11	13	1	0	5	6
Proof Obligations (PO)	28	58	78	10	0	9	28
Automatically Discharged POs	26	54	58	10	0	8	18
Interactively Discharged POs	2	4	20	0	0	1	10

The use of *ProB* [95] made it possible to better validate the adequacy between the *B System* specification and the needs identified in reference documents.

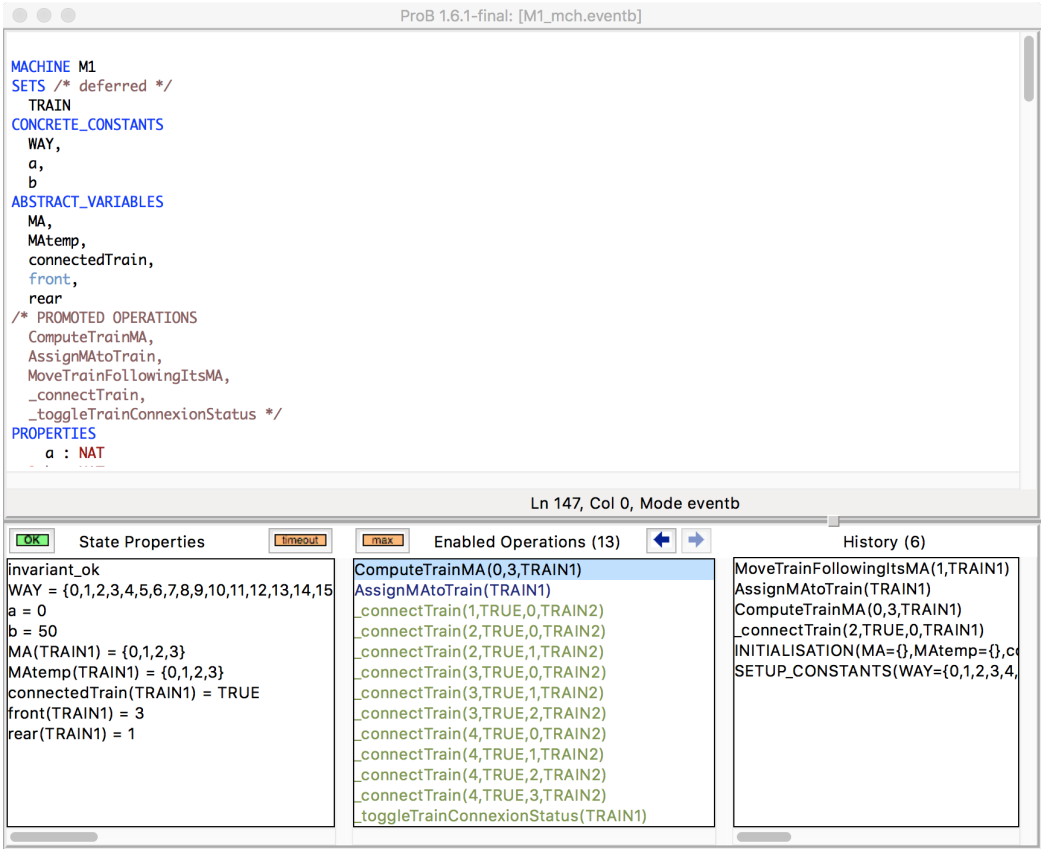


Figure 7.9 – Overview of the animation of the formal specification through proB

For example, Figure 7.9 is an overview of the animation of the specification of Figure 7.7. The top view presents an excerpt of the Event-B specification. The bottom view presents, from left to right, the current state of the system structure (view *State Properties*), the enabled operations (view *Enabled Operations*) and a summary of the performed operations (view *History*). View *State Properties* shows that a connected train `TRAIN1` is present on track on segment $1\,..\,3$. Its assigned MA extends over segment $0\,..\,3$ ($MA(TRAIN1) = \{0,1,2,3\}$). View *History* shows that `TRAIN1` was connected while being on segment $0\,..\,2$. An MA has then been computed and assigned to it and, following its assigned MA, it has moved from segment $0\,..\,2$ to segment $1\,..\,3$. Finally, view *Enabled Operations* shows that it is possible to disconnect `TRAIN1`, to connect another train or to proceed with computation of MAs. One conclusion of the animation work is that the principles described in

documents [54,79,107] do not guarantee the absence of train collisions. Indeed, as reported in [9], since the movement of unconnected trains is allowed, nothing is specified to guarantee that an unconnected train will not hit another train (connected or not). The only guarantee that the safety invariants expressed may bring is that a connected train will never hit another train.

7.5.2 Comparison with the Other Approaches

HEEL3 has been the purpose of several works presented at the ABZ2018 conference. Each work is characterised by the involved formal specification, verification and validation approach.

In Abrial's article [9], the Event-B method is used to specify the case study. This work, similar to ours, distinguishes the modeling of system requirements and of environment assumptions from the formal specification task. For Abrial [9], requirements modeling includes the elaboration of a reference document called the *requirement document* which defines two special kinds of elements namely *environment requirements* and *system functionalities*. Environment requirements are assumptions about the environment structure and behavior while system functionalities are statements about what is expected from the system. A refinement strategy is then defined to link elements of the requirement document to the four refinement levels of the specified Event-B model. However, this refinement strategy is informally specified and only slightly justified. The SysML/KAOS method makes it possible to explicit the refinement links through refinement operators. In addition, the proof obligations associated with these operators make it possible to ensure that the defined refinement strategy is formally correct when considering requirement specifications.

In [52], Dghaym *et al.* use *iUML-B* state and class diagrams [125] to define a high-level representation of HEEL3's principles. An Event-B specification is then semi-automatically deduced and verified. The high-level representation is built to improve specification validation by domain experts, since it is more readable than plain Event-B. Class diagrams are used to represent domain entities, their constraints and events where they are involved. However, there is no link between events. The state changes of system variables are modeled with statemachines. In SysML/KAOS, goal models are used to represent system requirements that produce events while domain entities and constraints are represented in domain models. This separation of concerns avoids conflicts during the generation of the formal specification as is the case with iUML-B class diagrams and statemachines when a variable appears in both models. Dghaym *et al* [52] consider the controller as the component that receives messages from trains and TTD and calculates the free VSS sections; the other components (trains, trackside equipments) being part of

the environment. Two kinds of trains are considered: those that communicate with controller and those that do not. Our study, on the other side, considers the controller as the component that computes available VSSs, updates train MAs and ensures that trains move following their assigned MAs (Figure 7.1). This is necessary since we consider that the main aim of the controller is to ensure the safe movement of trains on track. We consider not only trains that communicate and trains that do not, but we also distinguish trains that only communicate their position (ERTMS trains that are non-TIMS) from trains that communicate their position, length and integrity (TIMS trains). As in [52], the contiguity of VSSs is ensured through the modeling of VSSs as ranges of integers. Finally, as in [9], a refinement strategy is defined to provide a plan for how the Event-B specification is intend to be built. The Event-B model includes 8 refinement levels: 3 to model environment components, 4 for the controller and 1 for a component that computes and assigns MAs to trains. Rodin provers have been used to discharge proof obligations and, the ProB model checker, to find counter examples in case of proof failure and animate the Event-B model.

In [18], the SPIN model checker [80] is used to formally verify and validate a specification of the case study made with PROMELA. The use of PROMELA to specify the case study makes it possible to better take advantage of SPIN potentialities, but it limits the expressiveness of the specification as well as its readability. Moreover, it is only possible to perform model checking, unlike specification languages like *B* or *Event-B* that allow in addition to perform theorem proving. Unlike specifications built using refinement-based formal methods, a specification in PROMELA is hardly organisable into abstraction layers, making it difficult to specify complex systems and to validate these specifications.

In [77], a *B* specification [7] is proposed for a function called *Virtual Block Function* that computes VSS states following the HEEL3's principles. The specification is validated and executed at runtime using ProB. VSS state transitions are encoded using B definitions and operations: each transition guard is modeled as a definition while the transition action is described in an operation. A special operation is defined to handle priorities between transitions. The specification includes 13 refinement levels and 14 definition files. As in [18], the behaviors and constraints of the environment are explicitly modeled in a separate part of the specification. The environment specification handles the real state of components such as physical positions of trains.

In [44], the case study is specified in Electrum [32], a lightweight formal specification language built on top of Alloy [83]. Electrum extends the Alloy model checker with mutable relations and temporal logic operators. The structure of the case study specification is modeled using Alloy signatures, the specificity of Electrum being that a signature can be variable or static. The system dynamics is modeled using Electrum declarative predicates that relate the values of state variables to their

successors. As in [18], the specification is not organised into abstraction layers and can only be model checked. However, the Electrum Analyzer provides a graphical representation of explored scenarios, through graphs, which enhances validation with stakeholders.

We have also specified, in a companion paper [97], the case study using plain Event-B, in the traditional style. Two distinct specifiers (first author of [97] and first author of this paper) wrote each specification without interacting with each other during specification construction. Critical reviewing by the team was then conducted after the specifications were built. The specification in [97] includes four refinement levels. The TTDs and trains are introduced in the root level and the VSSs are introduced in the second refinement level, as refinements of TTDs. The MAs and VSS states are introduced in the third refinement level (M3), for train movement supervision. A strategy is proposed to prove the determinism of the transitions of VSS states. The state variables of [97] are partitioned into environment variables and controller variables, and similarly for events. Environment events only modify environment variables. Controller events read environment variables and update controller variables. In this paper, we only model controller events; state variables represent the controller view of the environment. The environment behavior is left nondeterministic with respects to domain constraints modeled in SysML/KAOS domain models. The execution ordering and the refinement strategy are enforced using proof obligations expressed as theorems, whereas in [97] there is no proof about these aspects. In [97], the safety properties are introduced in the last refinement level; here, we introduce them in the first (logical formula `p1.4`) and second (logical formulas `p2.9` .. `p2.13`) refinements. The SysML/KAOS method makes it possible to trace the source and justify the need for each formal component and its contents, in relation with the SysML/KAOS goal and domain models. The method therefore represents a more structured and methodological process to the formal specification of system.

7.6 Conclusion and Future Work

This paper focusses on the use of the SysML/KAOS method for the high level modeling of system requirements, of domain properties and of safety invariants related to the hybrid ERTMS/ETCS level 3 protocol [54, 79, 107]. Translation rules, supported by tools [102, 133], have then been applied to obtain a formal specification containing the system structure and the skeleton of events. The Rodin tool [35] has been used to verify and validate the formal specification, especially to prove the

safety invariants and the refinement logic, after the completion of the body of events. The full specification can be found in [134]. A comparison with the other case study specifications published in the ABZ2018 proceedings has been done. This includes a companion paper [97] where the case study is specified using only plain *Event-B*.

The specification obtained using the SysML/KAOS method is in the *problem space*, focused on validation, with regard to stakeholders needs, and verification of system functional requirements. The environment behavior is left nondeterministic within the limits imposed by constraints defined in domain models. Of course, focusing on requirements in itself is not enough. It is necessary to ensure the feasibility of an iterative incremental process encompassing requirements management, architecture design and system development [108]. This requires links between the associated models/specifications such as refinement links between the *B System* formalisation of requirements and the *B System* specification, in the *solution space*, that integrates the necessary design choices to ensure system development. Studying these links is a next step in our study.

Work in progress also aims at improving the representation of logical formulas (to make them more user-friendly) and at studying the propagation of updates and proof errors from *B System* specifications to SysML/KAOS models.

Chapitre 8

Spécification formelle des exigences d'un système de transport urbain : cas de la Ville de Montréal

Résumé

Ce chapitre décrit un cas d'application de la méthode *SysML/KAOS* dans le cadre de la spécification des exigences d'un système de transport routier pour le compte de la *Ville de Montréal (VdM)*, la deuxième plus grande ville du *Canada* et la plus grande ville du *Québec*. Le système de transport a initialement été développé à partir d'exigences non structurées représentées par de volumineux documents textuels et schématiques. La VdM a en conséquence émis le souhait d'explorer de nouveaux moyens d'organiser et d'analyser les exigences de projets routiers, afin d'augmenter le niveau de confiance quant à leur sureté, leur sécurité, leur utilisabilité et leur réutilisabilité. Ce chapitre présente la spécification, la vérification et la validation formelles des exigences identifiées.

La méthode SysML/KAOS a permis de définir les sept premiers niveaux de raffinement de la spécification. Elle a également permis d'expliciter la centaine d'exigences fonctionnelles et non-fonctionnelles des douze composants (humain, matériel, logiciel et cyber-physique) qui constituent le système de transport routier. En outre, l'utilisation de SysML/KAOS a permis de faciliter la validation des exigences avec les parties prenantes de la VdM qui n'avaient jamais été en contact ni avec les méthodes formelles, ni avec l'ingénierie des exigences. Les outils d'animation, en l'occurrence

ProB et *B-Motion Studio*, ont également contribué à faciliter la validation de la spécification formelle avec les parties prenantes de la VdM. Ce chapitre décrit également les points d'amélioration de l'expressivité des langages SysML/KAOS, identifiés à l'issue des sessions de validation avec la VdM. Il s'agit en l'occurrence de l'introduction (1) d'un moyen de quantifier les impacts et contributions des buts, (2) d'une stratégie de raffinement des buts non-fonctionnels basée sur la définition de formules logiques, (3) d'une approche de raffinement des buts de contribution, et (4) d'un langage de modélisation d'obstacles.

Commentaires

La contribution ici réside dans l'évaluation de la méthode SysML/KAOS sur une étude de cas d'envergure industrielle, en l'occurrence la spécification, la vérification et la validation formelles des exigences d'un système de transport urbain. Cette évaluation a notamment permis de faire connaître et illustrer l'usage de SysML/KAOS et d'identifier des points d'amélioration de l'expressivité de ses langages.

L'évaluation décrite dans ce chapitre a fait l'objet d'un article accepté et publié dans le cadre de la 21^{e} édition de la conférence internationale sur les méthodes formelles d'ingénierie *ICFEM (International Conference on Formal Engineering Methods)*.

La spécification réalisée et l'article afférent ont été élaborés par mes soins en tenant compte des remarques et commentaires issus de mon équipe d'encadrement et des parties prenantes de la *Ville de Montréal*.

Assessment of a Formal Requirements Modeling Approach on a Transportation System

Steve Jeffrey Tueno Fotso
Université Paris Est Créteil, LACL, Créteil, France
Université de Sherbrooke, GRIL, Québec, Canada
steve.tuenofotso@univ-paris-est.fr

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

Marc Frappier
Université de Sherbrooke, GRIL, Québec, Canada
Marc.Frappier@usherbrooke.ca

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

Francois Thibodeau
Ville de Montréal, Québec, Canada
francoisthibodeau@ville.montreal.qc.ca

Mama Nsangou Mouchili
Ville de Montréal, Québec, Canada
Mama.NsangouMouchili@stantec.com

Keywords: Road Transportation System, Requirements Engineering, Formal Models, Domain Modeling, *SysML/KAOS*, *B System*, *Event-B*

Abstract

This paper describes a case study of the SysML/KAOS method for a road transportation system for the City of Montreal (VdM), the second-largest city in Canada. The transportation system was developed from unstructured requirements represented in textual and schematic documents. Therefore, the VdM wanted to investigate new ways of organising and analysing the requirements of traffic projects, in order to increase the level of confidence in their safety, usability and reusability. This paper describes the formal specification, verification and validation of system requirements and provides an appraisal of the SysML/KAOS requirements

engineering method on an industrial-scale case study. The specification is performed according to SysML/KAOS, a formal requirements engineering method developed in the *ANR FORMOSE* project for critical and complex systems. SysML/KAOS is designed to bridge the gap between stakeholder needs and the formal specification of system functionalities and domain constraints. The method has proven useful to deal with the seven refinement levels, twelve components (human, hardware, software and cyber-physical) and a hundred functional and non-functional goals that constitute the specification of the road transportation system, mainly focused on the safe movement of vehicles on roads. It especially facilitated their validation with VdM stakeholders who had never dealt with formal methods and requirements engineering. Animation tools (*ProB* and *B-Motion Studio*) were also used to validate the formal specification with VdM stakeholders. This paper also reports improvements identified to enhance the expressiveness of SysML/KAOS goal modeling languages during validation sessions with VdM stakeholders. This includes the introduction of (1) a way to quantify impacts and contributions of goals, (2) a non-functional goal refinement strategy based on logical formulas, (3) an approach to refine contribution goals, and (4) an obstacle modeling language. The improvements are planned to appear in future releases of supporting tools.

8.1 Introduction

SysML/KAOS is a requirements engineering method which aims to emphasize the impact of formal specification and verification activities on the quality of requirements, while taking into account the domain constraints and improving validation with stakeholders. The main interest is on critical and complex areas such as railway, aeronautics or road transportation. The method involves a functional [92] and a non-functional [68, 71] goal modeling languages to represent system requirements extracted from artifacts that describe stakeholder needs. The functional goal model represents system functionalities while the non-functional one represents constraints on their satisfaction. In addition, a domain modeling language [61, 65] is used to represent application domain entities and their properties. The system complexity is mastered in SysML/KAOS thanks to refinements and decompositions. In [102], Matoussi *et al.* have defined translation rules to produce a *B System* specification [41] from *SysML/KAOS* functional goal models. They provide the *behavioral part* (events) of the specification. Regarding domain models, rules have been defined and formally verified [63, 65] to generate the *structural part* (sets,

constant and their properties, variables and their invariant) of the specification and the initialisation of state variables. Once the event bodies manually specified, the *B System* specification can be formally verified and validated to assess the requirements. This can be done using the full range of tools that support the *B* method [10], positively assessed on a number of industrial projects for more than 25 years [93].

In 2014, *La Ville de Montréal* (VdM) proceeded to replace the *Bonaventure highway (A-10)* with an urban boulevard [47, 48]. As part of this reconfiguration, the *Québec Ministry of Transport (MTQ)* emphasized the importance of ensuring that the interventions carried out do not reduce the safety of road users. In addition, the VdM requires ensuring the functionality of the municipal road network. To allow the identified requirements to be taken into account, a number of additional options have been developed including (1) the addition of signaling equipments such as thermal imaging cameras and traffic control radars, and (2) the setting up of an intelligent transportation system that includes an automated incident detection system [82] provided by the MTQ. The transportation system was developed based on textual and schematic documents ([48] and its annexes, [82], etc.). Not only does this documentation not allow a clear identification of requirements, but it rarely shows the justification and validity of the choices made. Therefore, the VdM wanted to investigate a way of organising and analysing the requirements of traffic projects, in order to increase the level of confidence in their safety, usability, reusability and efficiency. This paper describes the formal specification, verification and validation of requirements of the transportation system and of the supervisor in charge of ensuring optimal operation of the involved components. SysML/KAOS was chosen because it includes an expressive and intuitive goal modeling language to represent system requirements, and a domain modeling language to represent application domain entities and their properties using ontologies. Furthermore, the rules required to generate a *B System* specification from goal and domain models are defined and the most relevant ones have been formally verified [65]. This paper also reports improvements identified to enhance the expressiveness of SysML/KAOS goal modeling languages and validated with VdM stakeholders. This includes the introduction of (1) a way to quantify the impact or contribution of a goal (a contribution goal is a satisficity solution to a non-functional requirement [38]), (2) a non-functional goal refinement strategy based on logical formulas, (3) an approach to refine contribution goals similar to that of Chung *et al.* [38], and (4) an obstacle modeling language such as the one proposed by Lamsweerde in [139].

The remainder of this paper is structured as follows: Section 2 briefly describes the *B System* formal method, the SysML/KAOS requirements engineering method and its goal and domain modeling languages, and the *B System* formalisation of SysML/KAOS models. Follows a presentation, in Section 3, of the work done on the case study. Section 4 discusses validation and verification of the formal specification and describes the relevant lessons learned from this case study. Finally, Section 5 reports our conclusion and future work.

8.2 Context

8.2.1 B System

Event-B [8] is an industrial-strength formal method for *system modeling*. It allows the incremental construction of system specifications, using stepwise refinement, and the proof of useful properties. *B System* is an *Event-B* syntactic variant proposed by *ClearSy*, an industrial partner in the *FORMOSE* project [17], and supported by *Atelier B* [41]. It shares the same semantics with *Event-B*.

A *B System* specification consists of components. Each component can be either a system or a refinement and it may define static or dynamic elements. A refinement is a component which refines another one in order to concretise the system construction: addition of functionalities or specification of the achievement of some purposes. Constants, abstract and enumerated sets (user-defined types), and their properties, constitute the static part. The dynamic part includes the representation of system state using variables constrained through invariants (first-order predicates that constrain the possible values that the variables may hold) and updated through events.

SYSTEM *S*
SETS *T*
CONSTANTS C
PROPERTIES *P*
VARIABLES *V*
INVARIANT *Inv*
EVENTS *E*
END

Each event has a ***guard*** *G* and an ***action*** *Act*. An event is said to be enabled when its guard *G* holds.

E ≙
SELECT *X*
WHERE *G*
THEN *Act*
END

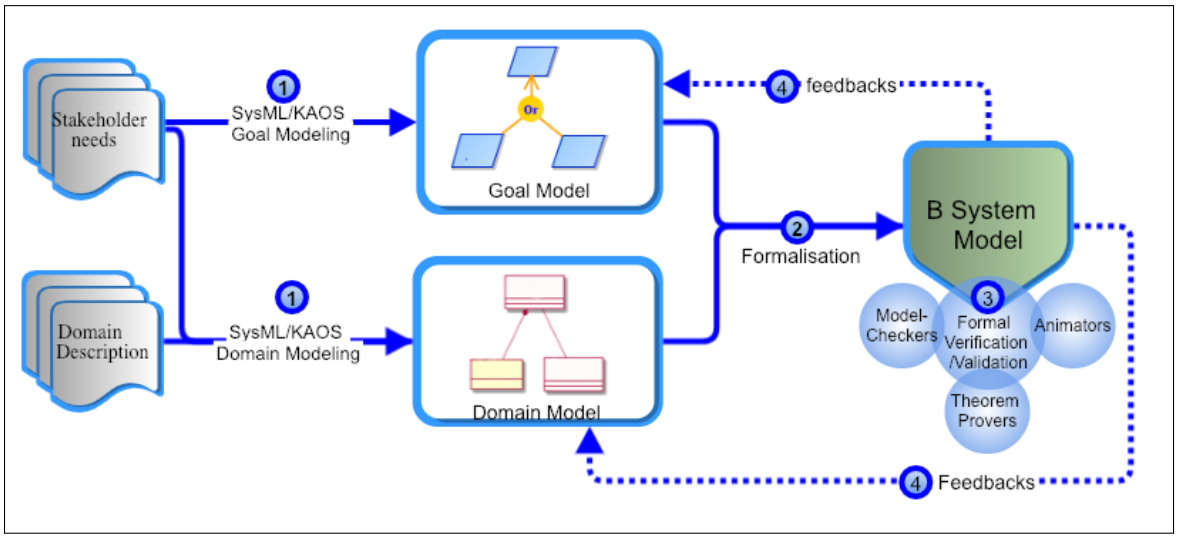


Figure 8.1 – Overview of the SysML/KAOS specification process [57]

A system transition consists in the triggering of a single event, among all enabled ones. Action *Act* of an event describes the updates made to state variables.

The triggering of an event should maintain the invariant *Inv*. To this aim, a proof obligation is generated for each event: $\forall T, C, X.\ (P \wedge G \wedge Inv \Rightarrow [Act]Inv)$. The expression $[Act]Inv$ denotes the weakest precondition such that the execution of *Act* terminates in *Inv*. Other proof obligations include *event feasibility* (existence, for each event, of a state where it can be triggered) and *system refinement* (the specification of a refinement conforms to that of the refined component) [8].

8.2.2 SysML/KAOS

SysML/KAOS is a requirements engineering method which defines a functional and non-functional goal modeling and a domain modeling languages. Figure 8.1 provides an overview of its specification process [57].

The first step is to use SysML/KAOS languages to build models of the system and of its application domain. The second step is to translate the goal model into a *B System* specification, following the rules provided in [59, 102], and to complete the specification with the result of the translation of domain models, following the formally verified rules provided in [65, 132]. Goal models provide the **behavioral part** (events) of the specification while domain models provide its **structural part** (sets, constant and their properties, variables and their invariant) and the initialisation of state variables. It remains to manually specify the body of

events and to formally verify and validate the specification with *B System* tools. When updates are performed within the *B System* specification, back propagation rules such as those described in [57] are used to update SysML/KAOS models accordingly.

SysML/KAOS is supported by integrated development environments *Openflexo* [109] and *Atelier B* [41]. Openflexo supports goal and domain modeling while Atelier B supports the specification, verification and validation of *B System* models. These last activities can also be carried out under Rodin [35] since Event-B and *B System* share the same semantics.

SysML/KAOS Functional Goal Modeling

The SysML/KAOS functional goal modeling language [92] combines the traceability provided by *SysML* [78] with goal expressiveness provided by KAOS [138]. It allows the representation of functional requirements to be satisfied by a system and of functional expectations with regards to the environment through a hierarchy of goals. A functional goal in SysML/KAOS describes the expected behaviour of the system once a certain condition holds: *[if* ***CurrentCondition*** *then] sooner-or-later* ***TargetCondition***. A functional goal can also be defined without specifying a current condition. In this case, the expected behaviour can be observed from any system state. The functional 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. The refinement process ends when it is possible to assign the leaf goals to a subsystem or to an agent (environment agent or software agent). Subsequently, if needed, further goal diagrams can be defined for the different subsystems.

SysML/KAOS Non-Functional Goal Modeling

Non-functional goals are represented in SysML/KAOS using a language similar to that of functional goals [68, 71] and which borrows concepts from the *NFR Framework* [38]. As with functional goals, the non-functional goal hierarchy is built through a succession of refinements using operators ***AND*** and ***OR***. However, the non-functional goal hierarchy is built in a model different from the one that structures the functional goals. Each non-functional goal is represented as ***NFRType[Topic]*** where ***NFRType*** identifies the constraint type (security, safety, etc.) and ***Topic*** identifies the system entity that the constraint targets. A goal ***NFRType[Topic]*** can be refined either by $\boldsymbol{NFRType_i}$***[Topic]*** (*refinement by type*) or by ***NFRType[***$\boldsymbol{Topic_i}$***]*** (*refinement by topic*),

knowing that ***NFRType***$_i$ is a subtype of ***NFRType*** and ***Topic***$_i$ is a subentity of ***Topic***. For instance, a non-functional goal ***Security[System]*** can be refined by subgoals ***Confidentiality[System]***, ***Integrity[System]*** and ***Availability[System]*** according to the taxonomy of non-functional goal types provided in [38] (refinement by type). A refinement by topic of goal ***Security[System]*** gives subgoals ***Security[Hardware]*** and ***Security[Software]*** for a system consisting of a hardware and a software. The refinement process ends when it is possible to provide satisficity solutions to leaf goals called *contribution goals*.

Each contribution goal can contribute *positively* (+) or *negatively* (-) to the satisfaction of a non-functional goal. Similarly, each contribution goal can have a *positive* (+) or *negative* (-) impact on the achievement of a functional goal. Impacts are represented in a distinct model called the *integrated model* [68]. They can (1) constrain the refinement of functional goals, (2) lead to the definition of new functional goals, or (3) constrain the way some leaf functional goals are achieved by agents to which they are assigned.

SysML/KAOS Domain Modeling

Domain models in SysML/KAOS are represented using ontologies. These ontologies are expressed using the SysML/KAOS domain modeling language [61, 132], based on *OWL* [118] and *PLIB* [112], two well-known and complementary ontology modeling formalisms. Each domain model corresponds to a refinement level in the functional goal model. Domain models can be linked together to form a hierarchy. A domain model can define multiple elements. *Concepts* designate collections of *individuals* with common properties. A concept can be declared *variable* when the set of its individuals can be updated by adding or deleting individuals. Otherwise, it is considered to be *constant*. In addition, a concept can be an enumeration if all its individuals are defined within the domain model. An individual can be *variable* if it is introduced to represent a system state variable: it can represent different individuals at different system states. Otherwise, it is *constant*. *Associations* are concepts used to capture links between concepts. *Maplet individuals* capture associations between individuals through associations. The variability of an association is related to the ability to add or remove maplets. *Logical formulas* are used to represent constraints between different elements of the domain model in the form of *Horn clauses*. *Gluing invariants* are logical formulas used to represent links between data defined within a domain model and those appearing in more abstract domain models. They capture relationships between abstract and concrete data during refinement and are used to discharge proof obligations.

8.2.3 B System Formalisation of SysML/KAOS Models

The formalisation of SysML/KAOS functional goal models is detailed in [102]. The proposed rules allow the generation of a formal model whose structure reflects the hierarchy of the functional goal model: one component is associated with each level of the goal hierarchy; this component defines one event for each goal. As the semantics of the refinement between goals is different from that of the refinement between *B System* components, new proof obligations for goal refinement are defined in [102]. They depend on the goal refinement operator used and complete the *B System* proof obligations for invariant preservation and for event feasibility. For instance, the following proof obligations formalise the AND refinement of an abstract goal G into two concrete goals G_1 and G_2 (for an event `G`, `G_Guard` represents the guards of G and `G_Post` represents the post condition of its actions):

- $G_1\_Guard \Rightarrow G\_Guard$
- $G_2\_Guard \Rightarrow G\_Guard$
- $(G_1\_Post \wedge G_2\_Post) \Rightarrow G\_Post$

It should be noted that variables updated by subgoals must be distinct.

Nevertheless, the generated *B System* specification does not contain the system structure, that are variables with their associated invariant and constants with their associated properties. This structure is provided by the translation of SysML/KAOS domain models. The corresponding translation rules are fully described in annex B. In short, domain models identify *B System* components. Concepts give *B System* types while individuals give set items. Logical formulas give *B System* properties and invariants. The rules also allow the extraction of the initialisation of state variables.

8.3 Specification of the Road Transportation System

8.3.1 Main Characteristics of the System

The VdM needs to proceed with the replacement of the Bonaventure highway (A-10) with an urban boulevard while ensuring that the interventions carried out do not reduce the safety of road users (MTQ) and that the municipal road traffic is at least maintained (VdM) [48]. Regarding the *Nazareth* street and especially the exit of the *Ville-Marie* highway to Nazareth street, it was difficult to respond to both the issues identified by the VdM and the safety issue formulated by the MTQ, especially because of the curvature of the highway exit (see [47]). Indeed, the accumulation of vehicles at the highway exit is likely to cause accidents because the curvature limits the line of sight of drivers that engage on the exit when they are at the upstream of the curvature. It is thus necessary (i) to determine the level of traffic at every

moment, (ii) to regulate the traffic level in order to limit the exit congestion in reasonable proportions, and (iii) to notify drivers, especially those located at the upstream of the curvature, as to the level of the traffic and the expected behavior. The VdM has therefore decided the addition of: (1) two travel lanes for the Ville-Marie highway exit to Nazareth street to the three lanes of Nazareth street (see [48]) and (2) sensors such as thermal imaging cameras and traffic control radars to ensure the determination of the level of traffic. Traffic regulation consists in defining the most appropriate traffic signal program, taking into account the level of traffic. It is performed by an automaton connected to VdM sensors. An urban mobility management center (CGMU) has been set up by the VdM to ensure that the level of traffic is properly regulated (traffic level supervision) and notify drivers (level of traffic and expected behavior). To ensure the satisfaction of its safety requirement, the MTQ has also set up a mobility management center (CIGC) and an intelligent transportation system that includes an automated incident detection system (AID). The AID is connected to the CGMU and provides a more accurate measurement of the level of traffic that helps to validate the inputs from VdM sensors. It uses thermal cameras and a software to analyse the traffic in real-time and detect road incidents. As the CGMU, the CIGC is responsible for sending some notifications to drivers through variable message signs (PMVs) or through GPS navigation softwares such as *Waze* or *Google Maps*.

The SysML/KAOS method is used to provide a framework for the specification, verification and validation of requirements of the integrated components and of the supervisor responsible for ensuring the optimal operation of these components.

8.3.2 Functional Goal and Obstacle Modeling

Functional Goal Modeling

Figure 8.2 [2] provides an overview of the goal diagram that represents the functionalities of the high-level system. The main identified purpose is *to allow each vehicle on the Ville-Marie highway exit that connects to Nazareth street to exit*. The purpose gives the most abstract goal *BringOutEachVehiclePresentInTunnel* of the goal diagram which is refined using the ***AND*** operator into two subgoals: drive vehicle according to road signing (goal *MoveVehicle*) and manage congestion (goal *ManageCongestion*). The leaf goal *MoveVehicle* is assigned to *environment agent VehicleDriver* (the vehicle driver) to state the assumption that the driver has the responsibility to drive its vehicle according to road signs. The assumptions are expressed in domain models as domain constraints. For instance, the previous assumption entails that *"each vehicle speed does not exceed the speed limit"*. For congestion management, it is necessary to be able to: (1) determine the traffic level from sensors (goal *DetermineTrafficLevel*),

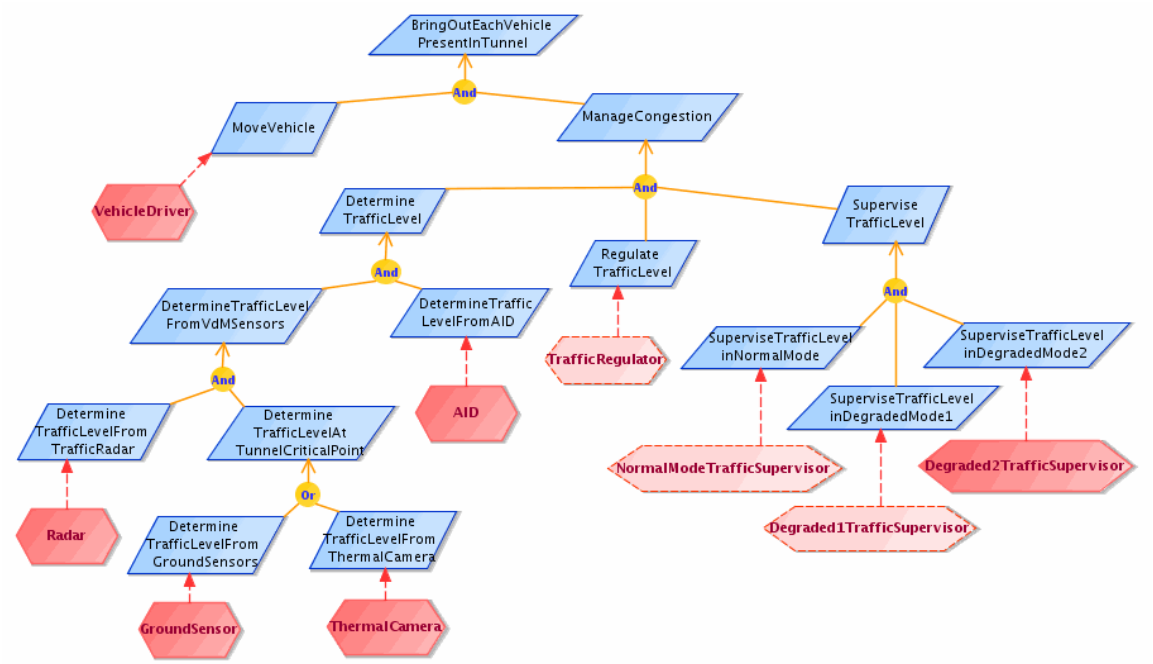
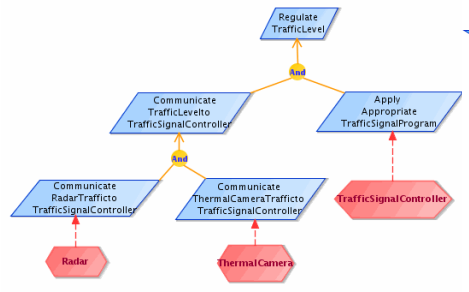


Figure 8.2 – High-level system functional goal diagram [2]

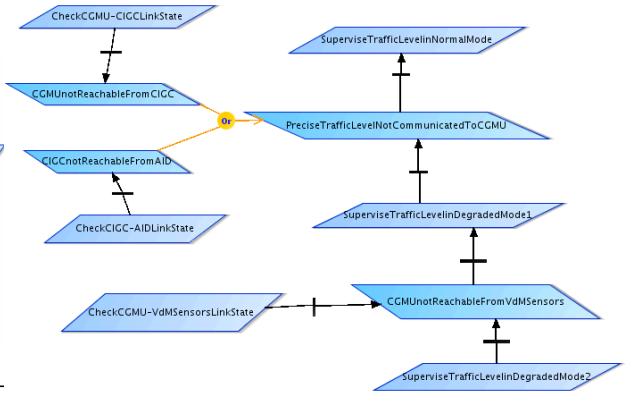
(2) regulate the traffic (goal *RegulateTrafficLevel*), and (3) supervise traffic regulation and, if necessary, adjust the traffic signal program defined by the traffic signal controller (goal *SuperviseTrafficLevel*). The goal *RegulateTrafficLevel* is assigned to the *TrafficRegulator subsystem* for which the functionalities are represented by the goal diagram of Figure 8.3a [2]: determine the level of traffic from measurements of VdM sensors (goal *CommunicateTrafficLeveltoTrafficSignalController*) and define the most appropriate traffic signal program (goal *ApplyAppropriateTrafficSignalProgram*).

Since the level of traffic is determined using VdM sensors and the MTQ's AID, goal *DetermineTrafficLevel* is AND-refined into subgoals *DetermineTrafficLevel-FromVdMSensors*, for VdM sensors, and *DetermineTrafficLevelFromAID*, for the MTQ's AID. The VdM sensors include a traffic control radar and a redundant sensor. Indeed, the highway exit is splitted into four zones, until the point where the last vehicle should be in case of maximum congestion lengthening (*Xmax*). The radar covers the four zones. However, a redundant sensor (ground sensor or thermal camera) is needed for the fourth zone (the one that ends at *Xmax*) to ensure that the maximum congestion lengthening will be detected even in case of a radar failure.



Functional goal diagram of the *TrafficRegulator* subsystem [2]

(a)



Obstacle model related to the unreliability of links to CGMU [3]

(b)

Knowing that the communication links from CGMU to VdM sensors and from CIGC to CGMU are subject to failure, an obstacle analysis was carried out based on the obstacle modeling language of KAOS [139].

Obstacle Modeling

An obstacle is an obstruction to the satisfaction of a functional goal. Obstacle modeling allows analysis of expected system behaviors when obstacles prevent the satisfaction of one or more functional goals [139]. Obstacles can be refined to specify their causes: an obstacle can be caused by a conjunction or disjunction of more specific ones. New functional goals or countermeasures can therefore be defined to prevent, detect or mitigate obstacles, thus ensuring adequate behavior of the system.

Figure 8.3b [3] illustrates the obstacle modeling, related to the unreliability of CGMU to VdM sensors and CIGC to CGMU links, that entailed the definition of the three supervision modes of Figure 8.2 (goals *SuperviseTrafficLevelinNormalMode*, *SuperviseTrafficLevelinDegradedMode1* and *SuperviseTrafficLevelinDegradedMode2*). Each black arrow goes from an introduced element (functional goal or obstacle) to the element that entails it.

When all is well, the supervision is performed in *normal mode* (goal *SuperviseTrafficLevelinNormalMode* refined in another goal diagram [2]): each management center (CGMU and CIGC) receives traffic data from its sensors and notifies the other as to its traffic knowledge. Since AID measurements are more accurate, in normal mode, they will be systematically used by CIGC and CGMU to undertake supervision actions: ensure the appropriateness of the traffic signal program and ensure the appropriateness of user notifications.

The normal mode traffic supervision may be obstructed by the impossibility for AID to send a precise traffic measurement to CGMU (obstacle *PreciseTrafficLevelNotCommunicatedToCGMU* of Figure 8.3b). This can be due to the unavailability of the communication channel between the CGMU and the CIGC (obstacle *CGMUnotReachableFromCIGC*) or by that of the one between AID and CIGC (obstacle *CIGCnotReachableFromAID*). A countermeasure to detect the occurrence of obstacle *CGMUnotReachableFromCIGC* is to regularly check the state of the communication channel between the CGMU and the CIGC (goal *CheckCGMU-CIGCLinkState*). Similarly, goal *CheckCIGC-AIDLinkState* is proposed as countermeasure to obstacle *CIGCnotReachableFromAID*. The functional goal *SuperviseTrafficLevelinDegradedMode1* (Figures 8.2 and 8.3b) allows the supervision to be performed properly despite an occurrence of obstacle *PreciseTrafficLevelNotCommunicatedToCGMU*, by defining an alternative that allows the CGMU to perform the supervision without the need of the CIGC: only VdM sensors are considered to determine the level of traffic. However, an obstacle to the satisfaction of goal *SuperviseTrafficLevelinDegradedMode1* is *CGMUnotReachableFromVdMSensors*, related to the impossibility for CGMU to obtain measurements from VdM sensors. A detection countermeasure therefore consists in regularly probing the state of the communication channel between CGMU and VdM sensors (goal *CheckCGMU-VdMSensorsLinkState*). An additional goal *SuperviseTrafficLevelinDegradedMode2* (Figures 8.2 and 8.3b) is defined as a mitigation countermeasure and consists in sending a human agent for local traffic supervision.

8.3.3 Non-Functional Goal Modeling

Figure 8.3 [3] provides an overview of an integrated goal diagram that represents the main non-functional goals of the high-level system and some relevant contribution goals (in green) with their identified contributions (black arrows) and impacts (red arrows). The main non-functional goal that has been identified is to ensure a high quality of service to the entire system. To ensure a high quality of service, it is necessary to ensure high safety, security and performance [38]. This gives the goals of the first refinement level which are derived from a refinement by type of the root goal. Similarly, to ensure a high performance of the system, it is

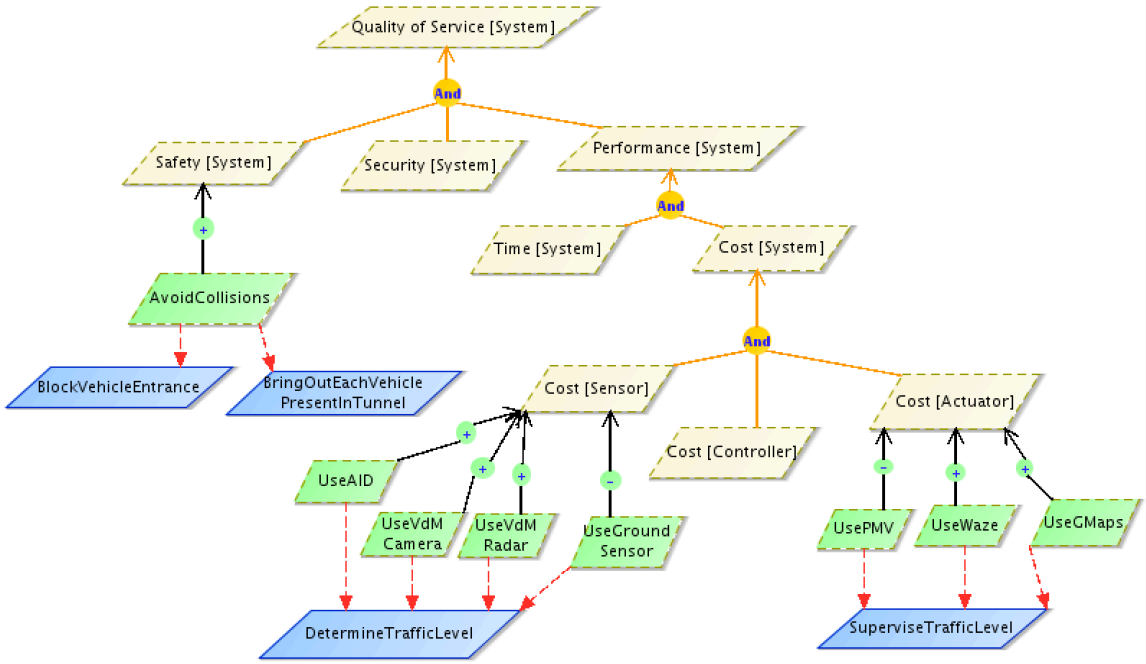


Figure 8.3 – High-level system integrated goal diagram [3]

necessary to minimise operating times and costs. Non-functional goals of the third refinement level, derived from a refinement by topic of goal *Cost [System]*, express that system operating costs are distributed between operating costs of sensors (such as thermal cameras), actuators (such as variable message signs) and controllers (such as the traffic signal controller).

A way to ensure the system safety (non-functional goal *Safety [System]*) is to avoid collisions between vehicles due to the curvature of the highway exit (contribution goal *AvoidCollisions*). Thus, *AvoidCollisions* contributes positively to *Safety [System]*. In addition, *AvoidCollisions* has a significant impact on the satisfaction of functional goal *BringOutEachVehiclePresentInTunnel* and requires addition of a functional goal *BlockVehicleEntrance* to prevent engagement of vehicles on the highway exit. This new functional goal is required to ensure that the congestion can always be regulated and was found during non-functional goal modeling.

Ways to mimise operating costs of sensors include the use of the MTQ's AID (goal *UseAID*), of the VdM's thermal camera (goal *UseVdMCamera*) and of the VdM's traffic radar (goal *UseVdMRadar*). The use of a ground sensor (goal *UseGroundSensor*) does not contribute to the minimisation of operating costs of sensors because ground sensors are hardly maintainable and reusable since they are underground.

Therefore, the redundant VdM sensor is a thermal camera: only goal *DetermineTrafficLevelFromThermalCamera* (Fig. 8.2) is retained in the OR decomposition of goal *DetermineTrafficLevelAtTunnelCriticalPoint*. Similarly, the use of GPS navigation softwares such as *Waze* or *Google Maps* contributes positively to the minimisation of operating costs of actuators. This is not the case for variable message signs (goal *UsePMV*) because their use requires their purchase and maintenance.

A non-functional goal diagram was built specifically for security requirements. It is not presented in this paper for space limitations.

8.3.4 Domain Modeling

Six domain models were constructed for the six refinement levels of the functional goal model [1]. For space limitations, we will focus only on the first two.

Root Level

Figure 8.4 [1] represents the domain model associated with the root goal *BringOutEachVehiclePresentInTunnel* of the diagram of Figure 8.2. The domain model introduces the entities required to represent the exit of the Ville-Marie highway to Nazareth street and to localise vehicles. Its aim is to enable the specification of vehicle exits. Therefore, a concept `VEHICLE` is defined to represent all vehicles likely to engage on the highway exit. Association `Vehicle_Length` captures the length of each vehicle as a natural number. A variable concept named `Vehicle` is defined as a subconcept of `VEHICLE` to represent the vehicles currently engaged on the highway exit. Its cardinality is used to quantify the level of traffic [82]. Each vehicle engaged on the highway exit is localised by the position of its front (variable association `Vehicle_Front_Position`) and by its travel lane (variable association `Vehicle_Travel_Lane`). Indeed, the highway exit has two travel lanes (see [47]): a main one represented by individual `TRAVEL_LANE_I` and a secondary one, represented by `TRAVEL_LANE_II`, which appears when the vehicle gets closer to the Nazareth street.

Logical formulas are defined to represent properties that need to be guaranteed in all system states. A logical formula can be defined to enforce (or represent) a contribution goal (non-functional goal model). For instance, the logical formula below ensures that the locations occupied by two distinct vehicles are always distinct (absence of collisions [99]):

$$\begin{aligned}&\forall xx1, xx2 \cdot ((xx1 \in Vehicle \wedge xx2 \in Vehicle \wedge xx1 \neq xx2\\ &\wedge Vehicle\_Travel\_Lane(xx1) = Vehicle\_Travel\_Lane(xx2))\\ &\Rightarrow((Vehicle\_Front\_Position(xx1) - Vehicle\_Length(xx1)) \mathinner{..} Vehicle\_Front\_Position(xx1)\\ &\cap(Vehicle\_Front\_Position(xx2) - Vehicle\_Length(xx2)) \mathinner{..} Vehicle\_Front\_Position(xx2) = \emptyset))\end{aligned}$$

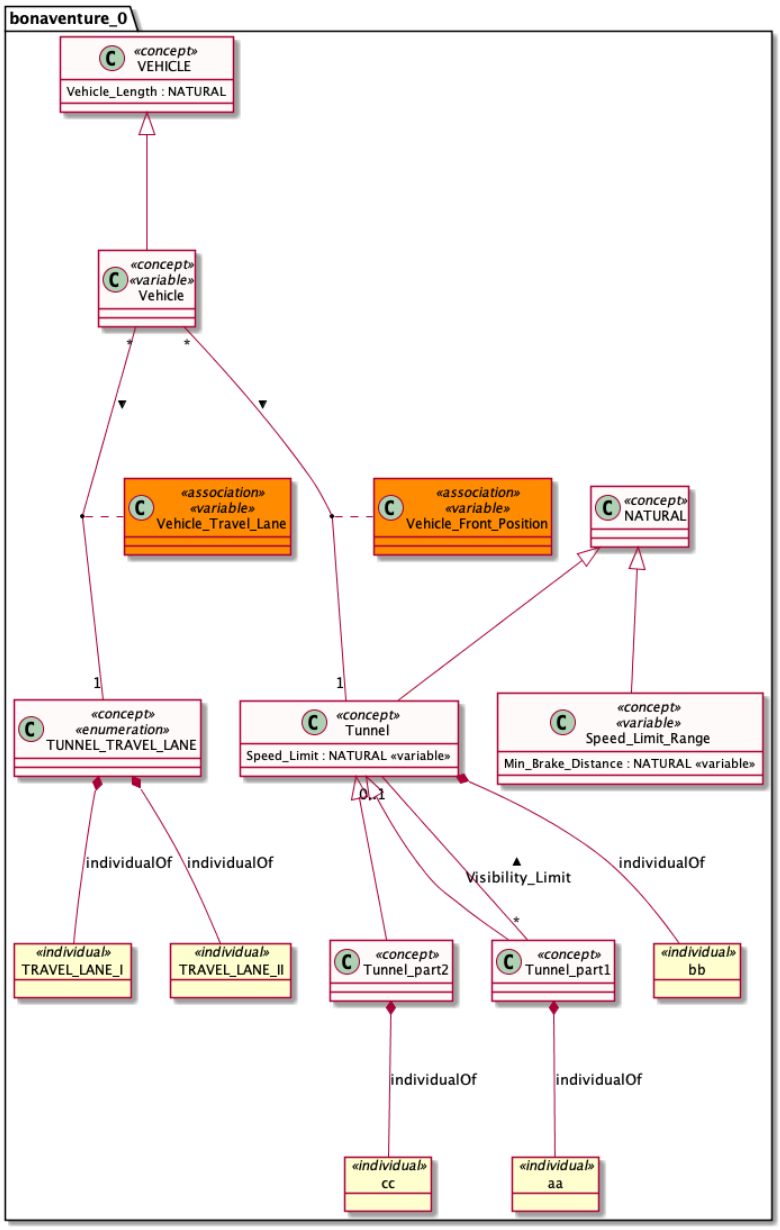


Figure 8.4 – Ontology associated with the root level of the goal diagram of Fig. 8.2 [1]

The highway exit is represented by a concept `Tunnel` defined as a range of integers (`Tunnel` = `aa` .. `cc`). Association `Speed_Limit` captures the speed limit (in *KM/H*) defined at each position of the highway exit. It is variable because the speed limit is likely to be updated depending on traffic level. Concept `Tunnel_part1` is the subpart of the highway exit that contains the curvature which limits the visibility of upstream vehicles (`Tunnel_part1` = `aa` .. `bb`, `bb` < `cc`). Therefore, an association named `Visibility_Limit` is used to associate a visibility limit to parts of `Tunnel_part1`: each user whose vehicle A has its front located at $xx \in$ `Tunnel` is supposed to be able to see vehicle B in front of him (and consequently to act in a way to avoid a collision) unless $xx \in dom($`Visibility_Limit`$)$ and the rear of vehicle B is located beyond `Visibility_Limit`(xx). Finally, association *Min_Brake_Distance* sets a minimum braking distance for each speed defined as speed limit. Therefore, it is necessary to ensure that for each speed limit defined for a location xx, if a visibility limit is applicable at xx ($xx \in dom($`Visibility_Limit`$)$), the speed limit is defined such that the minimum braking distance is less than the distance between xx and `Visibility_Limit`(xx):

$$\forall xx \cdot (xx \in dom(Visibility\_Limit) \Rightarrow Visibility\_Limit(xx) > xx)$$

First Refinement Level

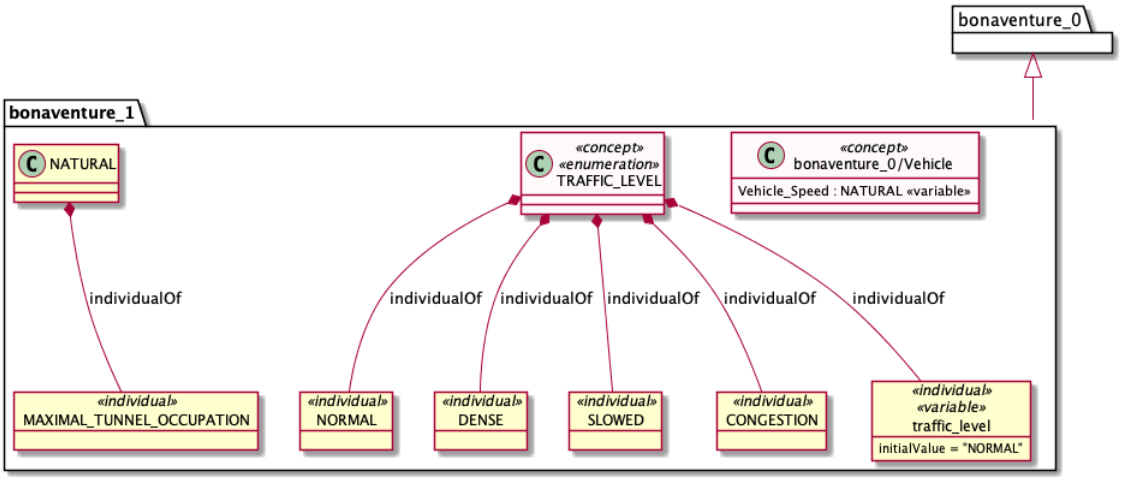


Figure 8.5 – Ontology associated with the first refinement level of the goal diagram of Fig. 8.2 [1]

Figure 8.5 [1] represents the domain model associated with the first refinement level of the goal diagram of Figure 8.2. It refines the one associated with the root level (Figure 8.4) and introduces the entities required to represent the traffic level which depends on vehicle speeds and locations [82]. A natural number (individual `MAXIMAL_TUNNEL_OCCUPATION`) is defined to represent the maximum number of vehicles allowed at the highway exit and a variable association `Vehicle_Speed` is defined to represent speeds of vehicles. We assume that the vehicles are driven according to road signing. The assumption is represented by a logical formula stating that the speed of any vehicle must always be lower than the speed limit associated with its location:

$\forall xx \cdot (xx \in Vehicle \Rightarrow Vehicle\_Speed(xx) \leq Speed\_Limit(Vehicle\_Front\_Position(xx)))$

Four traffic levels are considered: *normal*, *dense*, *slowed* and *congestion* [82]. The variable individual `traffic_level` is defined to represent the current known traffic level. Each traffic level is defined by an individual and a logical formula that specifies its requirements. For instance, the traffic level is normal when the highway exit is occupied at 40% or less and vehicle speeds are higher than 40 KM/H [82]:

$(traffic\_level = NORMAL$
$\Rightarrow (((card(Vehicle) * 100)/MAXIMAL\_TUNNEL\_OCCUPATION) < 40$
$\wedge (\forall xx \cdot (xx \in Vehicle \Rightarrow Vehicle\_Speed(xx) \geq 40))))$

The domain model associated with the second refinement level of the goal model introduces the entities required to distinguish between environment variables, which represent the actual state of the real environment and controller variables, which represent the measured value of the environment, as seen by the controller (measured vehicle front positions, measured vehicle speeds, etc.). This distinction is necessary to handle measurement errors and control delays [111]. The next domain model introduces the traffic level sensors and supervision modes (normal and degraded). It also introduces traffic lights and signaling programs to allow the specification of traffic regulation. Finally, the fifth and sixth domain models introduces the communication channels, from sensors to management centers (CGMU and CIGC) and between management centers, to allow the specification of traffic supervisions.

8.3.5 The B System Specification

The full specification, verified using the *Rodin* platform [35], can be found in [4]. Each refinement level is the result of the translation of goal and domain models, except the body of events that are provided manually. For instance, the root level of the goal diagram of Fig. 8.2 gives the *B System* event *BringOutEachVehiclePresentInTunnel* specified in the root machine as:

Event *BringOutEachVehiclePresentInTunnel* $\widehat{=}$

SELECT Vehicle_Out, Vehicle_In, newVehicleFronts, newTravelLanes
WHERE
grd0: $Vehicle \neq \emptyset$
grd1: $partition(Vehicle, Vehicle\_Out, Vehicle\_In)$
grd2: $newVehicleFronts \in Vehicle\_In \rightarrow Tunnel$
grd3: $newTravelLanes \in Vehicle\_In \rightarrow TUNNEL\_TRAVEL\_LANE$
grd4: $\forall xx \cdot ((xx \in Vehicle\_In \wedge newVehicleFronts(xx) \in Tunnel\_part1)$
$\Rightarrow newTravelLanes(xx) = TRAVEL\_LANE\_I)$
grd5: $\forall xx1, xx2 \cdot ((xx1 \in Vehicle\_In \wedge xx2 \in Vehicle\_In \wedge xx1 \neq xx2)$
$\Rightarrow ((newVehicleFronts(xx1) - Vehicle\_Length(xx1)) \mathinner{..} newVehicleFronts(xx1)$
$\cap (newVehicleFronts(xx2) - Vehicle\_Length(xx2)) \mathinner{..} newVehicleFronts(xx2) = \emptyset$
$\vee newTravelLanes(xx1) \neq newTravelLanes(xx2)))$
THEN
act0: $Vehicle := Vehicle \setminus Vehicle\_Out$
act1: $Vehicle\_Front\_Position := newVehicleFronts$
act2: $Vehicle\_Travel\_Lane := newTravelLanes$
END

This event states that when vehicles are present on the highway exit (grd0), we observe some exiting (act0) and others moving, by nondeterministically changing their traffic lanes (grd3 and act2) and front positions (grd2 and act1), while ensuring the preservation of safety invariants (grd4 and grd5). Guard grd1 ensures that each vehicle ($x \in Vehicle$) either exits ($x \in Vehicle\_Out$) or moves ($x \in Vehicle\_In$). In the first refinement level of the *B System* specification, event *BringOutEachVehiclePresentInTunnel* is refined by events *ManageCongestion* and *MoveVehicle*, the last being specified as [1]:

Event *MoveVehicle* $\widehat{=}$
SELECT newTravelLanes, updatedVehicleFronts, newVehicleSpeeds, Vehicle_Out, Vehicle_In, trafficLevel, newVehicleFronts
WHERE
grd0: $delay \in \mathbb{N}_1$
grd1: $Vehicle \neq \emptyset$
grd2: $updatedVehicleFronts = (\lambda xx \cdot xx \in Vehicle | Vehicle\_Front\_Position(xx) + Vehicle\_Speed(xx) * delay)$
grd3: $Vehicle\_In = updatedVehicleFronts^{-1}[Tunnel]$
grd4: $Vehicle\_Out = Vehicle \setminus Vehicle\_In$
grd5: $newVehicleSpeeds \in Vehicle\_In \rightarrow \mathbb{N}$
grd6: $\forall xx \cdot (xx \in Vehicle\_In \Rightarrow newVehicleSpeeds(xx) \in 0 \mathinner{..} Speed\_Limit(updatedVehicleFronts(xx)))$
grd7: $newTravelLanes \in Vehicle\_In \rightarrow TUNNEL\_TRAVEL\_LANE$
grd8: $newVehicleFronts = Vehicle\_Out \lhd updatedVehicleFronts$
grd9: $trafficLevel \in TRAFFIC\_LEVEL$
grd10: $(trafficLevel = NORMAL \Rightarrow (((card(Vehicle\_In) * 100)/MAXIMAL\_TUNNEL\_OCCUPATION) < 40$
$\wedge (\forall xx \cdot (xx \in Vehicle\_In \Rightarrow newVehicleSpeeds(xx) \geq 40))))$
•••
THEN
•••
act3: $traffic\_level := trafficLevel$
act4: $Vehicle\_Speed := newVehicleSpeeds$
END

1. Event specification restricted to show only the most relevant part with respect to the one of event *BringOutEachVehiclePresentInTunnel*. The full version can be found in [4].

It states that after a certain delay *delay* (grd0), all vehicles present on the highway exit move a distance corresponding to the product of their speed by *delay* (grd2). Exiting vehicles (*Vehicle_Out*) are those that are driven out of the highway by their displacement (grd4). The others (*Vehicle_In*: vehicles that remain in the highway after their displacement (grd3)) nondeterministically change their speed (grd5, grd6 and act4) and lane (grd7) while ensuring the preservation of safety invariants. Finally, the traffic level is updated (act3) to reflect the new system state (grd9, grd10, ...).

8.4 Discussion

8.4.1 Validation and Verification

The SysML/KAOS method not only makes it possible to verify the consistency of requirements and their refinement logic, but also to better present and validate the requirements with the various stakeholders. Indeed, SysML/KAOS includes semi-formal languages for a high-level representation of system goals and application domain properties. This ensures a better reusability and readability of models. Improved readability is confirmed by VdM stakeholders who were involved to assess each modeling deliverable during scheduled validation sessions: four validation sessions were organised and allowed to introduce SysML/KAOS to VdM stakeholders and to obtain their feedback related to the constructed SysML/KAOS models. The improved readability was also confirmed after an evaluation was conducted among members of the FORMOSE project, within the framework of another case study [58]. Of the fifteen or so surveyed members representing various academic [2] and industrial [3] partners, all found the readability of SysML/KAOS models much better than that of a *B System* specification.

The method also includes rules for obtaining a *B System* specification and the proof obligations required to guarantee consistency of goal refinements and accuracy of requirements with respect to environment constraints. For instance, proof obligations related to SysML/KAOS refinements allowed us to identify a missing goal in goal diagrams. Indeed, the first version of the goal diagram of Fig. 8.3a was not defining a goal to ensure that vehicles are driven according to road signs. Therefore, it was impossible to ensure that a vehicle in the tunnel would be driven out. Thus, trying to formally ensure root goal satisfaction allowed us to introduce the *MoveVehicle* goal assigned to agent *VehicleDriver*.

2. *University Paris Est Créteil; University of Sherbrooke; IMT Brest, France; etc.*
3. *THALES, France; ClearSy Systems Engineering, France; Openflexo, France*

Using SysML/KAOS, we have methodically built the formal refinement hierarchy and we have determined and formally expressed the safety invariants. The method bridges the gap between the system textual description and its *B System* specification. Table B.1 summarises the key characteristics related to the formal specification of the first four refinement levels. The proof obligations have been discharged using the Rodin tool extended with *Atelier B provers* [115] and *SMT solvers* [127]. The interactive proof was more required for level **L3** because of the introduction of a distinction between the real and measured (by traffic sensors) views of traffic level. Indeed, this introduction required several adaptations and additions, of invariants and events, related for example to order in measurement acquisitions (enforced using controlled variables), sensor coverages and measurement defects (handled with degraded modes).

Table 8.1 – Key characteristics related to the formal specification

Refinement level	L0	L1	L2	L3
Invariants	**8**	**8**	**14**	**26**
Proof Obligations (PO)	**21**	**52**	**36**	**85**
Automatically Discharged POs	**19**	**51**	**36**	**66**
Interactively Discharged POs	**2**	**1**	**0**	**19**

Mashkoor *et al.* [99, 100] advocate the use of animation, supported by tools, to assist validation of a formal specification with non-expert stakeholders. *ProB* [95] and *B-Motion Studio* [90] are industrial-strength tools used to animate and validate a *B System* specification. They provide a way to define a high-level graphical representation of the states of the system. We used them to validate the formal specification with VdM stakeholders, in addition to graphical models constructed using the SysML/KAOS goal and domain modeling languages.

The validation by animation was performed following the *VTA (Verify-Transform-Animate)* framework [100]. The SysML/KAOS functional goal model provides the way to group requirements into observation levels (each observation level corresponds to a refinement level) as required by the VTA. The specification obtained from SysML/KAOS models, once completed with event bodies, has been verified with Rodin provers, transformed and animated. The formal model transformation has for instance consisted in (1) transforming abstract sets into concrete ones such as $VEHICLE$ in $\{V1, V2, V3, V4, V5\}$ and $Tunnel$ in $0\,..\,30$, and (2) introducing events to specify changes in environment structure such as *ctrl_ChangeSpeed* used to change a vehicle speed during animation. In addition, units were converted (*KM* to *M* for distances, hours (*H*) to seconds (*S*) for times, *KM/H* to *M/S* for speeds) to precisely observe the system behavior. The transformed model can be found in [5].

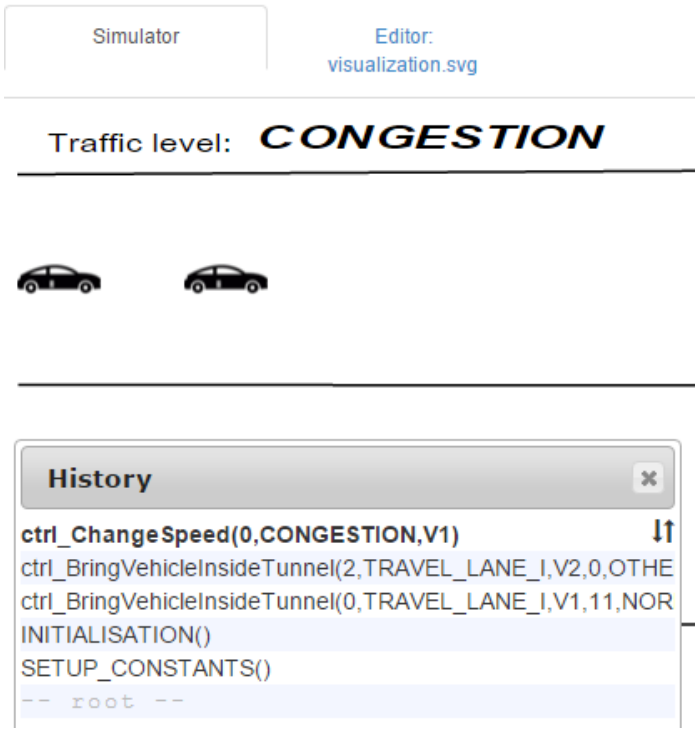


Figure 8.6 – Overview of a validation session performed using *ProB* and *B-Motion Studio*

For example, Figure 8.6 is an overview of a validation session with VdM stakeholders performed using ProB and B-Motion Studio. The top view presents an illustration of the traffic state on the highway exit while the bottom view presents a history of events triggered to reach this state. The maximum number of vehicles allowed is set to 4 and vehicles are not moving (speeds are set to 0). Therefore, the traffic level is congestion (highway exit occupied at 40% or more and vehicle speeds less than 15 KM/H [82]).

The formal validation allowed us to detect inconsistencies in textual documents that describe the road transportation system. For instance, we have detected that the four defined traffic levels were not sufficient [82]: *normal* (highway exit occupation is lower than 40% and vehicle speeds are greater than 40 KM/H), *dense* (occupancy lower than 40% and vehicle speeds between 35 and 39 KM/H), *slowed* (occupancy greater than 40% and vehicle speeds between 25 and 34 KM/H) and *congestion* (occupancy greater than 40% and vehicle speeds lower than 15 KM/H). Indeed, the

ProB model checker has determined traffic states that do not correspond to any of the defined traffic levels. This is for instance the case when occupancy is exactly 40% or when the speeds are between 15 and 24 KM/H. The observations, validated with VdM stakeholders, were reported to document authors from VdM and MTQ.

8.4.2 Lessons Learned, Improvements and Related Work

The SysML/KAOS method makes it possible to correctly model the functional and non-functional goals and to analyse the various ways of satisfying them in order to justify the choices made.

It took three months (September-December, 2018: 16 hours per week) to formally specify, verify and validate requirements of the VdM's road transportation system with SysML/KAOS. The development team was composed of six members (the authors of this paper). Four are academia stakeholders with good expertise in the formal specification of complex systems while the others are VdM stakeholders with expertise neither in requirements engineering nor in formal methods. The specification of the body of formal events and logical formulas and the formal assessment (verification and validation) of the specification can only be manual and therefore required time, in addition to experts in formal methods. But this is the price to pay to achieve a formal verification and validation of requirements.

From the textual description of the road transportation system [47,48] and of the AID [82], seven goal model refinement levels with a hundred functional and non-functional goals were defined [2,3]. This allowed us to specify and ensure consistency of the high level requirements of twelve components: humans, hardware (like radar or thermal camera), software (like the traffic supervisor) and cyber-physical systems (like CGMU or AID). Furthermore, six domain models were constructed to formally specify the entities and constraints of the application domain required to ensure satisfaction of functional requirements [1]. At each deliverable release, a plenary meeting was held with VdM stakeholders to validate the work done, through semi-structured interviews, and assess the method contributions and progress. We noted the need of:

- A way to quantify the impact or contribution of a goal, in addition to qualifying its nature (positive or negative). This would, for example, allow to distinguish positive contributions of goals *UseAID* (goal diagram of Figure 8.3), *UseVdMCamera* and *UseVdMRadar* to non-functional requirement *Cost [Sensor]*, since the operating costs are not exactly the same. The quantification could be done for example using conditional probabilities on the requirement satisficity: a value between *-1* and *1* to represent the probability that the requirement will be satisfied (or not) knowing that the contribution goal is selected; the sign qualifies the contribution/impact.

- A non-functional goal refinement strategy based on logical formulas that allows to refine a non-functional goal *NFG* into (NFG, P_1), (NFG, P_2), ..., (NFG, P_n) where P_1, P_2, ..., P_n are logical formulas: the satisfaction of *NFG* depends on the satisfaction of *NFG* when P_1 is true, of *NFG* when P_2 is true, ..., and of *NFG* when P_n is true. For example, the satisfaction of goal *Cost [Actuator]* (goal diagram of Figure 8.3) depends on the satisfaction of *Cost [Actuator]* when the user has a smart device and when the user doesn't. Indeed, if we only consider goal *Cost [Actuator]*, the contribution goal *UsePMV* seems useless. But it is better to send notifications through GPS platforms only when a user has a smart device. When a smart device is not available, the only viable option is to use variable message signs (goal *UsePMV*). Thus, goal *UsePMV* cannot be removed because it is the only alternative when user does not have a smart device. This can only be reflected when the non-funtional goal is considered with specific conditions. In fact, a combined use of GPS platforms and PMVs is the most satisfactory alternative.
- A way to refine contribution goals similar to that of Chung *et al.* [38] in order to allow the definition of abstract contribution goals and of sub-contribution goals with specific impacts or contributions. This will, for instance, allow the definition of a contribution goal *useVdMSensors* that will be refined by goals *useVdMCamera* and *UseVdMRadar*.
- An obstacle modeling language, such as the one proposed by Lamsweerde in [139], that distinguishes countermeasures used to detect the occurrence of an obstacle from those used to circumvent it.
- A tool support of the propagation of errors and inconsistencies detected when discharging proof obligations, during the formal verification step, to the corresponding SysML/KAOS models.
- A tool support of the propagation of changes made within a model (SysML/KAOS or *B System* model) to the corresponding one following rules similar to those proposed in [57] for additions performed within a *B System* model.

This work is closely related with the one of Mashkoor *et al.* [99]. While in [99], the transportation system is directly specified in *Event-B*, the SysML/KAOS method uses goal models to represent system requirements, and, as advocated in [31], ontologies to represent domain entities and constraints. Ontologies give the structural part of the *B System* model while goal models provide the behavioral part. The use of SysML/KAOS modeling languages has several advantages, such as a better reusability, maintainability and readability of models. They also facilitate validations with stakeholders while providing and enforcing the refinement logic.

8.5 Conclusion and Future Work

This paper focusses on the use and assessment of the SysML/KAOS method for the high level modeling of functional and non-functional requirements, of domain properties and of safety invariants related to a road transportation system for the City of Montreal (VdM) (see [47, 48]). Translation rules, supported by tools, were used to obtain a formal specification containing the system structure and the skeleton of events. The Rodin platform [35] was used to verify the specification and *ProB* [95] and *B-Motion Studio* [90] to animate and validate it. Compared to other requirements engineering methods such as *KAOS* [139] or *i** [142], SysML/KAOS fills the gap between the goal and domain models on one hand and *B System* (and Event-B) models on the other hand, while being supported by an open-source tool.

VdM stakeholders were involved to assess the modeling deliverables and process and expressed the wish to see the method used in other VdM transportation projects. SysML/KAOS has proven its usefulness and the proposed improvements will be taken into account in next releases of supporting tools.

Chapitre 9

Outillage de la méthode SysML/KAOS à travers la plateforme Openflexo

Résumé

Ce Chapitre introduit la fédération de modèles, une approche qui permet de lier un ensemble de modèles issus de paradigmes hétérogènes. Il décrit également comment elle a été exploitée afin de construire un outil, *FORMOD*, supportant la méthode SysML/KAOS. Dans le cadre de l'application de SysML/KAOS, FORMOD supporte la construction graphique des modèles de buts et de domaine tout en assurant la fédération de ces modèles avec une spécification *B System* évoluant au sein d'un projet de spécification système *Atelier B*. Ceci passe par la fédération des langages de modélisation des buts et du domaine ainsi que du langage de spécification des modèles *B System* de l'*Atelier B*, l'environnement de développement intégré édité par *ClearSy* [41]. Le lecteur qui souhaite utiliser l'outil FORMOD est prié de se référer à l'annexe C pour une description illustrée des différentes étapes qui constituent son scénario principal d'utilisation.

Cette implémentation a été rendue possible grâce à *Openflexo*, une plateforme open source qui implémente les principes de la fédération de modèles. Openflexo définit les mécanismes techniques nécessaires pour maintenir un certain niveau de cohérence entre les divers modèles d'une fédération.

9.1 Généralités sur la fédération de modèles Openflexo

La fédération de modèles est une approche qui permet de lier un ensemble de modèles issus de paradigmes hétérogènes [73] à travers la définition de liens de correspondance et de traçabilité. Cette liaison est définie de façon à assurer une cohabitation souple des modèles : chaque modèle peut continuer d'évoluer dans son paradigme d'origine.

Une fédération de modèles est un ensemble de modèles définissant des liens d'interdépendance. Les liens établis dépendent des objectifs de la fédération. Toute action effectuée sur un modèle est succeptible d'impacter tout ou partie de la fédération [73].

Openflexo [109] est une plateforme open source qui implémente les principes de la fédération de modèles tels que décrits dans [72]. Elle définit les mécanismes techniques nécessaires pour maintenir un certain niveau de cohérence (niveau succeptible de varier suivant les objectifs de la fédération) entre les divers modèles d'une fédération. Elle a été utilisée afin de fédérer les divers langages intervenant au sein de la méthode SysML/KAOS.

La figure 9.1, adaptée de [72], illustre la mise en place d'une fédération de modèles sous *Openflexo*. Une fédération de modèles comprend généralement un ensemble de *modèles conceptuels* ou *modèles virtuels*, internes à Openflexo, et un ensemble de *modèles technologiques* ou *modèles fédérés*, externes à Openflexo [73]. Chaque modèle technologique évolue dans un espace technologique dédié (langage et outil spécifiques). Par contre, tout modèle virtuel est construit sous *Openflexo* en utilisant le langage de modélisation de fédérations *FML (Federation Modeling Language)*. Chaque modèle virtuel représente un langage ; les instanciations du modèle virtuel représentent des modèles conformes au langage défini. Au coeur du langage FML se situe la notion de *concept (flexo concept)*. Un concept est une entité caractérisée par des *propriétés (roles)* et *comportements (behaviors)* spécifiques. Chaque concept peut hériter des propriétés et comportements d'autres concepts et peut également contenir des concepts propres. Le modèle virtuel est un concept spécial qui ne peut être contenu que dans un autre modèle virtuel.

Un adaptateur technologique est une bibliothèque qui définit les connexions entre le moteur d'exécution FML et un espace technologique particulier [73]. Ces adaptateurs permettent l'interfaçage entre des modèles virtuels définis sous Openflexo et des modèles évoluant dans des espaces technologiques spécifiques. Un *model slot* est une entité permettant, au sein d'un modèle virtuel, d'accéder aux éléments définis dans un autre modèle virtuel ou dans un autre espace technologique. Le model slot accède à un espace technologique en utilisant l'adapteur technologique associé. En fonction de l'implémentation de l'adaptateur technologique, le model slot peut donner accès à tout ou partie du modèle qu'il représente.

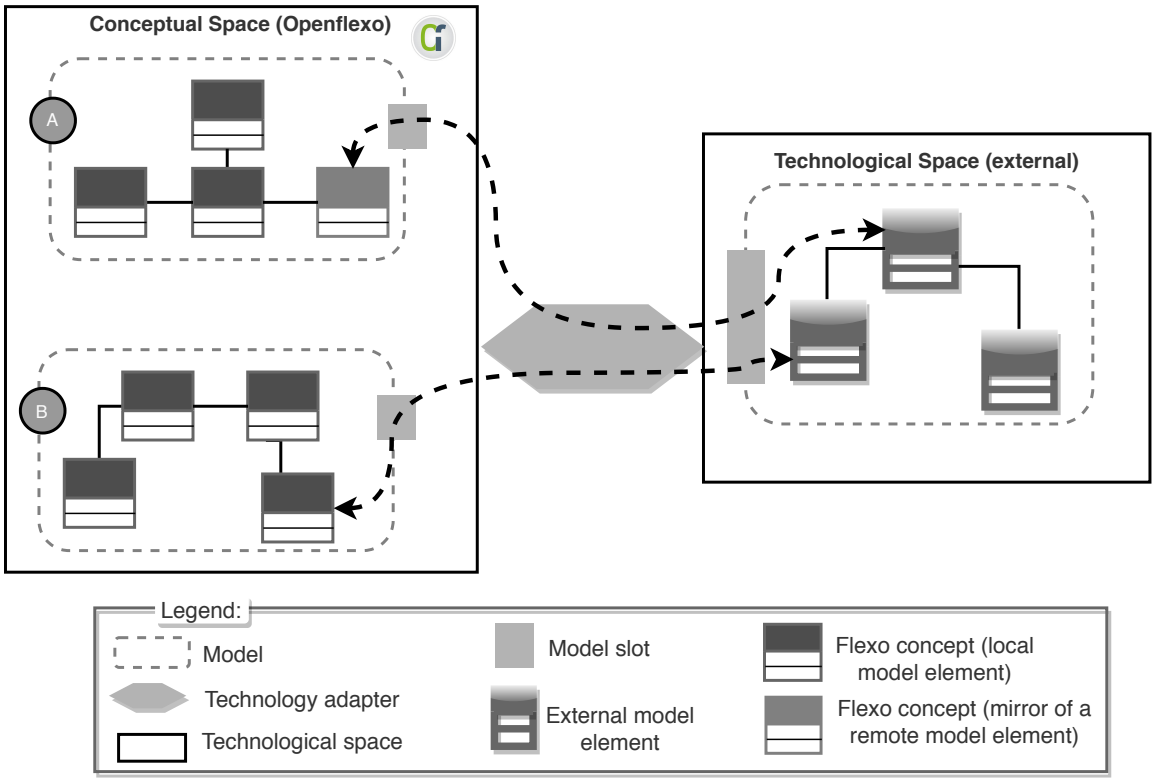


figure 9.1 – Vue d'ensemble de la fédération de modèles sous *Openflexo*

La construction d'une fédération de modèles sous Openflexo passe par l'utilisation de trois principaux éditeurs :

- L'éditeur de modèles virtuels appelé *ViewPointModeller* qui supporte le langage FML et permet la définition des langages fédérés et des règles de fédération.
- L'éditeur *FreeModellingEditor* qui permet l'instanciation des modèles virtuels construits à travers le ViewPointModeller afin de définir des modèles conformes aux langages fédérés.
- L'éditeur *ViewEditor* qui permet d'associer des représentations graphiques tant aux modèles virtuels qu'à leurs instances.

Openflexo, étant lui aussi construit selon les principes de la fédération de modèles, permet un usage simultané des trois éditeurs sus-cités dans le contexte d'une même fédération de modèles. Ceci permet la construction de fédérations selon une approche *top-down* (du métamodèle d'un langage vers des modèles conformes à ce dernier), *bottom-up* (des modèles vers le métamodèle) et *hybride* (*top-down* et *bottom-up*).

9.2 Implémentation Openflexo de SysML/KAOS

9.2.1 Vue générale

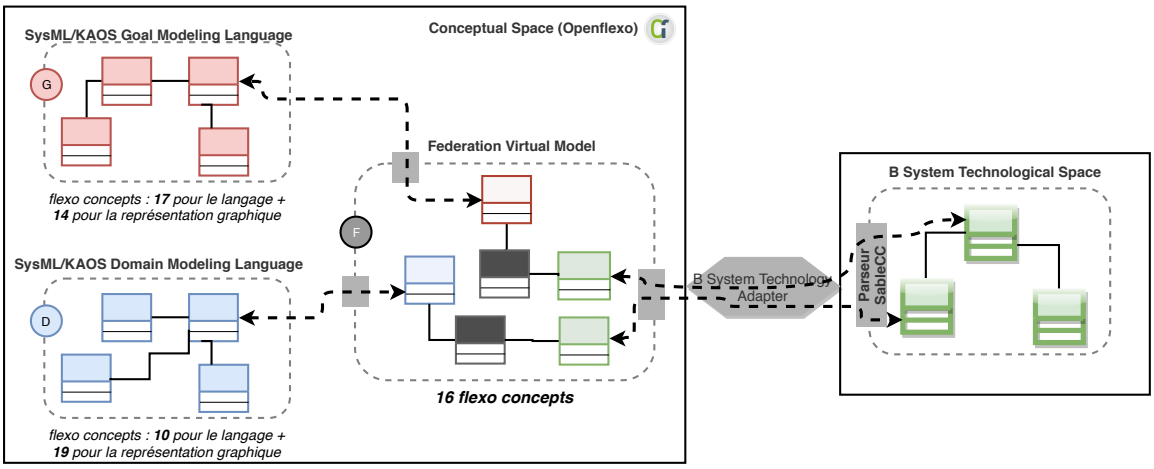


figure 9.2 – Illustration de l'implémentation Openflexo de SysML/KAOS

La figure 9.2 illustre la fédération des divers modèles qui définissent la méthode SysML/KAOS. La fédération a été construite suivant une approche *top-down* du fait de l'existence, au moment de l'outillage, d'une définition rigoureuse et structurée des langages à fédérer et des règles de fédération. Les langages SysML/KAOS de modélisation des buts et domaine sont définis au travers des modèles virtuels Openflexo. Chaque modèle virtuel définit des flexo concepts, un concept pour chaque classe du métamodèle associé au langage.

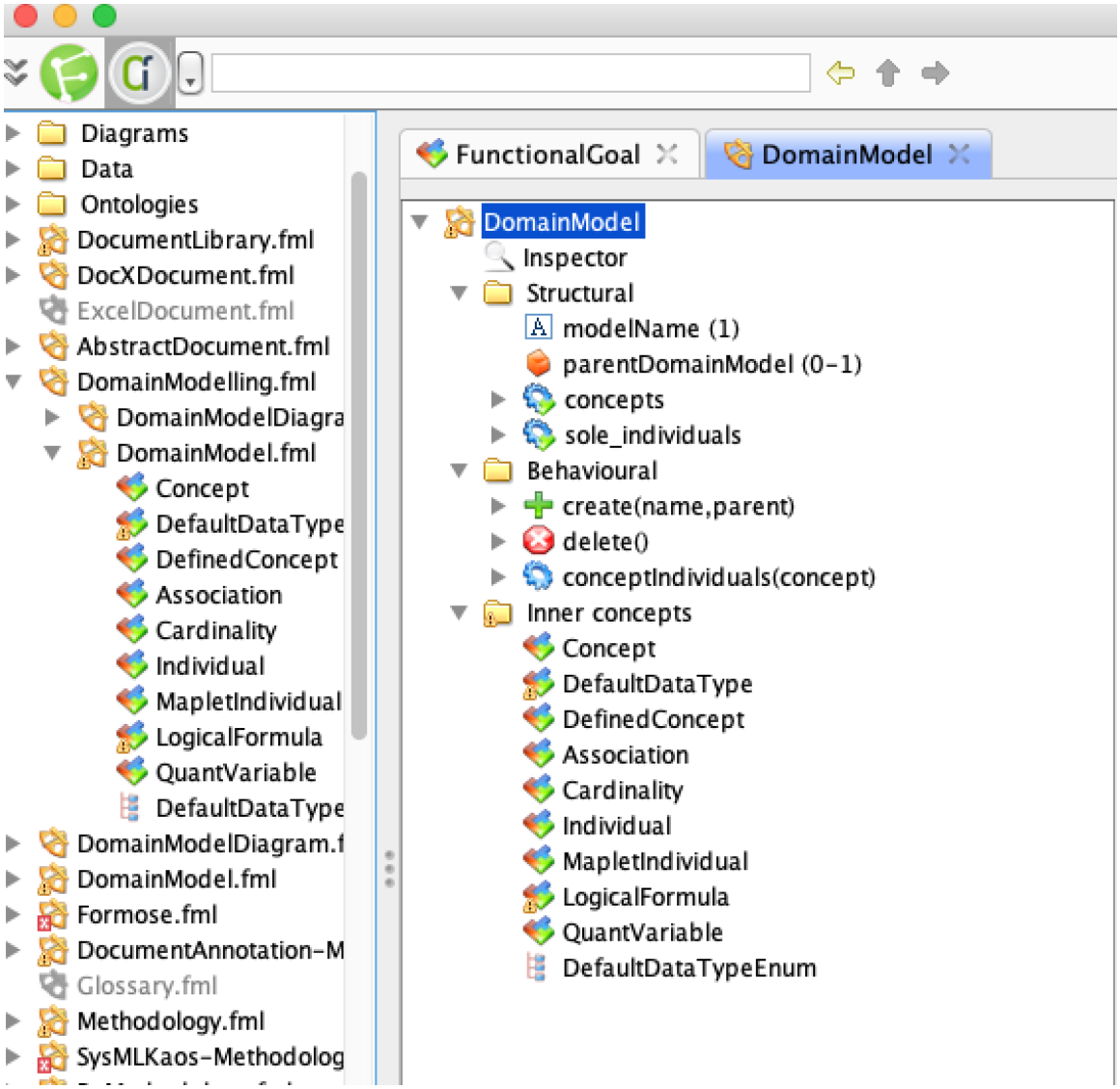


figure 9.3 – Illustration de l'implémentation Openflexo du langage de modélisation du domaine

La figure 9.3 donne un aperçu du modèle virtuel qui implémente le langage SysML/KAOS de modélisation du domaine. Ce modèle virtuel définit les flexo concepts associés aux classes du métamodèle du langage (Association, Individual, LogicalFormula, etc.) ainsi que les comportements permettant de créer et supprimer ses instances (modèles de domaine).

Un troisième modèle virtuel, dit modèle de fédération, est défini afin d'établir les liens de correspondance entre modèles de domaine et de buts, d'une part, et modèles *B System*, d'autre part. Il accède aux différents modèles en passant par des model slots ; le model slot qui le lie aux modèles *B System* utilise un adaptateur technologique spécifique pour l'interaction avec des projets *Atelier B*, ce qui permet à la spécification *B System* d'être enrichie, vérifiée et validée en utilisant cet environnement de développement intégré ainsi que toute la pile de logiciels connexes. Chaque lien de correspondance (voir annexe B) est matérialisé par un flexo concept associé à deux autres de type mirroir : un qui représente un élément de la spécification *B System* et un autre qui représente un élément d'un modèle de domaine ou de buts.

Un comportement spécifique est défini au sein du modèle de fédération afin de : (i) détecter les éléments du modèle de domaine ou de buts nouvellement introduits et, pour chacun, déclencher la création d'une instance du flexo concept de correspondance associé au type de l'élément, ce qui a pour conséquence de déclencher la définition de l'élément *B System* correspondant ; (ii) détecter les éléments du modèle de domaine ou de buts supprimés afin de déclencher la suppression des instances qui leurs sont associées. De même, un comportement est défini afin de détecter et propager les ajouts et suppressions d'éléments au sein d'une spécification *B System*. Il est à noter que pour des raisons techniques liées à *Openflexo*, seuls sont supportés les ajouts et suppressions d'ensembles abstraits et énumérés *B System* et d'éléments d'ensembles énumérés.

9.2.2 Vue détaillée

Au total, 10 flexo concepts ont été définis afin de capturer les diverses entités sur lesquelles repose la modélisation du domaine et 17 flexo concepts ont été définis pour ce qui a trait à la modélisation des buts. La suite de ce Chapitre décrit la définition des principaux éléments sur lesquels repose FORMOD.

Modélisation du domaine

En ce qui concerne la modélisation du domaine, le modèle virtuel `DomainModel` implémente la classe DomainModel du métamodèle de la figure 5.3 et définit :

- Une propriété `modelName` ayant le rôle *chaîne de caractères* qui capture le nom du modèle de domaine. La cardinalité de cette propriété est définie de façon à garantir que tout modèle de domaine soit nommé.

- Une propriété `parentDomainModel` ayant le rôle *instance de flexo concept* qui permet de lier chaque modèle de domaine au modèle, dit parent, qu'il étend. Il est à noter que, par défaut, à la racine de toute hiérarchie de modèles de domaine se trouve un modèle appelé `rootDomainModel` qui définit les éléments de base présents au sein de tout modèle de domaine à l'exemple des types de données primaires (*INTEGER*, *NATURAL*, etc.).
- Un flexo concept `Concept` qui implémente la classe Concept du métamodèle de la figure 5.3 et définit des propriétés primitives représentant les nom, variabilité et énumérabilité de l'entité. Il définit également, en plus des comportements de création et de suppression, une propriété `parentConcept` ayant le rôle *instance de flexo concept* afin de lier chaque concept parent à son sous-concept.
- Un flexo concept `Association` qui implémente la classe Association du métamodèle de la figure 5.3. Le langage FML permet de définir des liens d'héritage entre flexo concepts. Ainsi, le flexo concept `Association` est défini comme un sous-concept du flexo concept `Concept`. Il définit en outre des propriétés propres aux associations à l'exemple du domaine (source de l'association), du range (cible de l'association) et des cardinalités.
- Un flexo concept `Individual` qui implémente la classe Individual du métamodèle de la figure 5.3 et définit des propriétés primitives représentant les nom et variabilité de l'instance. Il définit également, en plus des comportements de création et de suppression, des propriétés, instances de flexo concepts, afin de définir le concept d'appartenance de l'individu (propriété `individualOf`) ainsi que l'individu constant qui représente sa valeur initiale (propriété `initialValue`). Il est à noter que la notion de valeur initiale n'a de sens que lorsque l'individu est variable.
- Un flexo concept `MapletIndividual` qui spécialise le flexo concept `Individual` et définit des propriétés propres aux couples d'individus à l'exemple de l'antécédent et de l'image. Il spécialise également la propriété `individualOf` afin de garantir que tout couple (instance de MapletIndividual) soit individu d'une association.

L'utilisation de l'éditeur *ViewEditor* d'Openflexo a permis d'associer une représentation graphique à chaque flexo concept du modèle virtuel `DomainModel`. Par exemple, la figure 9.4 donne un aperçu des représentations graphiques associées aux flexo concepts sus-cités.

Chaque représentation est définie à travers un flexo concept de type graphique qui décrit les attributs graphiques de la forme. Ce flexo concept décrit également les comportements spécifiques attendus. Il s'agit par exemple du comportement attendu en cas de création, de suppression ou de glisser-déposer. Au total, *19* flexo concepts graphiques ont été définis afin d'assurer une représentation adéquate

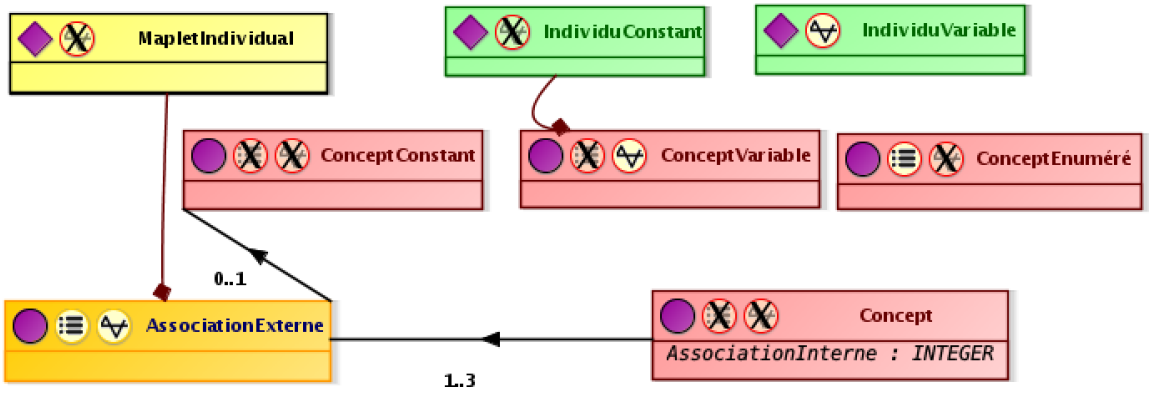


figure 9.4 – Formes graphiques associées aux principaux éléments d'une modélisation du domaine

des différents éléments associés à la modélisation du domaine. Des model slots assurent la jonction entre les modèles de domaine et leurs représentations : un model slot permet aux flexo concepts graphiques d'accéder (et manipuler) aux formes graphiques et un autre permet l'accès (et la manipulation) aux instances de flexo concepts du modèle virtuel `DomainModel`.

Modélisation des buts

Suivant le même principe que pour la modélisation du domaine, la modélisation des buts est implémentée à travers un modèle virtuel qui définit :

- Des propriétés permettant l'identification des diagrammes de buts et la construction des hiérarchies de diagrammes. La hiérarchisation repose sur une propriété `parentModel` qui, si instanciée, lie chaque diagramme de buts à son diagramme parent. Pour un diagramme donné, le diagramme parent est celui qui introduit son (ses) but(s) racine(s) [92].
- Un flexo concept `FunctionalGoal` pour les buts fonctionnels. Des propriétés y sont définies afin de lier chaque but fonctionnel élémentaire à l'agent auquel il est assigné.
- Un flexo concept `Refinement` qui représente les raffinements entre buts. Des propriétés et comportements y sont définis afin de permettre l'identification et l'ajustement (i) du type de raffinement (cf Chapitre 1 : *And*, *Or*, *Milestone* et *Data Refinement*) et (ii) des buts abstrait et concret(s).

- Un flexo concept `NonFunctionalGoal` pour les buts non-fonctionnels, chaque but étant caractérisé par son type ainsi que par le sujet auquel il s'applique. Il est à noter que le sujet désigne un concept ou un individu du modèle de domaine.
- Des flexo concepts pour les buts de contributions, les niveaux de raffinement, les agents, etc.

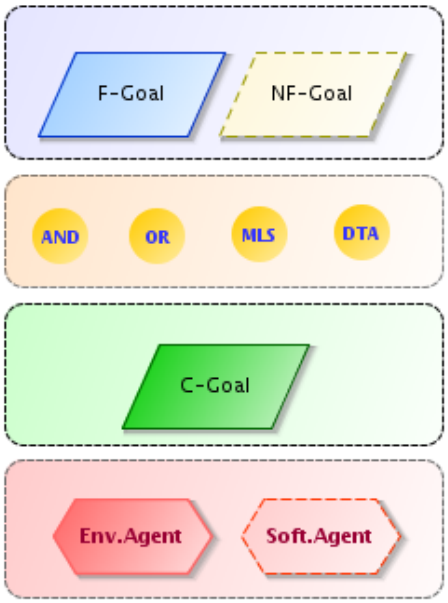


figure 9.5 – Formes graphiques associées aux principaux éléments d'une modélisation de buts

Pour assurer une représentation adéquate des diagrammes de buts, *14* flexo concepts graphiques ont été définis. La figure 9.5 donne un aperçu des représentations graphiques associées aux principaux éléments d'une modélisation de buts. De la gauche vers la droite et du haut vers le bas, sont représentés : les buts fonctionnels (`F-Goal`) et non-fonctionnels (`NF-Goal`), les opérateurs de raffinement de buts (*And*, *Or*, *Milestone* (`MLS`) et *Data Refinement* (`DTA`), les buts de contributions (`C-Goal`) et les agents auxquels peuvent être assignés des buts fonctionnels élémentaires : agent externe au système (`Env.Agent`) et agent interne (`Soft.Agent`). Un but élémentaire affecté à un agent externe est une *attente* tandis qu'un but élémentaire affecté à un agent interne est une *exigence*.

Règles de fédération entre modèles SysML/KAOS et B System

Les règles permettant de fédérer les modèles SysML/KAOS et *B System* sont implémentées au sein d'un modèle virtuel de fédération. Ce modèle virtuel définit des model slots afin de permettre à chaque règle d'accéder (manipuler) tant aux modèles de domaine et de buts qu'aux projets *Atelier B* contenant la spécification *B System*. Il définit également :

- Un comportement de création assurant qu'à l'instanciation de la fédération, un contexte et une machine soient associés à chaque niveau de raffinement du modèle des buts fonctionnels.
- Un comportement `update_structural_part` pour l'établissement des liens de correspondance entre modèles de domaine et spécifications *B System*. Il s'agit ici de (i) détecter les ajouts effectués au sein du modèle de domaine et pour chaque nouvel élément *xx* créer une instance du lien de correspondance associé au type de *xx* : l'instanciation du lien de correspondance se traduit par l'introduction d'un nouvel élément au sein de la spécification *B System*, conformément aux règles de correspondance (voir annexe B), lié à *xx* à travers l'instance du lien de correspondance ; (ii) détecter les suppressions et les propager par la suppression des instances de liens de correspondance associées : la suppression d'une instance de lien entraîne la suppression de l'élément *B System* correspondant.
- Un comportement `update_behavioral_part` pour l'établissement des liens de correspondance entre buts fonctionnels et évènements *B System*. Ce comportement fait également la correspondance entre raffinements de buts et raffinements d'évènements : *And* ⇒ *ref_and*, *Or* ⇒ *ref_or*, *Milestone* ⇒ *ref_milestone* et *Data Refinement* ⇒ *ref*. Les raffinements identifiés par *ref_and*, *ref_or* et *ref_milestone* sont des raffinements intégrés dans l'*Atelier B* et dont la correction requiert l'établissement des obligations de preuve de raffinement SysML/KAOS [8].
- Un comportement `back_Propagate_Structural_Part_Updates` pour la propagation des ajouts et suppressions d'éléments, de la partie structurelle d'une spécification *B System*, vers les modèles SysML/KAOS correspondants.
- *16* flexo concepts implémentant les liens de correspondance. Chaque flexo concept de correspondance définit une propriété qui référence un élément *xx* d'un modèle de domaine ou de but et une autre propriété qui référence l'élément de la spécification *B System* associé à *xx*. Il surcharge également ses constructeur et destructeur de façon à associer, pour chaque élément *xx*, sa création à l'insertion du correspondant de *xx* et sa suppression à celle du correspondant de *xx*.

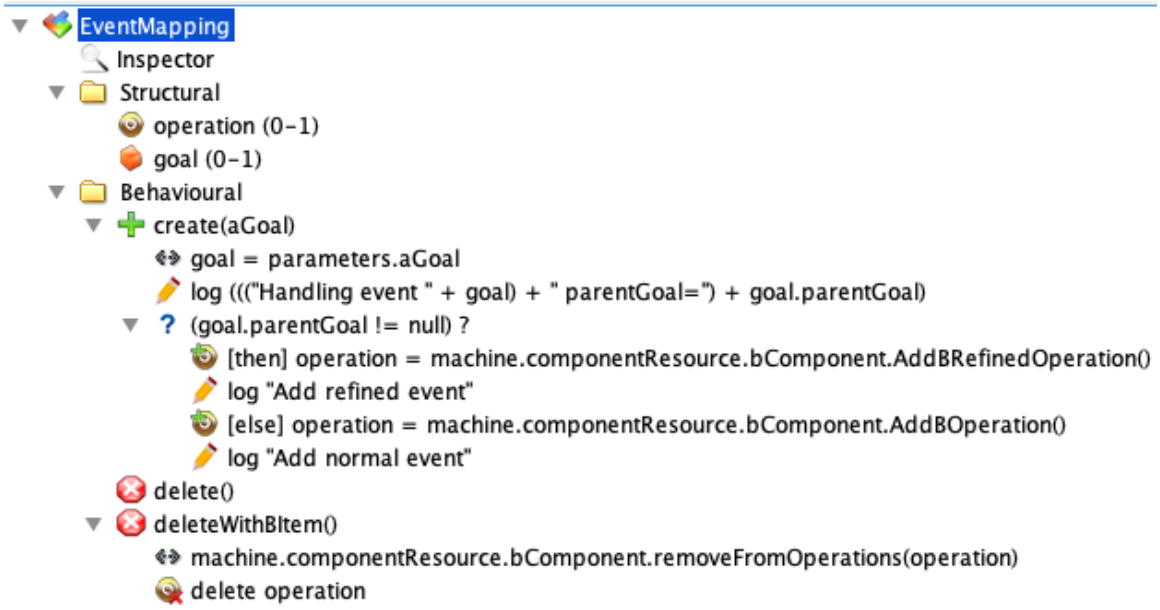


figure 9.6 – Implémentation du flexo concept `EventMapping`

Par exemple, la figure 9.6 illustre l'implémentation du flexo concept `EventMapping` qui représente le lien de correspondance entre un but fonctionnel SysML/KAOS (propriété `goal`) et un évènement *B System* (propriété `operation`). Comme l'illustre le corps du comportement `create`, un but sans parent est mis en correspondance avec un évènement abstrait tandis qu'un but ayant un parent est mis en correspondance avec un évènement concret qui raffine le correspondant du but parent ; le type de raffinement *B System* étant fonction de l'opérateur de raffinement SysML/KAOS utilisé.
Les primitives permettant de manipuler la spécification *B System* à l'exemple de `AddBRefinedOperation` ou `removeFromOperations` sont définies au sein de l'adaptateur technologique *B System* connecté au modèle virtuel de fédération. Ces primitives, exploitées au sein d'une spécification *FML*, sont implémentées en *Java* et utilisent *SableCC* [67] pour agir sur des modèles *B System*.

Conclusion

Cette thèse a consisté en l'extension, l'évaluation et l'outillage de la méthode d'ingénierie des exigences SysML/KAOS. Il a tout d'abord été question d'introduire un langage permettant de représenter le domaine d'un système dont les exigences sont capturées à travers le langage de modélisation des buts de SysML/KAOS. Il s'agit, de notre connaissance de la littérature, du premier langage de modélisation du domaine qui permet de distinguer les éléments statiques des éléments dynamiques, les changements d'états des éléments dynamiques, à mesure que le système satisfait ses exigences, pouvant être exprimés graphiquement au travers d'ASTDs [62]. Ceci permet de définir de façon non-ambigüe les aspects statique et dynamique de la partie structurelle de la spécification *B System* qui formalise les exigences du système. Le métamodèle du langage ainsi que les règles nécessaires pour l'établissement et le maintien des correspondances entre modèles de domaine et spécifications *B System* ont été spécifiés et vérifiés formellement en utilisant la méthode Event-B.

Couplé à la modélisation des buts SysML/KAOS, le langage a été évalué sur plusieurs études de cas d'envergure industrielle : spécification du protocole de transport ferrovier *hybrid ERTMS/ETCS level 3* [58], spécification du système de contrôle d'une chaudière à vapeur [57], spécification d'un système de gestion du transport routier pour le compte de la *Ville de Montréal*, etc [133]. La méthode ainsi définie permet la vérification formelle des exigences et facilite leur validation par des parties prenantes non spécialistes de méthodes formelles. Son passage à l'échelle est lié aux mécanismes de raffinement et de décomposition définis au sein des langages de modélisation de SysML/KAOS, étendus afin de permettre, à la suite de décompositions, la preuve formelle de la satisfaction des exigences et de la préservation des invariants. Toutefois, les tâches de spécification des formules logiques et du corps des évènements et de vérification et validation formelles nécessitent non seulement du temps, mais surtout l'implication d'experts en méthodes formelles. Il s'agit là du prix à payer pour des exigences formellement correctes.

Synthèse des contributions

La contribution majeure de ce travail de thèse réside dans la définition d'un langage de modélisation de domaine structuré, non ambigu et suffisamment expressif, supporté par un outil libre et compatible avec le langage de modélisation d'exigences de SysML/KAOS. La version initiale de ce langage (Chapitre 2), fondée sur *OWL (Ontology Web Language)* [118] et *PLIB (Part Library)* [112], a été ajustée au fil des études de cas (Chapitre 5).

Afin de permettre la génération automatique de la partie structurelle des spécifications *B System* d'exigences, des règles ont été définies. Ces règles ont également été formellement spécifiées et vérifiées afin de garantir des propriétés nécessaires liées à la criticité des systèmes considérés (Chapitre 3). Il s'agit notamment de la *convergence*, de la *cohérence* (vis-à-vis des invariants propres aux langages source et cible) et de l'*isomorphisme*.

Des règles ont également été définies et formellement vérifiées afin de maintenir l'adéquation, en cas d'ajout d'éléments, entre la partie structurelle d'une spécification *B System* et les modèles SysML/KAOS auxquels elle est associée (Chapitre 4).

Finalement, des mécanismes ont été définis afin de garantir formellement que chaque exigence SysML/KAOS affectée à un sous-système sera correctement satisfaite par ce dernier, dans la limite définie par les spécifications *B System* du système et des sous-systèmes (Chapitre 6).

Le langage de modélisation introduit et les règles ont été évalués, conjointement au langage de modélisation des buts, dans le cadre de la spécification formelle des exigences : (i) d'un protocole de transport ferroviaire *hybrid ERTMS/ETCS level 3* [79] (Chapitre 7) ; (ii) d'un système de transport urbain pour le compte de la *Ville de Montréal* [48] (Chapitre 8) ; (iii) d'un système de contrôle d'une chaudière à vapeur [22] (Chapitres 4, 6) ; (iv) et d'un protocole de communication proposé pour la transformation booléenne des données échangées entre plusieurs agents au sein d'une infrastructure ferroviaire [129] (Chapitre 5).

Menaces à la validité

Pour une meilleure analyse de l'utilité et de l'utilisabilité de la méthode formelle d'ingénierie des exigences proposée à l'issue de ce travail de thèse, il est nécessaire de l'évaluer sur d'avantage d'études de cas, d'échelles et de domaines plus divers. En effet, malgré la diversité des études de cas considérées dans le cadre de ce travail de thèse et ayant contribuées à affiner les méthode et langages proposés, il n'est pas possible de dire avec certitude que les résultats obtenus sontgénéralisables. Quelques facteurs contribuant à limiter la généralisabilité des résultats :

- Le nombre peu conséquent des parties prenantes : entre 6 (spécification formelle des exigences d'un système de transport urbain pour le compte de la *Ville de Montréal*) et 20 (spécification formelle des exigences du protocole de communication ferroviaire *Saturn*) ;
- L'absence de séparation stricte entre les experts en charge respectivement (i) de modéliser les exigences du système, (ii) de modéliser le domaine et (iii) de formaliser et vérifier formellement les exigences ;
- La faible diversité des domaines considérés : il s'est agit principalement du transport ferroviaire, de l'aéronautique, et du transport routier.

Par ailleurs, la méthode requiert l'existence d'une description (textuelle, graphique, ...) du système considéré, comprenant des objectifs de haut niveau, ainsi que du domaine d'application, afin de permettre la spécification, la vérification ainsi que la validation formelle des exigences. Elle est donc difficilement exploitable, dans le cas de systèmes inexistants, sans une analyse préalable.

Perspectives

La spécification obtenue au travers de SysML/KAOS se situe dans *l'espace des problèmes*. En effet, elle se focalise sur la validation, au regard des besoins exprimés par les parties prenantes, et la vérification, au regard des propriétés du domaine et des contraintes de satisfaction, des exigences du système. Le comportement de l'environnement est en outre laissé non-déterministe, dans les limites imposées par les contraintes définies au sein des modèles de domaine. Bien entendu, se focaliser uniquement sur les exigences n'est pas suffisant. Il est nécessaire d'assurer la faisabilité d'un processus itératif englobant la gestion des exigences, la conception des architectures et le développement du système [108]. Ceci nécessite des liens entre les divers modèles associés à ces différentes phases. Il s'agit par exemple des liens de correspondance entre modèles SysML/KAOS et spécifications *B System*. Il s'agit également des liens de raffinement entre la spécification *B System* issue de la formalisation des exigences et la spécification, dans *l'espace des solutions*, qui intègre les choix de conception nécessaires à l'implémentation du système. Étudier l'existence, l'établissement et la préservation de ces liens constitue un prolongement des contributions de ce travail de thèse.

Une extension de ce travail de thèse se situe également dans la définition et l'outillage de mécanismes assurant la propagation des erreurs de preuve, d'une spécification *B System*, vers les modèles SysML/KAOS qui y sont associés. De tels mécanismes exploiteraient tant les liens de correspondance établis dans [65] que les études réalisées pour la propagation des ajouts d'éléments au sein d'une spécification *B System* [57]. Il est à noter que la liste complète des liens de correspondance qui ont été définis est disponible à l'annexe B.

Une perspective plus pratique et industrielle réside dans l'évaluation de la méthode d'ingénierie des exigences définie sur une plus grande varieté d'études de cas afin de confirmer ou infirmer les observations rapportées dans cette thèse.

Appendix A

Comprehensive Definition of the Domain Modeling Language and of the Correspondence Rules

A.1 Event-B Specification of the SysML/KAOS Domain Modeling and B System Specification Languages

CONTEXT Domain_Metamodel_Context
SETS DomainModel_Set Relation_Set Concept_Set Relation_Maplet_Set Individual_Set Attribute_Maplet_Set Attribute_Set DataValue_Set DataSet_Set RelationCharacteristics_Set
CONSTANTS _NATURAL _INTEGER _FLOAT _BOOL _STRING isTransitive isSymmetric
AXIOMS

axiom1: $finite(DataValue\_Set)$
axiom2: $\{\_NATURAL, \_INTEGER, \_FLOAT, \_BOOL, \_STRING\} \subseteq DataSet\_Set$
axiom3: $partition(\{\_NATURAL, \_INTEGER, \_FLOAT, \_BOOL, \_STRING\}, \{\_NATURAL\}, \{\_INTEGER\}, \{\_FLOAT\}, \{\_BOOL\}, \{\_STRING\})$
axiom4: $partition(RelationCharacteristics\_Set, \{isTransitive\}, \{isSymmetric\})$
axiom5: $finite(DomainModel\_Set) \land finite(Concept\_Set) \land finite(DataSet\_Set)$
axiom6: $finite(DataValue\_Set) \land finite(Individual\_Set) \land finite(Relation\_Set)$
axiom7: $finite(Attribute\_Set) \land finite(Relation\_Maplet\_Set) \land finite(Attribute\_Maplet\_Set)$

END

CONTEXT EventB_Metamodel_Context
SETS Component_Set Variable_Set Constant_Set Set_Set SetItem_Set LogicFormula_Set
the subset of logical formulas that can directly be expressed within the specification,
without the need for an explicit constructor, will not be contained in this set.
This is for example the case of equality between elements.
Operator InitialisationAction_Set
CONSTANTS B_NATURAL B_INTEGER B_FLOAT B_BOOL B_STRING Inclusion_OP Belonging_OP BecomeEqual2SetOf_OP RelationSet_OP FunctionSet_OP Maplet_OP Equal2SetOf_OP BecomeEqual2EmptySet_OP RelationComposition_OP Inversion_OP Equality_OP
AXIOMS

axiom1: $finite(SetItem\_Set)$
axiom2: $\{B\_NATURAL, B\_INTEGER, B\_FLOAT, B\_BOOL, B\_STRING\} \subseteq Set\_Set$
axiom3: $partition(\{B\_NATURAL, B\_INTEGER, B\_FLOAT, B\_BOOL, B\_STRING\}, \{B\_NATURAL\}, \{B\_INTEGER\}, \{B\_FLOAT\}, \{B\_BOOL\}, \{B\_STRING\})$

axiom4: $partition(Operator, \{Inclusion\_OP\}, \{Belonging\_OP\}, \{BecomeEqual2SetOf\_OP\}, \{RelationSet\_OP\}, \{Maplet\_OP\}, \{Equal2SetOf\_OP\}, \{BecomeEqual2EmptySet\_OP\}, \{FunctionSet\_OP\}, \{RelationComposition\_OP\}, \{Inversion\_OP\}, \{Equality\_OP\})$

axiom5: $finite(Variable\_Set) \land finite(Set\_Set) \land finite(Constant\_Set) \land finite(Component\_Set) \land finite(LogicFormula\_Set)$

END

MACHINE event_b_specs_from_ontologies
SEES EventB_Metamodel_Context,Domain_Metamodel_Context
VARIABLES Component System Refinement Refinement_refines_Component DomainModel DomainModel_parent_DomainModel DomainModel_corresp_Component
INVARIANTS

inv0_1: $Component \subseteq Component\_Set$
inv0_2: $partition(Component, System, Refinement)$
Domain Model
inv0_3: $DomainModel \subseteq DomainModel\_Set$
inv0_4: $DomainModel\_parent\_DomainModel \in DomainModel \nrightarrow DomainModel$
inv0_5: $DomainModel\_corresp\_Component \in DomainModel \nrightarrow Component$
inv0_6: $Refinement\_refines\_Component \in Refinement \rightarrowtail Component$
inv0_7: $\forall xx \cdot (\forall px \cdot ((xx \in dom(DomainModel\_parent\_DomainModel) \land px = DomainModel\_parent\_DomainModel(xx) \land px \in dom(DomainModel\_corresp\_Component) \land xx \notin dom(DomainModel\_corresp\_Component)) \Rightarrow DomainModel\_corresp\_Component(px) \notin ran(Refinement\_refines\_Component)))$
inv0_8: $\forall xx, pxx \cdot ((xx \in dom(DomainModel\_parent\_DomainModel) \land pxx = DomainModel\_parent\_DomainModel(xx) \land \{xx, pxx\} \subseteq dom(DomainModel\_corresp\_Component)) \Rightarrow (DomainModel\_corresp\_Component(xx) \in dom(Refinement\_refines\_Component) \land Refinement\_refines\_Component(DomainModel\_corresp\_Component(xx)) = DomainModel\_corresp\_Component(pxx)))$

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

inv0_10: $\forall xx, pxx \cdot ((xx \in dom(DomainModel\_parent\_DomainModel) \land pxx = DomainModel\_parent\_DomainModel(xx) \land pxx \notin dom(DomainModel\_corresp\_Component)) \Rightarrow xx \notin dom(DomainModel\_corresp\_Component))$
inv0_11: $\forall o\_xx, o\_pxx \cdot ((o\_xx \in dom(Refinement\_refines\_Component) \land o\_pxx = Refinement\_refines\_Component(o\_xx) \land o\_pxx \notin ran(DomainModel\_corresp\_Component)) \Rightarrow o\_xx \notin ran(DomainModel\_corresp\_Component))$

VARIANT

$DomainModel \setminus dom(DomainModel\_corresp\_Component)$

END

MACHINE event_b_specs_from_ontologies_ref_1
REFINES event_b_specs_from_ontologies
SEES EventB_Metamodel_Context,Domain_Metamodel_Context
VARIABLES DomainModel DomainModel_parent_DomainModel Variable Constant Set SetItem AbstractSet EnumeratedSet Invariant Property LogicFormula InitialisationAction
Event-B associations
Variable_definedIn_Component Constant_definedIn_Component Set_definedIn_Component LogicFormula_definedIn_Component Invariant_involves_Variables Constant_isInvolvedIn_LogicFormulas LogicFormula_involves_Sets LogicFormula_involves_SetItems LogicFormula_uses_Operators Variable_typing_Invariant Constant_typing_Property SetItem_itemOf_EnumeratedSet InitialisationAction_uses_Operators Variable_init_InitialisationAction InitialisationAction_involves_Constants
Domain Model sets

Concept Individual DataValue DataSet DefaultDataSet CustomDataSet EnumeratedDataSet Relation RelationMaplet AttributeMaplet Attribute

Domain Model attributes

Concept_isVariable Relation_isVariable Relation_isTransitive Relation_isSymmetric relation_isASymmetric Relation_isReflexive Relation_isIrreflexive Attribute_isVariable Attribute_isFunctional

Domain Model associations

Concept_definedIn_DomainModel DataSet_definedIn_DomainModel Concept_parentConcept_Concept Individual_individualOf_Concept DataValue_valueOf_DataSet DataValue_elements_EnumeratedDataSet Relation_definedIn_DomainModel Attribute_definedIn_DomainModel Relation_domain_Concept Relation_range_Concept Relation_DomainCardinality_minCardinality Relation_DomainCardinality_maxCardinality Relation_RangeCardinality_minCardinality Relation_RangeCardinality_maxCardinality RelationMaplet_mapletOf_Relation RelationMaplet_antecedent_Individual RelationMaplet_image_Individual Attribute_domain_Concept Attribute_range_DataSet AttributeMaplet_mapletOf_Attribute AttributeMaplet_antecedent_Individual AttributeMaplet_image_DataValue

correspondences

Concept_corresp_AbstractSet DomainModel_corresp_Component EnumeratedDataSet_corresp_EnumeratedSet DataValue_corresp_SetItem CustomDataSet_corresp_AbstractSet DefaultDataSet_corresp_AbstractSet Concept_corresp_Constant Individual_corresp_Constant DataValue_corresp_Constant Concept_corresp_Variable Relation_Type Relation_corresp_Constant Relation_corresp_Variable Attribute_Type Attribute_corresp_Constant Attribute_corresp_Variable RelationCharacteristic_corresp_LogicFormula RelationMaplet_corresp_Constant DataSet_corresp_Set AttributeMaplet_corresp_Constant

INVARIANTS

inv1_1: $Variable \subseteq Variable\_Set$

inv1_2: $Constant \subseteq Constant\_Set$

inv1_3: $Set \subseteq Set\_Set$

inv1_4: $partition(Set, AbstractSet, EnumeratedSet)$

inv1_5: $SetItem \subseteq SetItem\_Set$

inv1_6: $Variable\_definedIn\_Component \in Variable \rightarrow Component$

inv1_7: $Constant\_definedIn\_Component \in Constant \rightarrow Component$

inv1_8: $Set\_definedIn\_Component \in Set \rightarrow Component$

inv1_9: $SetItem\_itemOf\_EnumeratedSet \in SetItem \twoheadrightarrow EnumeratedSet$

Domain Model

inv1_10: $Concept \subseteq Concept\_Set$

inv1_11: $Individual \subseteq Individual\_Set$

inv1_12: $DataValue \subseteq DataValue\_Set$

inv1_13: $DataSet \subseteq DataSet\_Set$

inv1_14: $partition(DataSet, DefaultDataSet, CustomDataSet)$

inv1_15: $EnumeratedDataSet \subseteq CustomDataSet$

inv1_16: $Concept\_isVariable \in Concept \rightarrow BOOL$

inv1_17: $Concept\_definedIn\_DomainModel \in Concept \rightarrow DomainModel$

inv1_18: $DataSet\_definedIn\_DomainModel \in DataSet \rightarrow DomainModel$

inv1_19: $Concept\_parentConcept\_Concept \in Concept \nrightarrow Concept$

inv1_20: $Individual\_individualOf\_Concept \in Individual \rightarrow Concept$

inv1_21: $DataValue\_valueOf\_DataSet \in DataValue \rightarrow DataSet$

inv1_22: $DataValue\_elements\_EnumeratedDataSet \in DataValue \twoheadrightarrow EnumeratedDataSet$

inv1_23: $Concept\_corresp\_AbstractSet \in Concept \rightarrowtail AbstractSet$

inv1_24: $EnumeratedDataSet\_corresp\_EnumeratedSet \in EnumeratedDataSet \rightarrowtail EnumeratedSet$

inv1_25: $DataValue\_corresp\_SetItem \in DataValue \rightarrowtail SetItem$

inv1_26: $\forall xx \cdot (xx \in EnumeratedDataSet \land xx \notin dom(EnumeratedDataSet\_corresp\_EnumeratedSet) \Rightarrow DataValue\_elements\_EnumeratedDataSet^{-1}[\{xx\}] \cap dom(DataValue\_corresp\_SetItem) = \emptyset)$

inv1_27: $CustomDataSet\_corresp\_AbstractSet \in CustomDataSet \rightarrowtail AbstractSet$

inv1_28: $\{\_NATURAL, \_INTEGER, \_FLOAT, \_BOOL, \_STRING\} \cap CustomDataSet = \emptyset$

inv1_29: $DefaultDataSet\_corresp\_AbstractSet \in DefaultDataSet \rightarrowtail AbstractSet$

inv1_30: $\{B\_NATURAL, B\_INTEGER, B\_FLOAT, B\_BOOL, B\_STRING\} \cap EnumeratedSet = \emptyset$

inv1_31: $Concept\_corresp\_Constant \in Concept \rightarrowtail Constant$

inv1_33: $LogicFormula \subseteq LogicFormula\_Set$

inv1_34: $Property \subseteq LogicFormula$
inv1_35: $Invariant \subseteq LogicFormula$
inv1_36: $LogicFormula\_definedIn\_Component \in LogicFormula \rightarrow Component$
inv1_37: $Invariant\_involves\_Variables \in Invariant \rightarrow (\mathbb{N}_1 \nrightarrow Variable)$
logic formula operands can be variables, constants, sets or set items, indexed by their appearance order number. The first operand is indexed by 1, no matter it's type.
inv1_38: $ran(union(ran(Invariant\_involves\_Variables))) = Variable$
inv1_39: $Constant\_isInvolvedIn\_LogicFormulas \in Constant \rightarrow \mathbb{P}_1(\mathbb{N}_1 \times LogicFormula)$
When appearance order does not matter, we may index all constants using the same number.
inv1_40: $\forall cons \cdot (cons \in Constant \Rightarrow ran(Constant\_isInvolvedIn\_LogicFormulas(cons)) \cap Property \neq \emptyset)$
inv1_41: $LogicFormula\_involves\_Sets \in LogicFormula \rightarrow (\mathbb{N}_1 \nrightarrow Set)$
inv1_42: $LogicFormula\_uses\_Operators \in LogicFormula \rightarrow (\mathbb{N}_1 \nrightarrow Operator)$
inv1_44: $Individual\_corresp\_Constant \in Individual \rightarrowtail Constant$
inv1_45: $DataValue\_corresp\_Constant \in DataValue \rightarrowtail Constant$
inv1_46: $Concept\_corresp\_Variable \in Concept \rightarrowtail Variable$
inv1_47: $InitialisationAction \subseteq InitialisationAction\_Set$
inv1_49: $InitialisationAction\_uses\_Operators \in InitialisationAction \rightarrow (\mathbb{N}_1 \nrightarrow Operator)$
inv1_50: $Variable\_init\_InitialisationAction \in Variable \rightarrowtail\!\!\!\!\!\rightarrow InitialisationAction$
for initialisation actions, the assigned operand is the involved variable.
inv1_52: $InitialisationAction\_involves\_Constants \in InitialisationAction \rightarrow (\mathbb{N}_1 \nrightarrow Constant)$
**********relations/attributes**********
inv1_53: $Relation \subseteq Relation\_Set$
inv1_56: $RelationMaplet \subseteq Relation\_Maplet\_Set$
inv1_57: $AttributeMaplet \subseteq Attribute\_Maplet\_Set$
inv1_58: $Attribute \subseteq Attribute\_Set$
inv1_59: $Relation\_isVariable \in Relation \rightarrow BOOL$
inv1_60: $Relation\_isTransitive \in Relation \nrightarrow BOOL$
inv1_61: $Relation\_isSymmetric \in Relation \nrightarrow BOOL$
inv1_62: $relation\_isASymmetric \in Relation \nrightarrow BOOL$
inv1_63: $Relation\_isReflexive \in Relation \nrightarrow BOOL$
inv1_64: $Relation\_isIrreflexive \in Relation \nrightarrow BOOL$
inv1_65: $Relation\_DomainCardinality\_minCardinality \in Relation \nrightarrow \mathbb{N}$
inv1_66: $Relation\_DomainCardinality\_maxCardinality \in Relation \nrightarrow (\mathbb{N} \cup \{-1\})$
inv1_67: $Relation\_RangeCardinality\_minCardinality \in Relation \nrightarrow \mathbb{N}$
inv1_68: $Relation\_RangeCardinality\_maxCardinality \in Relation \nrightarrow (\mathbb{N} \cup \{-1\})$
inv1_69: $Attribute\_isVariable \in Attribute \rightarrow BOOL$
inv1_70: $Attribute\_isFunctional \in Attribute \nrightarrow BOOL$
inv1_71: $Relation\_definedIn\_DomainModel \in Relation \rightarrow DomainModel$
inv1_72: $Attribute\_definedIn\_DomainModel \in Attribute \rightarrow DomainModel$
inv1_73: $Relation\_domain\_Concept \in Relation \rightarrow Concept$
inv1_74: $Relation\_range\_Concept \in Relation \rightarrow Concept$
inv1_77: $RelationMaplet\_mapletOf\_Relation \in RelationMaplet \rightarrow Relation$
inv1_78: $RelationMaplet\_antecedent\_Individual \in RelationMaplet \rightarrow Individual$
inv1_79: $RelationMaplet\_image\_Individual \in RelationMaplet \rightarrow Individual$
inv1_80: $Attribute\_domain\_Concept \in Attribute \rightarrow Concept$
inv1_81: $Attribute\_range\_DataSet \in Attribute \rightarrow DataSet$
inv1_82: $AttributeMaplet\_mapletOf\_Attribute \in AttributeMaplet \rightarrow Attribute$
inv1_83: $AttributeMaplet\_antecedent\_Individual \in AttributeMaplet \rightarrow Individual$
inv1_84: $AttributeMaplet\_image\_DataValue \in AttributeMaplet \rightarrow DataValue$
inv1_85: $\forall rm \cdot (rm \in RelationMaplet \Rightarrow Individual\_individualOf\_Concept(RelationMaplet\_antecedent\_Individual(rm)) = Relation\_domain\_Concept(RelationMaplet\_mapletOf\_Relation(rm)))$
inv1_86: $\forall rm \cdot (rm \in RelationMaplet \Rightarrow Individual\_individualOf\_Concept(RelationMaplet\_image\_Individual(rm)) = Relation\_range\_Concept(RelationMaplet\_mapletOf\_Relation(rm)))$

inv1_87: $\forall am \cdot (am \in AttributeMaplet \Rightarrow Individual\_individualOf\_Concept(AttributeMaplet\_antecedent\_Individual(am)) = Attribute\_domain\_Concept(AttributeMaplet\_mapletOf\_Attribute(am)))$

inv1_88: $\forall am \cdot (am \in AttributeMaplet \Rightarrow DataValue\_valueOf\_DataSet(AttributeMaplet\_image\_DataValue(am)) = Attribute\_range\_DataSet(AttributeMaplet\_mapletOf\_Attribute(am)))$

inv1_89: $Relation\_Type \in Relation \rightarrowtail\!\!\!\!\!\!\rightarrow Constant$

inv1_90: $Relation\_corresp\_Constant \in Relation \rightarrowtail\!\!\!\!\!\!\rightarrow Constant$

inv1_91: $Relation\_corresp\_Variable \in Relation \rightarrowtail\!\!\!\!\!\!\rightarrow Variable$

inv1_92: $\forall re \cdot (re \in dom(Relation\_Type) \Leftrightarrow (re \in dom(Relation\_corresp\_Constant) \lor (re \in dom(Relation\_corresp\_Variable))))$

inv1_93: $Attribute\_Type \in Attribute \rightarrowtail\!\!\!\!\!\!\rightarrow Constant$

inv1_94: $Attribute\_corresp\_Constant \in Attribute \rightarrowtail\!\!\!\!\!\!\rightarrow Constant$

inv1_95: $Attribute\_corresp\_Variable \in Attribute \rightarrowtail\!\!\!\!\!\!\rightarrow Variable$

inv1_96: $\forall re \cdot (re \in dom(Attribute\_Type) \Leftrightarrow (re \in dom(Attribute\_corresp\_Constant) \lor (re \in dom(Attribute\_corresp\_Variable))))$

inv1_97: $Variable\_typing\_Invariant \in Variable \rightarrowtail Invariant$

inv1_98: $Constant\_typing\_Property \in Constant \rightarrowtail Property$

inv1_99: $RelationCharacteristic\_corresp\_LogicFormula \in (Relation \nrightarrow RelationCharacteristics\_Set) \rightarrowtail\!\!\!\!\!\!\rightarrow LogicFormula$

inv1_100: $RelationMaplet\_corresp\_Constant \in RelationMaplet \rightarrowtail\!\!\!\!\!\!\rightarrow Constant$

inv1_101: $DataSet\_corresp\_Set \in DataSet \rightarrowtail\!\!\!\!\!\!\rightarrow Set$

inv1_102: $AttributeMaplet\_corresp\_Constant \in AttributeMaplet \rightarrowtail\!\!\!\!\!\!\rightarrow Constant$

inv1_103: $LogicFormula\_involves\_SetItems \in LogicFormula \nrightarrow (\mathbb{N}_1 \nrightarrow SetItem)$

inv1_104: $EnumeratedDataSet\_corresp\_EnumeratedSet \subseteq DataSet\_corresp\_Set$

inv1_105: $CustomDataSet\_corresp\_AbstractSet \subseteq DataSet\_corresp\_Set$

inv1_106: ⟨theorem⟩
$card(Concept \setminus (dom(Concept\_corresp\_AbstractSet) \cup dom(Concept\_corresp\_Constant)))$
$+ card(DataSet \setminus dom(DataSet\_corresp\_Set))$
$+ card(DataValue \setminus (dom(DataValue\_corresp\_SetItem) \cup dom(DataValue\_corresp\_Constant)))$
$+ card(Individual \setminus dom(Individual\_corresp\_Constant))$
$+ card(Concept\_isVariable^{-1}[\{TRUE\}] \setminus dom(Concept\_corresp\_Variable))$
$+ card(Relation \setminus (dom(Relation\_corresp\_Constant) \cup dom(Relation\_corresp\_Variable)))$
$+ card(Attribute \setminus (dom(Attribute\_corresp\_Constant) \cup dom(Attribute\_corresp\_Variable)))$
$+ card(RelationMaplet \setminus dom(RelationMaplet\_corresp\_Constant))$
$+ card(AttributeMaplet \setminus dom(AttributeMaplet\_corresp\_Constant)) \in \mathbb{N}$

inv1_107: $\forall xx, pxx, o\_lg \cdot (\ xx \in dom(Concept\_parentConcept\_Concept)$
$\land\ pxx = Concept\_parentConcept\_Concept(xx)$
$\land\ xx \in dom(Concept\_corresp\_Constant)$
$\land\ pxx \in dom(Concept\_corresp\_AbstractSet)$
$\land\ o\_lg = Constant\_typing\_Property(Concept\_corresp\_Constant(xx))\ )$
$\Rightarrow (\ LogicFormula\_uses\_Operators(o\_lg) = \{1 \mapsto Inclusion\_OP\}$
$\land\ (2 \mapsto Concept\_corresp\_AbstractSet(pxx)) \in LogicFormula\_involves\_Sets(o\_lg)\ )\ )$

inv1_108: $\forall o\_xx, o\_pxx, o\_lg \cdot (\ ($
$o\_xx \in dom(Constant\_typing\_Property) \cap ran(Concept\_corresp\_Constant)$
$\land\ o\_lg = Constant\_typing\_Property(o\_xx)$
$\land\ LogicFormula\_uses\_Operators(o\_lg) = \{1 \mapsto Inclusion\_OP\}$
$\land\ o\_pxx \in ran(Concept\_corresp\_AbstractSet)$
$\land\ (2 \mapsto o\_pxx) \in LogicFormula\_involves\_Sets(o\_lg)\ )$
$\Rightarrow (\ Concept\_corresp\_Constant^{-1}(o\_xx) \in dom(Concept\_parentConcept\_Concept)$
$\land\ Concept\_corresp\_AbstractSet^{-1}(o\_pxx) = Concept\_parentConcept\_Concept(Concept\_corresp\_Constant^{-1}(o\_xx))\ )\ )$

inv1_109: ⟨theorem⟩
$card(AbstractSet \setminus (ran(Concept\_corresp\_AbstractSet) \cup ran(DataSet\_corresp\_Set)))$
$+ card(EnumeratedSet \setminus ran(DataSet\_corresp\_Set))$
$+ card(dom(SetItem\_itemOf\_EnumeratedSet) \setminus ran(DataValue\_corresp\_SetItem))$
$+ card(dom(Constant\_typing\_Property) \setminus (ran(Concept\_corresp\_Constant)$
$\cup\ ran(Individual\_corresp\_Constant) \cup ran(DataValue\_corresp\_Constant) \cup ran(Relation\_corresp\_Constant)$
$\cup ran(Attribute\_corresp\_Constant) \cup ran(RelationMaplet\_corresp\_Constant) \cup ran(AttributeMaplet\_corresp\_Constant)$
$\cup\ ran(Attribute\_Type) \cup ran(Relation\_Type)))$
$+ card(dom(Variable\_typing\_Invariant) \setminus (ran(Concept\_corresp\_Variable)$
$\cup\ ran(Relation\_corresp\_Variable) \cup ran(Attribute\_corresp\_Variable))) \in \mathbb{N}$

inv1_110: $\forall xx, pxx \cdot ((xx \in dom(Concept\_parentConcept\_Concept)$
$\wedge pxx = Concept\_parentConcept\_Concept(xx)$
$\wedge pxx \notin (dom(Concept\_corresp\_AbstractSet)) \cup dom(Concept\_corresp\_Constant))$
$\Rightarrow xx \notin dom(Concept\_corresp\_Constant))$

inv1_111: $\forall o\_xx, o\_pxx, o\_lg \cdot ((o\_xx \in dom(Constant\_typing\_Property)$
$\wedge o\_lg = Constant\_typing\_Property(o\_xx)$
$\wedge LogicFormula\_uses\_Operators(o\_lg) = \{1 \mapsto Inclusion\_OP\}$
$\wedge (2 \mapsto o\_pxx) \in LogicFormula\_involves\_Sets(o\_lg)$
$\wedge o\_pxx \notin (ran(Concept\_corresp\_AbstractSet) \cup ran(DataSet\_corresp\_Set)))$
$\Rightarrow o\_xx \notin ran(Concept\_corresp\_Constant))$

inv1_112: $partition(dom(Concept\_corresp\_AbstractSet) \cup dom(Concept\_corresp\_Constant),$
$dom(Concept\_corresp\_Constant), dom(Concept\_corresp\_AbstractSet))$

VARIANT

$card(Concept \setminus (dom(Concept\_corresp\_AbstractSet) \cup dom(Concept\_corresp\_Constant)))$
$+ card(DataSet \setminus dom(DataSet\_corresp\_Set))$
$+ card(DataValue \setminus (dom(DataValue\_corresp\_SetItem) \cup dom(DataValue\_corresp\_Constant)))$
$+ card(Individual \setminus dom(Individual\_corresp\_Constant))$
$+ card(Concept\_isVariable^{-1}[\{TRUE\}] \setminus dom(Concept\_corresp\_Variable))$
$+ card(Relation \setminus (dom(Relation\_corresp\_Constant) \cup dom(Relation\_corresp\_Variable)))$
$+ card(Attribute \setminus (dom(Attribute\_corresp\_Constant) \cup dom(Attribute\_corresp\_Variable)))$
$+ card(RelationMaplet \setminus dom(RelationMaplet\_corresp\_Constant))$
$+ card(AttributeMaplet \setminus dom(AttributeMaplet\_corresp\_Constant))$
$+ \ card(AbstractSet \setminus (ran(Concept\_corresp\_AbstractSet) \cup ran(DataSet\_corresp\_Set)))$
$+ card(EnumeratedSet \setminus ran(DataSet\_corresp\_Set))$
$+ card(dom(SetItem\_itemOf\_EnumeratedSet) \setminus ran(DataValue\_corresp\_SetItem))$
$+ card(dom(Constant\_typing\_Property) \setminus (ran(Concept\_corresp\_Constant)$
$\cup ran(Individual\_corresp\_Constant) \cup ran(DataValue\_corresp\_Constant) \cup ran(Relation\_corresp\_Constant)$
$\cup ran(Attribute\_corresp\_Constant) \cup ran(RelationMaplet\_corresp\_Constant) \cup ran(AttributeMaplet\_corresp\_Constant)$
$\cup ran(Attribute\_Type) \cup ran(Relation\_Type)))$
$+ card(dom(Variable\_typing\_Invariant) \setminus (ran(Concept\_corresp\_Variable)$
$\cup ran(Relation\_corresp\_Variable) \cup ran(Attribute\_corresp\_Variable)))$

END

A.2 Definition of the Translation Rules

A.2.1 Informal Definition

In the following, we describe a set of rules that allow to obtain a formal specification from domain models associated with refinement levels of a SysML/KAOS goal model.

Table A.1 summarises 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".

Table A.1 – The translation rules

		Domain Model		B System	
	Translation Of	**Element**	**Constraint**	**Element**	**Constraint**
1	**Abstract domain model**	DM	*DM* ∈ DomainModel *DM* is not associated with a parent domain model	o_DM	*o_DM* ∈ System

A.2. Definition of the Translation Rules

2	**Domain model with parent**	DM PDM	$\{DM, PDM\} \subseteq$ DomainModel *DM* is associated with *PDM* through the *parent* association and *PDM* has already been translated	o_DM	$o\_DM \in$ Refinement *o_DM* refines *o_PDM*
3	**Abstract concept**	CO	$CO \in$ Concept *CO* is not associated with a parent concept	o_CO	$o\_CO \in$ AbstractSet
4	**Concept with parent**	CO PCO	$\{CO, PCO\} \subseteq$ Concept *CO* is associated with *PCO* through the *parentConcept* association and *PCO* has already been translated	o_CO	$o\_CO \in$ Constant **LogicFormula:** $o\_CO \subseteq o\_PCO$
5	**Relation**	RE CO1 CO2	$\{CO1, CO2\} \subseteq$ Concept $RE \in$ Relation *CO1* is the *domain* of *RE* *CO2* is the *range* of *RE* *CO1* and *CO2* have already been translated	o_RE	**IF** the *isVariable* property of *RE* is set to *FALSE* **THEN** $o\_RE \in$ Constant **ELSE** $o\_RE \in$ Variable **LogicFormula:** $o\_RE \in o\_CO1 \leftrightarrow o\_CO2$ (As usual, this relation becomes a *function*, an *injection*, ... according to the cardinalities of *RE*)
6	**Attribute**	AT CO DS	$CO \in$ Concept $DS \in$ DataSet $AT \in$ Attribute *CO* is the *domain* of *AT* *DS* is the *range* of *AT* *CO* and *DS* have already been translated	o_AT	**IF** the *isVariable* property of *AT* is set to *FALSE* **THEN** $o\_AT \in$ Constant **ELSE** $o\_AT \in$ Variable **IF** *isFunctional* and *isTotal* are set to *TRUE* **THEN LogicFormula:** $o\_AT \in o\_CO \rightarrow o\_DS$ **ELSE IF** *isFunctional* is set to *TRUE* **THEN LogicFormula:** $o\_AT \in o\_CO \nrightarrow o\_DS$ **ELSE LogicFormula:** $o\_AT \in o\_CO \leftrightarrow o\_DS$
7	**Concept changeability**	CO	$CO \in$ Concept the *isVariable* property of *CO* is set to *TRUE* *CO* has already been translated	X_CO	$X\_CO \in$ Variable **LogicFormula:** $X\_CO \subseteq o\_CO$
8	**Individual**	Ind CO	$Ind \in$ Individual $CO \in$ Concept *Ind* is an individual of *CO* *CO* has already been translated	o_Ind	**IF** $Ind \in$ VariableIndividual **THEN** $o\_Ind \in$ Variable **ELSE** $o\_Ind \in$ Constant **LogicFormula:** $o\_Ind \in o\_CO$
9	**Data value**	Dva DS	$Dva \in$ DataValue $DS \in$ DataSet *Dva* is a value of *DS* *DS* has already been translated	o_Dva	**IF** $Dva \in$ VariableDataValue **THEN** $o\_Dva \in$ Variable **ELSE** $o\_Dva \in$ Constant **LogicFormula:** $o\_Dva \in o\_DS$
10	**Relation transitivity**	RE	$RE \in$ Relation the *isTransitive* property of *RE* is set to *TRUE* *RE* has already been translated		**LogicFormula:** $(o\_RE ; o\_RE) \subseteq o\_RE$ (All other optional properties of an instance of Relation are translated in the same way (Sect. A.2.1))
11	**Relation maplets**	RE $(M_j)_{j=1..n}$ $(a_j,$ $i_j)_{j=1..n}$	$RE \in$ Relation; $(M_j)_{j=1..n}$ are *maplets* of *RE* $\forall j \in 1..n, a_j$ is the antecedent of M_j $\forall j \in 1..n, i_j$ is the image of M_j *RE* and $(a_j, i_j)_{j=1..n}$ have already been translated		**IF** the *isVariable* property of *RE* is set to *FALSE* **THEN Property:** $o\_RE = \{(o\_a_j, o\_i_j)_{j=1..n}\}$ **ELSE Initialisation:** $o\_REbcmeq\{(o\_a_j, o\_i_j)_{j=1..n}\}$
12	**Attribute maplets**	AT $(M_j)_{j=1..n}$ $(a_j,$ $i_j)_{j=1..n}$	$AT \in$ Attribute; $(M_j)_{j=1..n}$ are *maplets* of *AT* $\forall j \in 1..n, a_j$ is the antecedent of M_j $\forall j \in 1..n, i_j$ is the image of M_j *AT* and $(a_j, i_j)_{j=1..n}$ have already been translated		**IF** the *isVariable* property of *AT* is set to *FALSE* **THEN Property:** $o\_AT = \{(o\_a_j, o\_i_j)_{j=1..n}\}$ **ELSE Initialisation:** $o\_AT := \{(o\_a_j, o\_i_j)_{j=1..n}\}$

13	**Data function**	DF $(DSd_i)_{i=1..n}$ $(DSr_j)_{j=1..m}$	$DF \in$ DataFunction; $\{DSd_i\}_{i=1..n} \cup \{DSr_j\}_{j=1..m} \subseteq$ DataSet $(DSd_i)_{i=1..n}$ form the *domain* of DF $(DSr_j)_{j=1..m}$ form the *range* of DF $(DSd_i)_{i=1..n}$ and $(DSr_j)_{j=1..m}$ have already been translated	o_DF	$o\_DF \in$ Constant **LogicFormula:** $o\_DF \in (\bigtimes_{i=1}^{n} DSd_i) \nrightarrow (\bigtimes_{j=1}^{m} DSr_j)$

Generation of B System Components

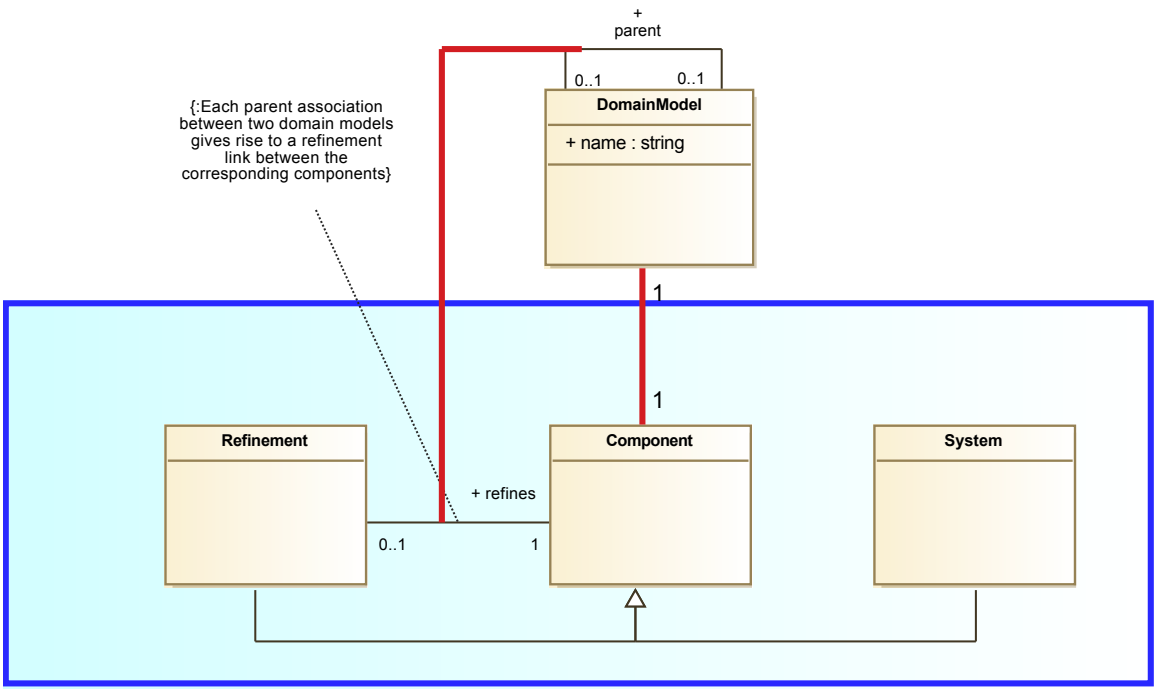


Figure A.1 – Generation of *B System* components from SysML/KAOS domain models

Any domain model that is not associated with another domain model (Fig. A.1), through the *parent* association, gives a system (line 1 of Table A.1).

A domain model associated with another one representing its parent (Fig. A.1) gives a refinement (line 2 of Table A.1). This component refines the one corresponding to the parent domain model.

Generation of B System Sets

Any concept that is not associated with another one through the parentConcept association, gives an abstract set (line 3 of Table A.1).

Any custom data set cds, defined through an enumeration (instance of EnumeratedDataSet), gives a *B System* enumerated set. Otherwise, if cds is defined with a predicate P, then it gives a constant for which the typing axiom is the result of the translation of P. Finally, cds gives an abstract set if no typing predicate is provided.

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

Generation of B System Constants

Any concept associated with another one through the parentConcept association, gives a constant typed as a subset of the *B System* element corresponding to the parent concept (line 4 of Table A.1).

Each relation having its isVariable property set to *FALSE* gives a *B System* constant (line 5 of Table A.1). The constant can be typed as a *surjection*, *injection*, etc. according to relation cardinalities [91, 124].

Similarly to relations, each attribute for which the isVariable property is set to *FALSE* gives a *B System* constant (line 6 of Table A.1). However, when the isFunctional property is set to *TRUE*, the constant type is defined as the set of functions between the *B System* element corresponding to the attribute domain and the one corresponding to the attribute range. Furthermore, when isFunctional is set to *TRUE*, the isTotal property is used to assert if the function is total (isTotal=TRUE) or partial (isTotal=FALSE).

Finally, each constant individual (resp. data value) gives a *B System* constant typed as an item of the correspondence of its concept (resp. data set) (lines 8 and 9 of Table A.1). In addition, each data function gives a *B System* constant typed as a function (line 13 of Table A.1).

Generation of B System Variables

A relation, a concept or an attribute, having its isVariable property set to *TRUE* gives a variable (Fig. A.2). For a concept, the variable represents the set of *B System* elements having this concept as type (line 7 of Table A.1). Thus, the fact that a variable concept C0 is a subconcept of another variable concept PC0 means that the set of elements that C0 can contain, over its whole existence, is included in the set of elements that PC0 can contain. However, it is possible in this approach that at some point, because of the variability of C0 and PC0, an element present in C0 is not

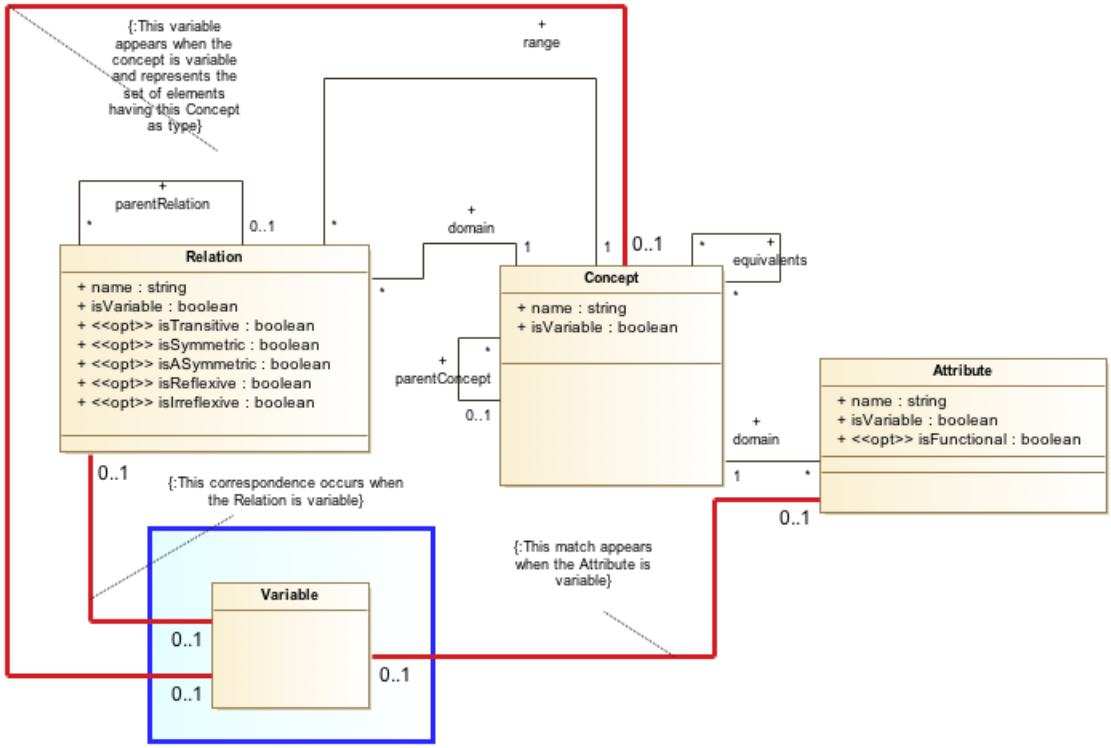


Figure A.2 – Generation of *B System* variables from SysML/KAOS concepts, relations and attributes

present in PC0. The adjusted version of the domain modeling language considers a different approach in which the inclusion of a variable concept into another one implies that at any point, elements of the variable subconcept are elements of the variable parent concept.

For a relation or an attribute, the variable represents the set of pairs between individuals (in case of relation) or between individuals and data values (in case of attribute) defined through it (lines 5 and 6 of Table A.1).

In addition, each variable individual (resp. data value) gives a *B System* variable typed as an item of the correspondence of its concept (resp. data set) (lines 8 and 9 of Table A.1). Furthermore, a substitution is added to the *initialisation* event in order to define the constant individual to whom the variable must be initialised (association initialValue of the metamodel of Fig. 4.3).

Generation of B System Invariants and Properties

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

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

A domain cardinality (respectively range cardinality) associated with a relation *re*, with bounds minCardinality and maxCardinality ($maxCardinality \geq 0$), gives the logic formula
$\forall x.(x \in ran(re) \Rightarrow card(re^{-1}[\{x\}]) \in minCardinality..maxCardinality)$ (respectively $\forall x.(x \in dom(re) \Rightarrow card(re[\{x\}]) \in minCardinality..maxCardinality)$).

When $minCardinality = maxCardinality$, then the logic formula is
$\forall x.(x \in ran(re) \Rightarrow card(re^{-1}[\{x\}]) = minCardinality)$ (respectively $\forall x.(x \in dom(re) \Rightarrow card(re[\{x\}]) = minCardinality)$).

Finally, when $maxCardinality = \infty$, then the logic formula is $\forall x.(x \in ran(re) \Rightarrow card(re^{-1}[\{x\}]) \geq minCardinality)$ (respectively $\forall x.(x \in dom(re) \Rightarrow card(re[\{x\}]) \geq minCardinality)$).

Relation maplets (respectively attribute maplets) associated with a relation (respectively attribute) *RE* give rise, in the case where the isVariable property of *RE* is set to *FALSE*, to property $RE = \{a_1 \mapsto i_1, a_2 \mapsto i_2, ..., a_j \mapsto i_j, ..., a_n \mapsto i_n\}$, where a_j designates the constant individual linked to the j-th relation maplet (respectively attribute maplet), through association antecedent, and i_j designates the constant

individual (respectively data value) linked through association image (lines 11 and 12 of Table A.1). When the isVariable property of *RE* is set to *TRUE*, it is the substitution $RE := \{a_1 \mapsto i_1, a_2 \mapsto i_2, ..., a_j \mapsto i_j, ..., a_n \mapsto i_n\}$ which is rather defined in the *INITIALISATION* clause of the *B System* component.

Finally, any predicate gives a *B System* logic formula. When the predicate is an instance of GluingInvariant, the logic formula is a *B System* gluing invariant.

A.2.2 Event-B Specification

MACHINE event_b_specs_from_ontologies
SEES EventB_Metamodel_Context,Domain_Metamodel_Context
EVENTS
Event rule_1 ⟨convergent⟩ $\widehat{=}$
correspondence of a domain model not associated to a parent domain model
any DM o_DM
where
grd0: $DomainModel \setminus (dom(DomainModel\_corresp\_Component) \cup dom(DomainModel\_parent\_DomainModel)) \neq \emptyset$
grd1: $DM \in DomainModel$
grd2: $DM \notin dom(DomainModel\_corresp\_Component)$
grd3: $DM \notin dom(DomainModel\_parent\_DomainModel)$
grd4: $Component\_Set \setminus Component \neq \emptyset$
grd5: $o\_DM \in Component\_Set$
grd6: $o\_DM \notin Component$
then
act1: $System := System \cup \{o\_DM\}$
act2: $Component := Component \cup \{o\_DM\}$
act3: $DomainModel\_corresp\_Component(DM) := o\_DM$
end
Event rule_2 ⟨convergent⟩ $\widehat{=}$
correspondence of a domain model associated to a parent domain model
any DM PDM o_DM
where
grd0: $dom(DomainModel\_parent\_DomainModel) \setminus dom(DomainModel\_corresp\_Component) \neq \emptyset$
grd1: $DM \in dom(DomainModel\_parent\_DomainModel)$
grd2: $DM \notin dom(DomainModel\_corresp\_Component)$
grd3: $dom(DomainModel\_corresp\_Component) \neq \emptyset$
grd4: $PDM \in dom(DomainModel\_corresp\_Component)$
grd5: $DomainModel\_parent\_DomainModel(DM) = PDM$
grd6: $Component\_Set \setminus Component \neq \emptyset$
grd7: $o\_DM \in Component\_Set$
grd8: $o\_DM \notin Component$
then
act1: $Refinement := Refinement \cup \{o\_DM\}$
act2: $Component := Component \cup \{o\_DM\}$
act3: $Refinement\_refines\_Component(o\_DM) := DomainModel\_corresp\_Component(PDM)$
act4: $DomainModel\_corresp\_Component(DM) := o\_DM$
end
END
MACHINE event_b_specs_from_ontologies_ref_1

A.2. Definition of the Translation Rules

REFINES event_b_specs_from_ontologies
SEES EventB_Metamodel_Context,Domain_Metamodel_Context
EVENTS
Event rule_3 ⟨convergent⟩ ≙
correspondence of a concept not associated to a parent concept
any CO o_CO
where
grd0: $Concept \setminus (dom(Concept\_parentConcept\_Concept) \cup dom(Concept\_corresp\_AbstractSet)) \neq \emptyset$
grd1: $CO \in Concept$
grd2: $CO \notin dom(Concept\_parentConcept\_Concept)$
grd3: $CO \notin (dom(Concept\_corresp\_AbstractSet) \cup dom(Concept\_corresp\_Constant))$
grd4: $Concept\_definedIn\_DomainModel(CO) \in dom(DomainModel\_corresp\_Component)$
grd5: $Set\_Set \setminus Set \neq \emptyset$
grd6: $o\_CO \in Set\_Set$
grd7: $o\_CO \notin Set$
then
act1: $AbstractSet := AbstractSet \cup \{o\_CO\}$
act2: $Set := Set \cup \{o\_CO\}$
act3: $Concept\_corresp\_AbstractSet(CO) := o\_CO$
act4: $Set\_definedIn\_Component(o\_CO) := DomainModel\_corresp\_Component(Concept\_definedIn\_DomainModel(CO))$
end
Event rule_4 ⟨convergent⟩ ≙
correspondence of an instance of EnumeratedDataSet
any EDS o_EDS elements o_elements mapping_elements_o_elements
where
grd0: $EnumeratedDataSet \setminus dom(DataSet\_corresp\_Set) \neq \emptyset$
grd1: $EDS \in EnumeratedDataSet$
grd2: $EDS \notin dom(DataSet\_corresp\_Set)$
grd4: $DataSet\_definedIn\_DomainModel(EDS) \in dom(DomainModel\_corresp\_Component)$
grd5: $Set\_Set \setminus Set \neq \emptyset$
grd6: $o\_EDS \in Set\_Set$
grd7: $o\_EDS \notin Set$
grd8: $o\_EDS \notin \{B\_NATURAL, B\_INTEGER, B\_FLOAT, B\_BOOL, B\_STRING\}$
elements
grd9: $o\_elements \subseteq SetItem\_Set$
grd10: $o\_elements \cap SetItem = \emptyset$
grd11: $elements = DataValue\_elements\_EnumeratedDataSet^{-1}[\{EDS\}]$
grd12: $card(o\_elements) = card(elements)$
grd13: $mapping\_elements\_o\_elements \in elements \rightarrowtail\!\!\!\!\!\rightarrow o\_elements$
then
act1: $EnumeratedSet := EnumeratedSet \cup \{o\_EDS\}$
act2: $Set := Set \cup \{o\_EDS\}$
act3: $EnumeratedDataSet\_corresp\_EnumeratedSet(EDS) := o\_EDS$
act4: $Set\_definedIn\_Component(o\_EDS) := DomainModel\_corresp\_Component(DataSet\_definedIn\_DomainModel(EDS))$
elements
act5: $SetItem := SetItem \cup o\_elements$
act6: $SetItem\_itemOf\_EnumeratedSet := SetItem\_itemOf\_EnumeratedSet \cup \{(xx \mapsto yy) | xx \in o\_elements \wedge yy = o\_EDS\}$
act7: $DataValue\_corresp\_SetItem := DataValue\_corresp\_SetItem \cup mapping\_elements\_o\_elements$
act8: $DataSet\_corresp\_Set := DataSet\_corresp\_Set \leftarrow \{EDS \mapsto o\_EDS\}$
end

Event rule_5 ⟨convergent⟩ ≙
correspondence of an instance of CustomDataSet which is not an instance of EnumeratedDataSet
any CS o_CS
where
grd0: $CustomDataSet \setminus (EnumeratedDataSet \cup dom(DataSet\_corresp\_Set)) \neq \emptyset$
grd1: $CS \in CustomDataSet$
grd2: $CS \notin EnumeratedDataSet$
grd3: $CS \notin dom(DataSet\_corresp\_Set)$
grd4: $DataSet\_definedIn\_DomainModel(CS) \in dom(DomainModel\_corresp\_Component)$
grd5: $Set\_Set \setminus Set \neq \emptyset$
grd6: $o\_CS \in Set\_Set$
grd7: $o\_CS \notin Set$
then
act1: $AbstractSet := AbstractSet \cup \{o\_CS\}$
act2: $Set := Set \cup \{o\_CS\}$
act3: $CustomDataSet\_corresp\_AbstractSet(CS) := o\_CS$
act4: $Set\_definedIn\_Component(o\_CS) := DomainModel\_corresp\_Component(DataSet\_definedIn\_DomainModel(CS))$
act5: $DataSet\_corresp\_Set := DataSet\_corresp\_Set \lhd \{CS \mapsto o\_CS\}$
end

Event rule_6_1 ⟨convergent⟩ ≙
correspondence of a concept associated to a parent concept (where the parent concept corresponds to an abstract set)
any CO o_CO PCO o_lg o_PCO
where
grd0: $dom(Concept\_parentConcept\_Concept) \setminus (dom(Concept\_corresp\_Constant) \cup dom(Concept\_corresp\_AbstractSet)) \neq \emptyset$
grd1: $CO \in dom(Concept\_parentConcept\_Concept) \setminus (dom(Concept\_corresp\_Constant) \cup dom(Concept\_corresp\_AbstractSet))$
grd2: $dom(Concept\_corresp\_AbstractSet) \neq \emptyset$
grd3: $PCO \in dom(Concept\_corresp\_AbstractSet)$
grd4: $Concept\_parentConcept\_Concept(CO) = PCO$
grd5: $Concept\_definedIn\_DomainModel(CO) \in dom(DomainModel\_corresp\_Component)$
grd6: $Constant\_Set \setminus Constant \neq \emptyset$
grd7: $o\_CO \in Constant\_Set \setminus Constant$
grd8: $LogicFormula\_Set \setminus LogicFormula \neq \emptyset$
grd9: $o\_lg \in LogicFormula\_Set \setminus LogicFormula$
grd10: $o\_PCO \in AbstractSet$
grd11: $o\_PCO = Concept\_corresp\_AbstractSet(PCO)$
then
act1: $Constant := Constant \cup \{o\_CO\}$
act2: $Concept\_corresp\_Constant(CO) := o\_CO$
act3: $Constant\_definedIn\_Component(o\_CO) := DomainModel\_corresp\_Component(Concept\_definedIn\_DomainModel(CO))$
act4: $Property := Property \cup \{o\_lg\}$
act5: $LogicFormula := LogicFormula \cup \{o\_lg\}$
act6: $LogicFormula\_uses\_Operators(o\_lg) := \{1 \mapsto Inclusion\_OP\}$
act7: $Constant\_isInvolvedIn\_LogicFormulas(o\_CO) := \{1 \mapsto o\_lg\}$
act8: $LogicFormula\_involves\_Sets(o\_lg) := \{2 \mapsto o\_PCO\}$
act9: $LogicFormula\_definedIn\_Component(o\_lg) := DomainModel\_corresp\_Component(Concept\_definedIn\_DomainModel(CO))$
act10: $Constant\_typing\_Property(o\_CO) := o\_lg$

end

Event rule_6_2 ⟨convergent⟩ ≘
correspondence of a concept associated to a parent concept (where the parent concept corresponds to a constant)
any CO o_CO PCO o_lg o_PCO
where

grd0: $dom(Concept\_parentConcept\_Concept) \setminus (dom(Concept\_corresp\_Constant) \cup dom(Concept\_corresp\_AbstractSet)) \neq \emptyset$
grd1: $CO \in dom(Concept\_parentConcept\_Concept) \setminus (dom(Concept\_corresp\_Constant) \cup dom(Concept\_corresp\_AbstractSet))$

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

then

act1: $Constant := Constant \cup \{o\_CO\}$
act2: $Concept\_corresp\_Constant(CO) := o\_CO$
act3: $Constant\_definedIn\_Component(o\_CO) := DomainModel\_corresp\_Component(Concept\_definedIn\_DomainModel(CO))$

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

act10: $Constant\_typing\_Property(o\_CO) := o\_lg$

end

Event rule_7_1 ⟨convergent⟩ ≘
correspondence of an instance of Individual (where the concept corresponds to an abstract set)
any ind o_ind CO o_lg o_CO
where

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

then

act1: $Constant := Constant \cup \{o\_ind\}$

act2: $Individual\_corresp\_Constant(ind) := o\_ind$
act3: $Constant\_definedIn\_Component(o\_ind) := DomainModel\_corresp\_Component(Concept\_definedIn\_DomainModel(CO))$

act4: $Property := Property \cup \{o\_lg\}$
act5: $LogicFormula := LogicFormula \cup \{o\_lg\}$
act6: $LogicFormula\_uses\_Operators(o\_lg) := \{1 \mapsto Belonging\_OP\}$
act7: $Constant\_isInvolvedIn\_LogicFormulas(o\_ind) := \{1 \mapsto o\_lg\}$
act8: $LogicFormula\_involves\_Sets(o\_lg) := \{2 \mapsto o\_CO\}$
act9: $LogicFormula\_definedIn\_Component(o\_lg) := DomainModel\_corresp\_Component(Concept\_definedIn\_DomainModel(CO))$

act10: $Constant\_typing\_Property(o\_ind) := o\_lg$

end

Event rule_7_2 ⟨convergent⟩ ≘
correspondence of an instance of Individual (where the concept corresponds to a constant)
any ind o_ind CO o_lg o_CO
where

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

then

act1: $Constant := Constant \cup \{o\_ind\}$
act2: $Individual\_corresp\_Constant(ind) := o\_ind$
act3: $Constant\_definedIn\_Component(o\_ind) := DomainModel\_corresp\_Component(Concept\_definedIn\_DomainModel(CO))$

act4: $Property := Property \cup \{o\_lg\}$
act5: $LogicFormula := LogicFormula \cup \{o\_lg\}$
act6: $LogicFormula\_uses\_Operators(o\_lg) := \{1 \mapsto Belonging\_OP\}$
act7: $Constant\_isInvolvedIn\_LogicFormulas := Constant\_isInvolvedIn\_LogicFormulas \Leftarrow \{(o\_ind \mapsto \{1 \mapsto o\_lg\}), o\_CO \mapsto Constant\_isInvolvedIn\_LogicFormulas(o\_CO) \cup \{2 \mapsto o\_lg\}\}$
act8: $LogicFormula\_involves\_Sets(o\_lg) := \emptyset$
act9: $LogicFormula\_definedIn\_Component(o\_lg) := DomainModel\_corresp\_Component(Concept\_definedIn\_DomainModel(CO))$

act10: $Constant\_typing\_Property(o\_ind) := o\_lg$

end

Event rule_8 ⟨convergent⟩ ≘
correspondence of an instance of DataValue (When the data set is an instance of CustomDataSet not instance of EnumeratedDataSet
(for this last case, the rule for instances of EnumeratedDataSet also handles data values))
any dva o_dva DS o_lg o_DS
where

grd0: $dom(DataValue\_valueOf\_DataSet) \setminus (dom(DataValue\_corresp\_Constant) \cup dom(DataValue\_corresp\_SetItem)) \neq \emptyset$
grd1: $dva \in dom(DataValue\_valueOf\_DataSet) \setminus (dom(DataValue\_corresp\_Constant) \cup dom(DataValue\_corresp\_SetItem))$

grd2: $dom(CustomDataSet\_corresp\_AbstractSet) \neq \emptyset$
grd3: $DS \in dom(CustomDataSet\_corresp\_AbstractSet)$
grd4: $DataValue\_valueOf\_DataSet(dva) = DS$
grd5: $DataSet\_definedIn\_DomainModel(DS) \in dom(DomainModel\_corresp\_Component)$
grd6: $Constant\_Set \setminus Constant \neq \emptyset$
grd7: $o\_dva \in Constant\_Set \setminus Constant$
grd8: $LogicFormula\_Set \setminus LogicFormula \neq \emptyset$
grd9: $o\_lg \in LogicFormula\_Set \setminus LogicFormula$
grd10: $o\_DS \in AbstractSet$
grd11: $o\_DS = CustomDataSet\_corresp\_AbstractSet(DS)$

then

act1: $Constant := Constant \cup \{o\_dva\}$
act2: $DataValue\_corresp\_Constant(dva) := o\_dva$
act3: $Constant\_definedIn\_Component(o\_dva) := DomainModel\_corresp\_Component(DataSet\_definedIn\_DomainModel(DS))$

act4: $Property := Property \cup \{o\_lg\}$
act5: $LogicFormula := LogicFormula \cup \{o\_lg\}$
act6: $LogicFormula\_uses\_Operators(o\_lg) := \{1 \mapsto Belonging\_OP\}$
act7: $Constant\_isInvolvedIn\_LogicFormulas(o\_dva) := \{1 \mapsto o\_lg\}$
act8: $LogicFormula\_involves\_Sets(o\_lg) := \{2 \mapsto o\_DS\}$
act9: $LogicFormula\_definedIn\_Component(o\_lg) := DomainModel\_corresp\_Component(DataSet\_definedIn\_DomainModel(DS))$

act10: $Constant\_typing\_Property(o\_dva) := o\_lg$

end

Event rule_9_1 ⟨convergent⟩ $\widehat{=}$
handling the variability of a concept and initializing the associated variable (when the concept corresponds to an abstract set)

any CO x_CO o_lg o_CO o_ia o_inds bij_o_inds

where

grd0: $(dom(Concept\_corresp\_AbstractSet) \cap Concept\_isVariable^{-1}[\{TRUE\}]) \setminus dom(Concept\_corresp\_Variable) \neq \emptyset$
grd1: $CO \in (dom(Concept\_corresp\_AbstractSet) \cap Concept\_isVariable^{-1}[\{TRUE\}]) \setminus dom(Concept\_corresp\_Variable)$

grd2: $Concept\_definedIn\_DomainModel(CO) \in dom(DomainModel\_corresp\_Component)$
grd3: $Individual\_individualOf\_Concept^{-1}[\{CO\}] \subseteq dom(Individual\_corresp\_Constant)$
grd4: $Variable\_Set \setminus Variable \neq \emptyset$
grd5: $x\_CO \in Variable\_Set \setminus Variable$
grd6: $LogicFormula\_Set \setminus LogicFormula \neq \emptyset$
grd7: $o\_lg \in LogicFormula\_Set \setminus LogicFormula$
grd8: $o\_CO \in AbstractSet$
grd9: $o\_CO = Concept\_corresp\_AbstractSet(CO)$
grd10: $InitialisationAction\_Set \setminus InitialisationAction \neq \emptyset$
grd11: $o\_ia \in InitialisationAction\_Set \setminus InitialisationAction$
grd12: $o\_inds = Individual\_corresp\_Constant[Individual\_individualOf\_Concept^{-1}[\{CO\}]]$
grd13: $finite(o\_inds)$
grd14: $bij\_o\_inds \in 1 \mathinner{..} card(o\_inds) \rightarrowtail\!\!\!\!\!\!\rightarrow o\_inds$

then

act1: $Variable := Variable \cup \{x\_CO\}$
act2: $Concept\_corresp\_Variable(CO) := x\_CO$
act3: $Variable\_definedIn\_Component(x\_CO) := DomainModel\_corresp\_Component(Concept\_definedIn\_DomainModel(CO))$

act4: $Invariant := Invariant \cup \{o\_lg\}$
act5: $LogicFormula := LogicFormula \cup \{o\_lg\}$

act6: $LogicFormula\_uses\_Operators(o\_lg) := \{1 \mapsto Inclusion\_OP\}$
act7: $Invariant\_involves\_Variables(o\_lg) := \{1 \mapsto x\_CO\}$
act8: $LogicFormula\_involves\_Sets(o\_lg) := \{2 \mapsto o\_CO\}$
act9: $LogicFormula\_definedIn\_Component(o\_lg) := DomainModel\_corresp\_Component(Concept\_definedIn\_DomainModel(CO))$

act10: $InitialisationAction := InitialisationAction \cup \{o\_ia\}$
act11: $InitialisationAction\_uses\_Operators(o\_ia) := \{1 \mapsto BecomeEqual2SetOf\_OP\}$
act12: $Variable\_init\_InitialisationAction(x\_CO) := o\_ia$
act13: $InitialisationAction\_involves\_Constants(o\_ia) := bij\_o\_inds$
act14: $Variable\_typing\_Invariant(x\_CO) := o\_lg$

end

Event rule_9_2 ⟨convergent⟩ ≙
handling the variability of a concept and initializing the associated variable (when the concept corresponds to a constant)

any CO x_CO o_lg o_CO o_ia o_inds bij_o_inds

where

grd0: $(dom(Concept\_corresp\_Constant) \cap Concept\_isVariable^{-1}[\{TRUE\}]) \setminus dom(Concept\_corresp\_Variable) \neq \emptyset$
grd1: $CO \in (dom(Concept\_corresp\_Constant) \cap Concept\_isVariable^{-1}[\{TRUE\}]) \setminus dom(Concept\_corresp\_Variable)$
grd2: $Concept\_definedIn\_DomainModel(CO) \in dom(DomainModel\_corresp\_Component)$
grd3: $Individual\_individualOf\_Concept^{-1}[\{CO\}] \subseteq dom(Individual\_corresp\_Constant)$
grd4: $Variable\_Set \setminus Variable \neq \emptyset$
grd5: $x\_CO \in Variable\_Set \setminus Variable$
grd6: $LogicFormula\_Set \setminus LogicFormula \neq \emptyset$
grd7: $o\_lg \in LogicFormula\_Set \setminus LogicFormula$
grd8: $o\_CO \in Constant$
grd9: $o\_CO = Concept\_corresp\_Constant(CO)$
grd10: $InitialisationAction\_Set \setminus InitialisationAction \neq \emptyset$
grd11: $o\_ia \in InitialisationAction\_Set \setminus InitialisationAction$
grd12: $o\_inds = Individual\_corresp\_Constant[Individual\_individualOf\_Concept^{-1}[\{CO\}]]$
grd13: $finite(o\_inds)$
grd14: $bij\_o\_inds \in 1 \mathinner{..} card(o\_inds) \rightarrowtail\!\!\!\!\!\!\rightarrow o\_inds$

then

act1: $Variable := Variable \cup \{x\_CO\}$
act2: $Concept\_corresp\_Variable(CO) := x\_CO$
act3: $Variable\_definedIn\_Component(x\_CO) := DomainModel\_corresp\_Component(Concept\_definedIn\_DomainModel(CO))$

act4: $Invariant := Invariant \cup \{o\_lg\}$
act5: $LogicFormula := LogicFormula \cup \{o\_lg\}$
act6: $LogicFormula\_uses\_Operators(o\_lg) := \{1 \mapsto Inclusion\_OP\}$
act7: $Invariant\_involves\_Variables(o\_lg) := \{1 \mapsto x\_CO\}$
act8: $Constant\_isInvolvedIn\_LogicFormulas(o\_CO) := Constant\_isInvolvedIn\_LogicFormulas(o\_CO) \cup \{2 \mapsto o\_lg\}$

act9: $LogicFormula\_involves\_Sets(o\_lg) := \emptyset$
act10: $LogicFormula\_definedIn\_Component(o\_lg) := DomainModel\_corresp\_Component(Concept\_definedIn\_DomainModel(CO))$

act11: $InitialisationAction := InitialisationAction \cup \{o\_ia\}$
act12: $InitialisationAction\_uses\_Operators(o\_ia) := \{1 \mapsto BecomeEqual2SetOf\_OP\}$
act13: $Variable\_init\_InitialisationAction(x\_CO) := o\_ia$
act14: $InitialisationAction\_involves\_Constants(o\_ia) := bij\_o\_inds$
act15: $Variable\_typing\_Invariant(x\_CO) := o\_lg$

end

Event rule_10_1 ⟨convergent⟩ ≙
correspondence of an instance of Relation having its isVariable property set to false (case where domain and range correspond to abstract sets)
any RE T_RE o_RE CO1 o_CO1 CO2 o_CO2 o_lg1 o_lg2 DM
where
grd0: $Relation\_isVariable^{-1}[\{FALSE\}] \setminus dom(Relation\_Type) \neq \emptyset$
grd1: $RE \in Relation\_isVariable^{-1}[\{FALSE\}] \setminus dom(Relation\_Type)$
grd2: $dom(Concept\_corresp\_AbstractSet) \neq \emptyset$
grd3: $CO1 = Relation\_domain\_Concept(RE)$
grd4: $CO2 = Relation\_range\_Concept(RE)$
grd5: $\{CO1, CO2\} \subseteq dom(Concept\_corresp\_AbstractSet)$
grd6: $Relation\_definedIn\_DomainModel(RE) \in dom(DomainModel\_corresp\_Component)$
grd7: $Constant\_Set \setminus Constant \neq \emptyset$
grd8: $\{T\_RE, o\_RE\} \subseteq Constant\_Set \setminus Constant$
grd9: $LogicFormula\_Set \setminus LogicFormula \neq \emptyset$
grd10: $\{o\_lg1, o\_lg2\} \subseteq LogicFormula\_Set \setminus LogicFormula$
grd11: $o\_CO1 = Concept\_corresp\_AbstractSet(CO1)$
grd12: $o\_CO2 = Concept\_corresp\_AbstractSet(CO2)$
grd13: $DM = Relation\_definedIn\_DomainModel(RE)$
grd14: $T\_RE \neq o\_RE$
grd15: $o\_lg1 \neq o\_lg2$
then
act1: $Constant := Constant \cup \{T\_RE, o\_RE\}$
act2: $Relation\_Type(RE) := T\_RE$
act3: $Relation\_corresp\_Constant(RE) := o\_RE$
act4: $Constant\_definedIn\_Component := Constant\_definedIn\_Component \cup \{o\_RE \mapsto DomainModel\_corresp\_Component(DM), T\_RE \mapsto DomainModel\_corresp\_Component(DM)\}$
act5: $Property := Property \cup \{o\_lg1, o\_lg2\}$
act6: $LogicFormula := LogicFormula \cup \{o\_lg1, o\_lg2\}$
act7: $Constant\_typing\_Property := Constant\_typing\_Property \cup \{T\_RE \mapsto o\_lg1, o\_RE \mapsto o\_lg2\}$
act8: $Constant\_isInvolvedIn\_LogicFormulas := Constant\_isInvolvedIn\_LogicFormulas \cup \{T\_RE \mapsto \{1 \mapsto o\_lg1, 2 \mapsto o\_lg2\}, o\_RE \mapsto \{1 \mapsto o\_lg2\}\}$
act9: $LogicFormula\_uses\_Operators := LogicFormula\_uses\_Operators \cup \{o\_lg1 \mapsto \{1 \mapsto RelationSet\_OP\}, o\_lg2 \mapsto \{1 \mapsto Belonging\_OP\}\}$
act10: $LogicFormula\_involves\_Sets := LogicFormula\_involves\_Sets \cup \{o\_lg1 \mapsto \{2 \mapsto o\_CO1, 3 \mapsto o\_CO2\}, o\_lg2 \mapsto \emptyset\}$
act11: $LogicFormula\_definedIn\_Component := LogicFormula\_definedIn\_Component \cup \{o\_lg1 \mapsto DomainModel\_corresp\_Component(DM), o\_lg2 \mapsto DomainModel\_corresp\_Component(DM)\}$
end

Event rule_10_2 ⟨convergent⟩ ≙
correspondence of an instance of Relation having its isVariable property set to false (case where domain corresponds to an abstract set and range corresponds to a constant)
any RE T_RE o_RE CO1 o_CO1 CO2 o_CO2 o_lg1 o_lg2 DM
where
grd0: $Relation\_isVariable^{-1}[\{FALSE\}] \setminus dom(Relation\_Type) \neq \emptyset$
grd1: $RE \in Relation\_isVariable^{-1}[\{FALSE\}] \setminus dom(Relation\_Type)$
grd2: $dom(Concept\_corresp\_AbstractSet) \neq \emptyset$
grd3: $CO1 = Relation\_domain\_Concept(RE)$
grd4: $CO1 \in dom(Concept\_corresp\_AbstractSet)$
grd5: $dom(Concept\_corresp\_Constant) \neq \emptyset$
grd6: $CO2 = Relation\_range\_Concept(RE)$
grd7: $CO2 \in dom(Concept\_corresp\_Constant)$
grd8: $Relation\_definedIn\_DomainModel(RE) \in dom(DomainModel\_corresp\_Component)$

grd9: $Constant\_Set \setminus Constant \neq \emptyset$
grd10: $\{T\_RE, o\_RE\} \subseteq Constant\_Set \setminus Constant$
grd11: $LogicFormula\_Set \setminus LogicFormula \neq \emptyset$
grd12: $\{o\_lg1, o\_lg2\} \subseteq LogicFormula\_Set \setminus LogicFormula$
grd13: $o\_CO1 = Concept\_corresp\_AbstractSet(CO1)$
grd14: $o\_CO2 = Concept\_corresp\_Constant(CO2)$
grd15: $DM = Relation\_definedIn\_DomainModel(RE)$
grd16: $T\_RE \neq o\_RE$
grd17: $o\_lg1 \neq o\_lg2$

then

act1: $Constant := Constant \cup \{T\_RE, o\_RE\}$
act2: $Relation\_Type(RE) := T\_RE$
act3: $Relation\_corresp\_Constant(RE) := o\_RE$
act4: $Constant\_definedIn\_Component := Constant\_definedIn\_Component \cup \{o\_RE \mapsto DomainModel\_corresp\_Component(DM), T\_RE \mapsto DomainModel\_corresp\_Component(DM)\}$
act5: $Property := Property \cup \{o\_lg1, o\_lg2\}$
act6: $LogicFormula := LogicFormula \cup \{o\_lg1, o\_lg2\}$
act7: $Constant\_typing\_Property := Constant\_typing\_Property \cup \{T\_RE \mapsto o\_lg1, o\_RE \mapsto o\_lg2\}$
act8: $Constant\_isInvolvedIn\_LogicFormulas := Constant\_isInvolvedIn\_LogicFormulas \Leftarrow \{T\_RE \mapsto \{1 \mapsto o\_lg1, 2 \mapsto o\_lg2\}, o\_RE \mapsto \{1 \mapsto o\_lg2\}, o\_CO2 \mapsto \{3 \mapsto o\_lg1\} \cup Constant\_isInvolvedIn\_LogicFormulas(o\_CO2)\}$
act9: $LogicFormula\_uses\_Operators := LogicFormula\_uses\_Operators \cup \{o\_lg1 \mapsto \{1 \mapsto RelationSet\_OP\}, o\_lg2 \mapsto \{2 \mapsto Belonging\_OP\}\}$
act10: $LogicFormula\_involves\_Sets := LogicFormula\_involves\_Sets \cup \{o\_lg1 \mapsto \{2 \mapsto o\_CO1\}, o\_lg2 \mapsto \emptyset\}$
act11: $LogicFormula\_definedIn\_Component := LogicFormula\_definedIn\_Component \cup \{o\_lg1 \mapsto DomainModel\_corresp\_Component(DM), o\_lg2 \mapsto DomainModel\_corresp\_Component(DM)\}$

end

Event rule_10_3 ⟨convergent⟩ $\widehat{=}$
correspondence of an instance of Relation having its isVariable property set to false (case where range corresponds to an abstract set and domain corresponds to a constant)

any RE T_RE o_RE CO1 o_CO1 CO2 o_CO2 o_lg1 o_lg2 DM
where

grd0: $Relation\_isVariable^{-1}[\{FALSE\}] \setminus dom(Relation\_Type) \neq \emptyset$
grd1: $RE \in Relation\_isVariable^{-1}[\{FALSE\}] \setminus dom(Relation\_Type)$
grd2: $dom(Concept\_corresp\_Constant) \neq \emptyset$
grd3: $CO1 = Relation\_domain\_Concept(RE)$
grd4: $CO1 \in dom(Concept\_corresp\_Constant)$
grd5: $dom(Concept\_corresp\_AbstractSet) \neq \emptyset$
grd6: $CO2 = Relation\_range\_Concept(RE)$
grd7: $CO2 \in dom(Concept\_corresp\_AbstractSet)$
grd8: $Relation\_definedIn\_DomainModel(RE) \in dom(DomainModel\_corresp\_Component)$
grd9: $Constant\_Set \setminus Constant \neq \emptyset$
grd10: $\{T\_RE, o\_RE\} \subseteq Constant\_Set \setminus Constant$
grd11: $LogicFormula\_Set \setminus LogicFormula \neq \emptyset$
grd12: $\{o\_lg1, o\_lg2\} \subseteq LogicFormula\_Set \setminus LogicFormula$
grd13: $o\_CO2 = Concept\_corresp\_AbstractSet(CO2)$
grd14: $o\_CO1 = Concept\_corresp\_Constant(CO1)$
grd15: $DM = Relation\_definedIn\_DomainModel(RE)$
grd16: $T\_RE \neq o\_RE$
grd17: $o\_lg1 \neq o\_lg2$

then

act1: $Constant := Constant \cup \{T\_RE, o\_RE\}$
act2: $Relation\_Type(RE) := T\_RE$

act3: $Relation\_corresp\_Constant(RE) := o\_RE$

act4: $Constant\_definedIn\_Component := Constant\_definedIn\_Component \cup \{o\_RE \mapsto DomainModel\_corresp\_Component(DM), T\_RE \mapsto DomainModel\_corresp\_Component(DM)\}$

act5: $Property := Property \cup \{o\_lg1, o\_lg2\}$

act6: $LogicFormula := LogicFormula \cup \{o\_lg1, o\_lg2\}$

act7: $Constant\_typing\_Property := Constant\_typing\_Property \cup \{T\_RE \mapsto o\_lg1, o\_RE \mapsto o\_lg2\}$

act8: $Constant\_isInvolvedIn\_LogicFormulas := Constant\_isInvolvedIn\_LogicFormulas \Leftarrow \{T\_RE \mapsto \{1 \mapsto o\_lg1, 2 \mapsto o\_lg2\}, o\_RE \mapsto \{1 \mapsto o\_lg2\}, o\_CO1 \mapsto \{2 \mapsto o\_lg1\} \cup Constant\_isInvolvedIn\_LogicFormulas(o\_CO1)\}$

act9: $LogicFormula\_uses\_Operators := LogicFormula\_uses\_Operators \cup \{o\_lg1 \mapsto \{1 \mapsto RelationSet\_OP\}, o\_lg2 \mapsto \{1 \mapsto Belonging\_OP\}\}$

act10: $LogicFormula\_involves\_Sets := LogicFormula\_involves\_Sets \cup \{o\_lg1 \mapsto \{3 \mapsto o\_CO2\}, o\_lg2 \mapsto \emptyset\}$

act11: $LogicFormula\_definedIn\_Component := LogicFormula\_definedIn\_Component \cup \{o\_lg1 \mapsto DomainModel\_corresp\_Component(DM), o\_lg2 \mapsto DomainModel\_corresp\_Component(DM)\}$

end

Event rule_10_4 ⟨convergent⟩ $\widehat{=}$

correspondence of an instance of Relation having its isVariable property set to false (case where domain and range correspond to constants)

any RE T_RE o_RE CO1 o_CO1 CO2 o_CO2 o_lg1 o_lg2 DM

where

grd0: $Relation\_isVariable^{-1}[\{FALSE\}] \setminus dom(Relation\_Type) \neq \emptyset$

grd1: $RE \in Relation\_isVariable^{-1}[\{FALSE\}] \setminus dom(Relation\_Type)$

grd2: $dom(Concept\_corresp\_Constant) \neq \emptyset$

grd3: $CO1 = Relation\_domain\_Concept(RE)$

grd4: $CO2 = Relation\_range\_Concept(RE)$

grd5: $\{CO1, CO2\} \subseteq dom(Concept\_corresp\_Constant)$

grd6: $Relation\_definedIn\_DomainModel(RE) \in dom(DomainModel\_corresp\_Component)$

grd7: $Constant\_Set \setminus Constant \neq \emptyset$

grd8: $\{T\_RE, o\_RE\} \subseteq Constant\_Set \setminus Constant$

grd9: $LogicFormula\_Set \setminus LogicFormula \neq \emptyset$

grd10: $\{o\_lg1, o\_lg2\} \subseteq LogicFormula\_Set \setminus LogicFormula$

grd11: $o\_CO1 = Concept\_corresp\_Constant(CO1)$

grd12: $o\_CO2 = Concept\_corresp\_Constant(CO2)$

grd13: $DM = Relation\_definedIn\_DomainModel(RE)$

grd14: $T\_RE \neq o\_RE$

grd15: $o\_lg1 \neq o\_lg2$

grd16: $o\_CO1 \neq o\_CO2$

then

act1: $Constant := Constant \cup \{T\_RE, o\_RE\}$

act2: $Relation\_Type(RE) := T\_RE$

act3: $Relation\_corresp\_Constant(RE) := o\_RE$

act4: $Constant\_definedIn\_Component := Constant\_definedIn\_Component \cup \{o\_RE \mapsto DomainModel\_corresp\_Component(DM), T\_RE \mapsto DomainModel\_corresp\_Component(DM)\}$

act5: $Property := Property \cup \{o\_lg1, o\_lg2\}$

act6: $LogicFormula := LogicFormula \cup \{o\_lg1, o\_lg2\}$

act7: $Constant\_typing\_Property := Constant\_typing\_Property \cup \{T\_RE \mapsto o\_lg1, o\_RE \mapsto o\_lg2\}$

act8: $Constant\_isInvolvedIn\_LogicFormulas := Constant\_isInvolvedIn\_LogicFormulas \Leftarrow \{T\_RE \mapsto \{1 \mapsto o\_lg1, 2 \mapsto o\_lg2\}, o\_RE \mapsto \{1 \mapsto o\_lg2\}, o\_CO1 \mapsto \{2 \mapsto o\_lg1\} \cup Constant\_isInvolvedIn\_LogicFormulas(o\_CO1), o\_CO2 \mapsto \{3 \mapsto o\_lg1\} \cup Constant\_isInvolvedIn\_LogicFormulas(o\_CO2)\}$

act9: $LogicFormula\_uses\_Operators := LogicFormula\_uses\_Operators \cup \{o\_lg1 \mapsto \{1 \mapsto RelationSet\_OP\}, o\_lg2 \mapsto \{1 \mapsto Belonging\_OP\}\}$

act10: $LogicFormula\_involves\_Sets := LogicFormula\_involves\_Sets \cup \{o\_lg1 \mapsto \emptyset, o\_lg2 \mapsto \emptyset\}$

act11: $LogicFormula\_definedIn\_Component := LogicFormula\_definedIn\_Component \cup \{o\_lg1 \mapsto DomainModel\_corresp\_Component(DM), o\_lg2 \mapsto DomainModel\_corresp\_Component(DM)\}$

end

Event rule_11_1 ⟨convergent⟩ ≘
correspondence of an instance of RelationMaplet

any remap o_remap RE antecedent image o_lg o_antecedent o_image

where

grd0: $RelationMaplet \setminus dom(RelationMaplet\_corresp\_Constant) \neq \emptyset$
grd1: $remap \in RelationMaplet \setminus dom(RelationMaplet\_corresp\_Constant)$
grd2: $dom(Relation\_corresp\_Constant) \cup dom(Relation\_corresp\_Variable) \neq \emptyset$
grd3: $RelationMaplet\_mapletOf\_Relation(remap) = RE$
grd4: $RE \in dom(Relation\_corresp\_Constant) \cup dom(Relation\_corresp\_Variable)$
grd5: $Relation\_definedIn\_DomainModel(RE) \in dom(DomainModel\_corresp\_Component)$
grd6: $Constant\_Set \setminus Constant \neq \emptyset$
grd7: $o\_remap \in Constant\_Set \setminus Constant$
grd8: $LogicFormula\_Set \setminus LogicFormula \neq \emptyset$
grd9: $o\_lg \in LogicFormula\_Set \setminus LogicFormula$
grd10: $antecedent = RelationMaplet\_antecedent\_Individual(remap)$
grd11: $image = RelationMaplet\_image\_Individual(remap)$
grd12: $\{antecedent, image\} \subseteq dom(Individual\_corresp\_Constant)$
grd13: $o\_antecedent = Individual\_corresp\_Constant(antecedent)$
grd14: $o\_image = Individual\_corresp\_Constant(image)$
grd15: $o\_antecedent \neq o\_image$
then, for each relation already treated for which all the maplets have been processed,
if it is variable, we generate the initialization, otherwise, we generate the closure property,
knowing that the maplets give rise to variables in case of variable relation and constants
in case of constant relationship

then

act1: $Constant := Constant \cup \{o\_remap\}$
act2: $RelationMaplet\_corresp\_Constant(remap) := o\_remap$
act3: $Constant\_definedIn\_Component(o\_remap) := DomainModel\_corresp\_Component(Relation\_definedIn\_DomainModel(RE))$

act4: $Property := Property \cup \{o\_lg\}$
act5: $LogicFormula := LogicFormula \cup \{o\_lg\}$
act6: $LogicFormula\_uses\_Operators(o\_lg) := \{1 \mapsto Maplet\_OP\}$
act7: $Constant\_isInvolvedIn\_LogicFormulas := Constant\_isInvolvedIn\_LogicFormulas \Leftarrow \{o\_remap \mapsto \{1 \mapsto o\_lg\}, o\_antecedent \mapsto \{2 \mapsto o\_lg\} \cup Constant\_isInvolvedIn\_LogicFormulas(o\_antecedent), o\_image \mapsto \{3 \mapsto o\_lg\} \cup Constant\_isInvolvedIn\_LogicFormulas(o\_image)\}$
act8: $LogicFormula\_involves\_Sets(o\_lg) := \emptyset$
act9: $LogicFormula\_definedIn\_Component(o\_lg) := DomainModel\_corresp\_Component(Relation\_definedIn\_DomainModel(RE))$

act10: $Constant\_typing\_Property(o\_remap) := o\_lg$

end

Event rule_11_2_1 ⟨convergent⟩ ≘
correspondence of an instance of AttributeMaplet (case where the image (of type DataValue) corresponds to a constant (it can also corresponds to a set item)

any atmap o_atmap AT antecedent image o_lg o_antecedent o_image

where

grd0: $AttributeMaplet \setminus dom(AttributeMaplet\_corresp\_Constant) \neq \emptyset$
grd1: $atmap \in AttributeMaplet \setminus dom(AttributeMaplet\_corresp\_Constant)$
grd2: $dom(Attribute\_corresp\_Constant) \cup dom(Attribute\_corresp\_Variable) \neq \emptyset$
grd3: $AttributeMaplet\_mapletOf\_Attribute(atmap) = AT$
grd4: $AT \in dom(Attribute\_corresp\_Constant) \cup dom(Attribute\_corresp\_Variable)$
grd5: $Attribute\_definedIn\_DomainModel(AT) \in dom(DomainModel\_corresp\_Component)$
grd6: $Constant\_Set \setminus Constant \neq \emptyset$

grd7: $o\_atmap \in Constant\_Set \setminus Constant$
grd8: $LogicFormula\_Set \setminus LogicFormula \neq \emptyset$
grd9: $o\_lg \in LogicFormula\_Set \setminus LogicFormula$
grd10: $antecedent = AttributeMaplet\_antecedent\_Individual(atmap)$
grd11: $image = AttributeMaplet\_image\_DataValue(atmap)$
grd12: $antecedent \in dom(Individual\_corresp\_Constant)$
grd13: $image \in dom(DataValue\_corresp\_Constant)$
grd14: $o\_antecedent = Individual\_corresp\_Constant(antecedent)$
grd15: $o\_image = DataValue\_corresp\_Constant(image)$
grd16: $o\_antecedent \neq o\_image$

then

act1: $Constant := Constant \cup \{o\_atmap\}$
act2: $AttributeMaplet\_corresp\_Constant(atmap) := o\_atmap$
act3: $Constant\_definedIn\_Component(o\_atmap) := DomainModel\_corresp\_Component(Attribute\_definedIn\_DomainModel(AT))$

act4: $Property := Property \cup \{o\_lg\}$
act5: $LogicFormula := LogicFormula \cup \{o\_lg\}$
act6: $LogicFormula\_uses\_Operators(o\_lg) := \{1 \mapsto Maplet\_OP\}$
act7: $Constant\_isInvolvedIn\_LogicFormulas := Constant\_isInvolvedIn\_LogicFormulas \Leftarrow \{o\_atmap \mapsto \{1 \mapsto o\_lg\}, o\_antecedent \mapsto \{2 \mapsto o\_lg\} \cup Constant\_isInvolvedIn\_LogicFormulas(o\_antecedent), o\_image \mapsto \{3 \mapsto o\_lg\} \cup Constant\_isInvolvedIn\_LogicFormulas(o\_image)\}$
act8: $LogicFormula\_involves\_Sets(o\_lg) := \emptyset$
act9: $LogicFormula\_definedIn\_Component(o\_lg) := DomainModel\_corresp\_Component(Attribute\_definedIn\_DomainModel(AT))$

act10: $Constant\_typing\_Property(o\_atmap) := o\_lg$

end

Event rule_11_2_2 ⟨convergent⟩ ≙
correspondence of an instance of AttributeMaplet (case where the image (of type DataValue) corresponds to a set item

any atmap o_atmap AT antecedent image o_lg o_antecedent o_image
where

grd0: $AttributeMaplet \setminus dom(AttributeMaplet\_corresp\_Constant) \neq \emptyset$
grd1: $atmap \in AttributeMaplet \setminus dom(AttributeMaplet\_corresp\_Constant)$
grd2: $dom(Attribute\_corresp\_Constant) \cup dom(Attribute\_corresp\_Variable) \neq \emptyset$
grd3: $AttributeMaplet\_mapletOf\_Attribute(atmap) = AT$
grd4: $AT \in dom(Attribute\_corresp\_Constant) \cup dom(Attribute\_corresp\_Variable)$
grd5: $Attribute\_definedIn\_DomainModel(AT) \in dom(DomainModel\_corresp\_Component)$
grd6: $Constant\_Set \setminus Constant \neq \emptyset$
grd7: $o\_atmap \in Constant\_Set \setminus Constant$
grd8: $LogicFormula\_Set \setminus LogicFormula \neq \emptyset$
grd9: $o\_lg \in LogicFormula\_Set \setminus LogicFormula$
grd10: $antecedent = AttributeMaplet\_antecedent\_Individual(atmap)$
grd11: $image = AttributeMaplet\_image\_DataValue(atmap)$
grd12: $antecedent \in dom(Individual\_corresp\_Constant)$
grd13: $image \in dom(DataValue\_corresp\_SetItem)$
grd14: $o\_antecedent = Individual\_corresp\_Constant(antecedent)$
grd15: $o\_image = DataValue\_corresp\_SetItem(image)$

then

act1: $Constant := Constant \cup \{o\_atmap\}$
act2: $AttributeMaplet\_corresp\_Constant(atmap) := o\_atmap$
act3: $Constant\_definedIn\_Component(o\_atmap) := DomainModel\_corresp\_Component(Attribute\_definedIn\_DomainModel(AT))$

act4: $Property := Property \cup \{o\_lg\}$
act5: $LogicFormula := LogicFormula \cup \{o\_lg\}$
act6: $LogicFormula\_uses\_Operators(o\_lg) := \{1 \mapsto Maplet\_OP\}$
act7: $Constant\_isInvolvedIn\_LogicFormulas := Constant\_isInvolvedIn\_LogicFormulas \Leftarrow \{o\_atmap \mapsto \{1 \mapsto o\_lg\}, o\_antecedent \mapsto \{2 \mapsto o\_lg\} \cup Constant\_isInvolvedIn\_LogicFormulas(o\_antecedent)\}$
act8: $LogicFormula\_involves\_Sets(o\_lg) := \emptyset$
act9: $LogicFormula\_involves\_SetItems(o\_lg) := \{3 \mapsto o\_image\}$
act10: $LogicFormula\_definedIn\_Component(o\_lg) := DomainModel\_corresp\_Component(Attribute\_definedIn\_DomainModel(AT))$

act11: $Constant\_typing\_Property(o\_atmap) := o\_lg$

end

Event rule_12_1 ⟨ordinary⟩ ≘
closure property for constant relations
any RE o_lg o_RE maplets o_maplets
where

grd0: $dom(Relation\_corresp\_Constant) \neq \emptyset$
grd1: $RE \in dom(Relation\_corresp\_Constant)$
grd2: $o\_RE = Relation\_corresp\_Constant(RE)$
grd3: $LogicFormula\_uses\_Operators^{-1}[\{\{1 \mapsto Equal2SetOf\_OP\}\}] \cap ran(Constant\_isInvolvedIn\_LogicFormulas(o\_RE)) = \emptyset$
grd4: $RelationMaplet\_mapletOf\_Relation^{-1}[\{RE\}] = maplets$
grd5: $maplets \subseteq dom(RelationMaplet\_corresp\_Constant)$
grd6: $o\_maplets = RelationMaplet\_corresp\_Constant[maplets]$
grd7: $Relation\_definedIn\_DomainModel(RE) \in dom(DomainModel\_corresp\_Component)$
grd8: $LogicFormula\_Set \setminus LogicFormula \neq \emptyset$
grd9: $o\_lg \in LogicFormula\_Set \setminus LogicFormula$
grd10: $o\_RE \notin o\_maplets$

then

act1: $Property := Property \cup \{o\_lg\}$
act2: $LogicFormula := LogicFormula \cup \{o\_lg\}$
act3: $LogicFormula\_uses\_Operators(o\_lg) := \{1 \mapsto Equal2SetOf\_OP\}$
act4: $Constant\_isInvolvedIn\_LogicFormulas := Constant\_isInvolvedIn\_LogicFormulas \Leftarrow (\{o\_RE \mapsto \{1 \mapsto o\_lg\} \cup Constant\_isInvolvedIn\_LogicFormulas(o\_RE)\} \cup \{co \mapsto lgs | co \in o\_maplets \wedge lgs = \{2 \mapsto o\_lg\} \cup Constant\_isInvolvedIn\_LogicFormulas(co)\})$
appearence order does not matter
act5: $LogicFormula\_involves\_Sets(o\_lg) := \emptyset$
act6: $LogicFormula\_definedIn\_Component(o\_lg) := DomainModel\_corresp\_Component(Relation\_definedIn\_DomainModel(RE))$

end

Event rule_12_2 ⟨ordinary⟩ ≘
closure action for variable relations
any RE o_ia o_RE maplets o_maplets ex_o_ia bij_o_maplets
where

grd0: $dom(Relation\_corresp\_Variable) \neq \emptyset$
grd1: $RE \in dom(Relation\_corresp\_Variable)$
grd2: $o\_RE = Relation\_corresp\_Variable(RE)$
grd3: $Variable\_init\_InitialisationAction(o\_RE) \notin InitialisationAction\_uses\_Operators^{-1}[\{\{1 \mapsto BecomeEqual2SetOf\_OP\}\}]$

grd4: $RelationMaplet\_mapletOf\_Relation^{-1}[\{RE\}] = maplets$
grd5: $maplets \subseteq dom(RelationMaplet\_corresp\_Constant)$
grd6: $o\_maplets = RelationMaplet\_corresp\_Constant[maplets]$
grd7: $Relation\_definedIn\_DomainModel(RE) \in dom(DomainModel\_corresp\_Component)$
grd8: $InitialisationAction\_Set \setminus InitialisationAction \neq \emptyset$

grd9: $o\_ia \in InitialisationAction\_Set \setminus InitialisationAction$
grd10: $ex\_o\_ia = Variable\_init\_InitialisationAction(o\_RE)$
grd11: $Variable\_init\_InitialisationAction^{-1}[\{ex\_o\_ia\}] = \{o\_RE\}$
nous sommes certains que dans le cas d'espèce, l'action d'initialisation de o_RE ne fait intervenir que o_RE : en effet nous l'avons explicitement definie (rule 13)
grd12: $finite(o\_maplets)$
grd13: $bij\_o\_maplets \in 1 \,..\, card(o\_maplets) \rightarrowtail\!\!\!\!\!\twoheadrightarrow o\_maplets$

then

act1: $InitialisationAction := (InitialisationAction \setminus \{ex\_o\_ia\}) \cup \{o\_ia\}$
act2: $InitialisationAction\_uses\_Operators := (InitialisationAction\_uses\_Operators \setminus \{ex\_o\_ia \mapsto InitialisationAction\_uses\_Operators(ex\_o\_ia)\}) \lhd\!\!\!\!\!- \{o\_ia \mapsto \{1 \mapsto BecomeEqual2SetOf\_OP\}\}$
act3: $Variable\_init\_InitialisationAction(o\_RE) := o\_ia$
act4: $InitialisationAction\_involves\_Constants := (InitialisationAction\_involves\_Constants \setminus \{ex\_o\_ia \mapsto InitialisationAction\_involves\_Constants(ex\_o\_ia)\}) \lhd\!\!\!\!\!- \{o\_ia \mapsto bij\_o\_maplets\}$

end

Event rule_13_1 ⟨convergent⟩ $\widehat{=}$
correspondence of an instance of Relation having its isVariable property set to true (case where domain and range correspond to abstract sets. The others cases will not explicitely included here, since they can easily be obtained based on rules 10_2, 10_3 and 10_4)

any RE T_RE o_RE CO1 o_CO1 CO2 o_CO2 o_lg1 o_lg2 DM o_ia

where

grd0: $Relation\_isVariable^{-1}[\{TRUE\}] \setminus dom(Relation\_Type) \neq \emptyset$
grd1: $RE \in Relation\_isVariable^{-1}[\{TRUE\}] \setminus dom(Relation\_Type)$
grd2: $dom(Concept\_corresp\_AbstractSet) \neq \emptyset$
grd3: $CO1 = Relation\_domain\_Concept(RE)$
grd4: $CO2 = Relation\_range\_Concept(RE)$
grd5: $\{CO1, CO2\} \subseteq dom(Concept\_corresp\_AbstractSet)$
grd6: $Relation\_definedIn\_DomainModel(RE) \in dom(DomainModel\_corresp\_Component)$
grd7: $Constant\_Set \setminus Constant \neq \emptyset$
grd8: $T\_RE \in Constant\_Set \setminus Constant$
grd9: $Variable\_Set \setminus Variable \neq \emptyset$
grd10: $o\_RE \in Variable\_Set \setminus Variable$
grd11: $LogicFormula\_Set \setminus LogicFormula \neq \emptyset$
grd12: $\{o\_lg1, o\_lg2\} \subseteq LogicFormula\_Set \setminus LogicFormula$
grd13: $o\_CO1 = Concept\_corresp\_AbstractSet(CO1)$
grd14: $o\_CO2 = Concept\_corresp\_AbstractSet(CO2)$
grd15: $DM = Relation\_definedIn\_DomainModel(RE)$
grd16: $o\_lg1 \neq o\_lg2$
grd17: $InitialisationAction\_Set \setminus InitialisationAction \neq \emptyset$
grd18: $o\_ia \in InitialisationAction\_Set \setminus InitialisationAction$

then

act1: $Constant := Constant \cup \{T\_RE\}$
act2: $Variable := Variable \cup \{o\_RE\}$
act3: $Relation\_Type(RE) := T\_RE$
act4: $Relation\_corresp\_Variable(RE) := o\_RE$
act5: $Constant\_definedIn\_Component(T\_RE) := DomainModel\_corresp\_Component(DM)$
act6: $Variable\_definedIn\_Component(o\_RE) := DomainModel\_corresp\_Component(DM)$
act7: $Property := Property \cup \{o\_lg1\}$
act8: $Invariant := Invariant \cup \{o\_lg2\}$
act9: $LogicFormula := LogicFormula \cup \{o\_lg1, o\_lg2\}$
act10: $Constant\_typing\_Property(T\_RE) := o\_lg1$
act11: $Variable\_typing\_Invariant(o\_RE) := o\_lg2$

act12: $Constant\_isInvolvedIn\_LogicFormulas(T\_RE) := \{1 \mapsto o\_lg1, 2 \mapsto o\_lg2\}$
act13: $Invariant\_involves\_Variables(o\_lg2) := \{1 \mapsto o\_RE\}$
act14: $LogicFormula\_uses\_Operators := LogicFormula\_uses\_Operators \cup \{o\_lg1 \mapsto \{1 \mapsto RelationSet\_OP\}, o\_lg2 \mapsto \{1 \mapsto Belonging\_OP\}\}$
act15: $LogicFormula\_involves\_Sets := LogicFormula\_involves\_Sets \cup \{o\_lg1 \mapsto \{2 \mapsto o\_CO1, 3 \mapsto o\_CO2\}, o\_lg2 \mapsto \emptyset\}$
act16: $LogicFormula\_definedIn\_Component := LogicFormula\_definedIn\_Component \cup \{o\_lg1 \mapsto DomainModel\_corresp\_Component(DM), o\_lg2 \mapsto DomainModel\_corresp\_Component(DM)\}$
act17: $InitialisationAction := InitialisationAction \cup \{o\_ia\}$
act18: $InitialisationAction\_uses\_Operators(o\_ia) := \{1 \mapsto BecomeEqual2EmptySet\_OP\}$
act19: $Variable\_init\_InitialisationAction(o\_RE) := o\_ia$
act20: $InitialisationAction\_involves\_Constants(o\_ia) := \emptyset$

end

Event rule_14_1 ⟨convergent⟩ $\widehat{=}$
correspondence of an instance of Attribute having its isVariable property set to false and its isFunctional property set to false (case where the domain corresponds to an abstract set, knowing that the range always corresponds to a set)

any AT T_AT o_AT CO o_CO DS o_DS o_lg1 o_lg2 DM

where

grd0: $Attribute\_isVariable^{-1}[\{FALSE\}] \setminus dom(Attribute\_Type) \neq \emptyset$
grd1: $AT \in Attribute\_isVariable^{-1}[\{FALSE\}] \setminus dom(Attribute\_Type)$
grd2: $dom(Concept\_corresp\_AbstractSet) \neq \emptyset$
grd3: $CO = Attribute\_domain\_Concept(AT)$
grd4: $CO \in dom(Concept\_corresp\_AbstractSet)$
grd5: $dom(DataSet\_corresp\_Set) \neq \emptyset$
grd6: $DS = Attribute\_range\_DataSet(AT)$
grd7: $DS \in dom(DataSet\_corresp\_Set)$
grd8: $Attribute\_definedIn\_DomainModel(AT) \in dom(DomainModel\_corresp\_Component)$
grd9: $Constant\_Set \setminus Constant \neq \emptyset$
grd10: $\{T\_AT, o\_AT\} \subseteq Constant\_Set \setminus Constant$
grd11: $LogicFormula\_Set \setminus LogicFormula \neq \emptyset$
grd12: $\{o\_lg1, o\_lg2\} \subseteq LogicFormula\_Set \setminus LogicFormula$
grd13: $o\_CO = Concept\_corresp\_AbstractSet(CO)$
grd14: $o\_DS = DataSet\_corresp\_Set(DS)$
grd15: $DM = Attribute\_definedIn\_DomainModel(AT)$
grd16: $T\_AT \neq o\_AT$
grd17: $o\_lg1 \neq o\_lg2$
grd18: $AT \in Attribute\_isFunctional^{-1}[\{FALSE\}]$

then

act1: $Constant := Constant \cup \{T\_AT, o\_AT\}$
act2: $Attribute\_Type(AT) := T\_AT$
act3: $Attribute\_corresp\_Constant(AT) := o\_AT$
act4: $Constant\_definedIn\_Component := Constant\_definedIn\_Component \cup \{o\_AT \mapsto DomainModel\_corresp\_Component(DM), T\_AT \mapsto DomainModel\_corresp\_Component(DM)\}$
act5: $Property := Property \cup \{o\_lg1, o\_lg2\}$
act6: $LogicFormula := LogicFormula \cup \{o\_lg1, o\_lg2\}$
act7: $Constant\_typing\_Property := Constant\_typing\_Property \cup \{T\_AT \mapsto o\_lg1, o\_AT \mapsto o\_lg2\}$
act8: $Constant\_isInvolvedIn\_LogicFormulas := Constant\_isInvolvedIn\_LogicFormulas \cup \{T\_AT \mapsto \{1 \mapsto o\_lg1, 2 \mapsto o\_lg2\}, o\_AT \mapsto \{1 \mapsto o\_lg2\}\}$
act9: $LogicFormula\_uses\_Operators := LogicFormula\_uses\_Operators \cup \{o\_lg1 \mapsto \{1 \mapsto RelationSet\_OP\}, o\_lg2 \mapsto \{1 \mapsto Belonging\_OP\}\}$
act10: $LogicFormula\_involves\_Sets := LogicFormula\_involves\_Sets \cup \{o\_lg1 \mapsto \{2 \mapsto o\_CO, 3 \mapsto o\_DS\}, o\_lg2 \mapsto \emptyset\}$

act11: $LogicFormula\_definedIn\_Component := LogicFormula\_definedIn\_Component \cup \{o\_lg1 \mapsto DomainModel\_corresp\_Component(DM), o\_lg2 \mapsto DomainModel\_corresp\_Component(DM)\}$

end

Event rule_14_2 ⟨convergent⟩ ≙

correspondence of an instance of Attribute having its isVariable property set to false and its isFunctional property set to false (case where the domain corresponds to a constant, knowing that the range always corresponds to a set)

any AT T_AT o_AT CO o_CO DS o_DS o_lg1 o_lg2 DM

where

grd0: $Attribute\_isVariable^{-1}[\{FALSE\}] \setminus dom(Attribute\_Type) \neq \emptyset$
grd1: $AT \in Attribute\_isVariable^{-1}[\{FALSE\}] \setminus dom(Attribute\_Type)$
grd2: $dom(Concept\_corresp\_Constant) \neq \emptyset$
grd3: $CO = Attribute\_domain\_Concept(AT)$
grd4: $CO \in dom(Concept\_corresp\_Constant)$
grd5: $dom(DataSet\_corresp\_Set) \neq \emptyset$
grd6: $DS = Attribute\_range\_DataSet(AT)$
grd7: $DS \in dom(DataSet\_corresp\_Set)$
grd8: $Attribute\_definedIn\_DomainModel(AT) \in dom(DomainModel\_corresp\_Component)$
grd9: $Constant\_Set \setminus Constant \neq \emptyset$
grd10: $\{T\_AT, o\_AT\} \subseteq Constant\_Set \setminus Constant$
grd11: $LogicFormula\_Set \setminus LogicFormula \neq \emptyset$
grd12: $\{o\_lg1, o\_lg2\} \subseteq LogicFormula\_Set \setminus LogicFormula$
grd13: $o\_CO = Concept\_corresp\_Constant(CO)$
grd14: $o\_DS = DataSet\_corresp\_Set(DS)$
grd15: $DM = Attribute\_definedIn\_DomainModel(AT)$
grd16: $T\_AT \neq o\_AT$
grd17: $o\_lg1 \neq o\_lg2$
grd18: $AT \in Attribute\_isFunctional^{-1}[\{FALSE\}]$

then

act1: $Constant := Constant \cup \{T\_AT, o\_AT\}$
act2: $Attribute\_Type(AT) := T\_AT$
act3: $Attribute\_corresp\_Constant(AT) := o\_AT$
act4: $Constant\_definedIn\_Component := Constant\_definedIn\_Component \cup \{o\_AT \mapsto DomainModel\_corresp\_Component(DM), T\_AT \mapsto DomainModel\_corresp\_Component(DM)\}$
act5: $Property := Property \cup \{o\_lg1, o\_lg2\}$
act6: $LogicFormula := LogicFormula \cup \{o\_lg1, o\_lg2\}$
act7: $Constant\_typing\_Property := Constant\_typing\_Property \cup \{T\_AT \mapsto o\_lg1, o\_AT \mapsto o\_lg2\}$
act8: $Constant\_isInvolvedIn\_LogicFormulas := Constant\_isInvolvedIn\_LogicFormulas \Leftarrow \{T\_AT \mapsto \{1 \mapsto o\_lg1, 2 \mapsto o\_lg2\}, o\_AT \mapsto \{1 \mapsto o\_lg2\}, o\_CO \mapsto \{2 \mapsto o\_lg1\} \cup Constant\_isInvolvedIn\_LogicFormulas(o\_CO)\}$
act9: $LogicFormula\_uses\_Operators := LogicFormula\_uses\_Operators \cup \{o\_lg1 \mapsto \{1 \mapsto RelationSet\_OP\}, o\_lg2 \mapsto \{1 \mapsto Belonging\_OP\}\}$
act10: $LogicFormula\_involves\_Sets := LogicFormula\_involves\_Sets \cup \{o\_lg1 \mapsto \{3 \mapsto o\_DS\}, o\_lg2 \mapsto \emptyset\}$
act11: $LogicFormula\_definedIn\_Component := LogicFormula\_definedIn\_Component \cup \{o\_lg1 \mapsto DomainModel\_corresp\_Component(DM), o\_lg2 \mapsto DomainModel\_corresp\_Component(DM)\}$

end

Event rule_14_3 ⟨convergent⟩ ≙

correspondence of an instance of Attribute having its isVariable property set to false and its isFunctional property set to true (case where the domain corresponds to an abstract set, knowing that the range always corresponds to a set)

any AT T_AT o_AT CO o_CO DS o_DS o_lg1 o_lg2 DM

where

grd0: $Attribute\_isVariable^{-1}[\{FALSE\}] \setminus dom(Attribute\_Type) \neq \emptyset$
grd1: $AT \in Attribute\_isVariable^{-1}[\{FALSE\}] \setminus dom(Attribute\_Type)$
grd2: $dom(Concept\_corresp\_AbstractSet) \neq \emptyset$

grd3: $CO = Attribute\_domain\_Concept(AT)$
grd4: $CO \in dom(Concept\_corresp\_AbstractSet)$
grd5: $dom(DataSet\_corresp\_Set) \neq \emptyset$
grd6: $DS = Attribute\_range\_DataSet(AT)$
grd7: $DS \in dom(DataSet\_corresp\_Set)$
grd8: $Attribute\_definedIn\_DomainModel(AT) \in dom(DomainModel\_corresp\_Component)$
grd9: $Constant\_Set \setminus Constant \neq \emptyset$
grd10: $\{T\_AT, o\_AT\} \subseteq Constant\_Set \setminus Constant$
grd11: $LogicFormula\_Set \setminus LogicFormula \neq \emptyset$
grd12: $\{o\_lg1, o\_lg2\} \subseteq LogicFormula\_Set \setminus LogicFormula$
grd13: $o\_CO = Concept\_corresp\_AbstractSet(CO)$
grd14: $o\_DS = DataSet\_corresp\_Set(DS)$
grd15: $DM = Attribute\_definedIn\_DomainModel(AT)$
grd16: $T\_AT \neq o\_AT$
grd17: $o\_lg1 \neq o\_lg2$
grd18: $AT \in Attribute\_isFunctional^{-1}[\{TRUE\}]$

then

act1: $Constant := Constant \cup \{T\_AT, o\_AT\}$
act2: $Attribute\_Type(AT) := T\_AT$
act3: $Attribute\_corresp\_Constant(AT) := o\_AT$
act4: $Constant\_definedIn\_Component := Constant\_definedIn\_Component \cup \{o\_AT \mapsto DomainModel\_corresp\_Component(DM), T\_AT \mapsto DomainModel\_corresp\_Component(DM)\}$
act5: $Property := Property \cup \{o\_lg1, o\_lg2\}$
act6: $LogicFormula := LogicFormula \cup \{o\_lg1, o\_lg2\}$
act7: $Constant\_typing\_Property := Constant\_typing\_Property \cup \{T\_AT \mapsto o\_lg1, o\_AT \mapsto o\_lg2\}$
act8: $Constant\_isInvolvedIn\_LogicFormulas := Constant\_isInvolvedIn\_LogicFormulas \cup \{T\_AT \mapsto \{1 \mapsto o\_lg1, 2 \mapsto o\_lg2\}, o\_AT \mapsto \{1 \mapsto o\_lg2\}\}$
act9: $LogicFormula\_uses\_Operators := LogicFormula\_uses\_Operators \cup \{o\_lg1 \mapsto \{1 \mapsto FunctionSet\_OP\}, o\_lg2 \mapsto \{1 \mapsto Belonging\_OP\}\}$
act10: $LogicFormula\_involves\_Sets := LogicFormula\_involves\_Sets \cup \{o\_lg1 \mapsto \{2 \mapsto o\_CO, 3 \mapsto o\_DS\}, o\_lg2 \mapsto \emptyset\}$
act11: $LogicFormula\_definedIn\_Component := LogicFormula\_definedIn\_Component \cup \{o\_lg1 \mapsto DomainModel\_corresp\_Component(DM), o\_lg2 \mapsto DomainModel\_corresp\_Component(DM)\}$

end

Event rule_14_4 ⟨convergent⟩ ≙
correspondence of an instance of Attribute having its isVariable property set to false and its isFunctional property set to true (case where the domain corresponds to a constant, knowing that the range always corresponds to a set)

any AT T_AT o_AT CO o_CO DS o_DS o_lg1 o_lg2 DM

where

grd0: $Attribute\_isVariable^{-1}[\{FALSE\}] \setminus dom(Attribute\_Type) \neq \emptyset$
grd1: $AT \in Attribute\_isVariable^{-1}[\{FALSE\}] \setminus dom(Attribute\_Type)$
grd2: $dom(Concept\_corresp\_Constant) \neq \emptyset$
grd3: $CO = Attribute\_domain\_Concept(AT)$
grd4: $CO \in dom(Concept\_corresp\_Constant)$
grd5: $dom(DataSet\_corresp\_Set) \neq \emptyset$
grd6: $DS = Attribute\_range\_DataSet(AT)$
grd7: $DS \in dom(DataSet\_corresp\_Set)$
grd8: $Attribute\_definedIn\_DomainModel(AT) \in dom(DomainModel\_corresp\_Component)$
grd9: $Constant\_Set \setminus Constant \neq \emptyset$
grd10: $\{T\_AT, o\_AT\} \subseteq Constant\_Set \setminus Constant$
grd11: $LogicFormula\_Set \setminus LogicFormula \neq \emptyset$
grd12: $\{o\_lg1, o\_lg2\} \subseteq LogicFormula\_Set \setminus LogicFormula$
grd13: $o\_CO = Concept\_corresp\_Constant(CO)$

grd14: $o\_DS = DataSet\_corresp\_Set(DS)$
grd15: $DM = Attribute\_definedIn\_DomainModel(AT)$
grd16: $T\_AT \neq o\_AT$
grd17: $o\_lg1 \neq o\_lg2$
grd18: $AT \in Attribute\_isFunctional^{-1}[\{TRUE\}]$

then

act1: $Constant := Constant \cup \{T\_AT, o\_AT\}$
act2: $Attribute\_Type(AT) := T\_AT$
act3: $Attribute\_corresp\_Constant(AT) := o\_AT$
act4: $Constant\_definedIn\_Component := Constant\_definedIn\_Component \cup \{o\_AT \mapsto DomainModel\_corresp\_Component(DM), T\_AT \mapsto DomainModel\_corresp\_Component(DM)\}$
act5: $Property := Property \cup \{o\_lg1, o\_lg2\}$
act6: $LogicFormula := LogicFormula \cup \{o\_lg1, o\_lg2\}$
act7: $Constant\_typing\_Property := Constant\_typing\_Property \cup \{T\_AT \mapsto o\_lg1, o\_AT \mapsto o\_lg2\}$
act8: $Constant\_isInvolvedIn\_LogicFormulas := Constant\_isInvolvedIn\_LogicFormulas \Leftarrow \{T\_AT \mapsto \{1 \mapsto o\_lg1, 2 \mapsto o\_lg2\}, o\_AT \mapsto \{1 \mapsto o\_lg2\}, o\_CO \mapsto \{2 \mapsto o\_lg1\} \cup Constant\_isInvolvedIn\_LogicFormulas(o\_CO)\}$
act9: $LogicFormula\_uses\_Operators := LogicFormula\_uses\_Operators \cup \{o\_lg1 \mapsto \{1 \mapsto FunctionSet\_OP\}, o\_lg2 \mapsto \{1 \mapsto Belonging\_OP\}\}$
act10: $LogicFormula\_involves\_Sets := LogicFormula\_involves\_Sets \cup \{o\_lg1 \mapsto \{3 \mapsto o\_DS\}, o\_lg2 \mapsto \emptyset\}$
act11: $LogicFormula\_definedIn\_Component := LogicFormula\_definedIn\_Component \cup \{o\_lg1 \mapsto DomainModel\_corresp\_Component(DM), o\_lg2 \mapsto DomainModel\_corresp\_Component(DM)\}$

end

Event rule_15_1 ⟨ordinary⟩ ≘
closure property for constant attribute

any AT o_lg o_AT maplets o_maplets
where

grd0: $dom(Attribute\_corresp\_Constant) \neq \emptyset$
grd1: $AT \in dom(Attribute\_corresp\_Constant)$
grd2: $o\_AT = Attribute\_corresp\_Constant(AT)$
grd3: $LogicFormula\_uses\_Operators^{-1}[\{\{1 \mapsto Equal2SetOf\_OP\}\}] \cap ran(Constant\_isInvolvedIn\_LogicFormulas(o\_AT)) = \emptyset$
grd4: $AttributeMaplet\_mapletOf\_Attribute^{-1}[\{AT\}] = maplets$
grd5: $maplets \subseteq dom(AttributeMaplet\_corresp\_Constant)$
grd6: $o\_maplets = AttributeMaplet\_corresp\_Constant[maplets]$
grd7: $Attribute\_definedIn\_DomainModel(AT) \in dom(DomainModel\_corresp\_Component)$
grd8: $LogicFormula\_Set \setminus LogicFormula \neq \emptyset$
grd9: $o\_lg \in LogicFormula\_Set \setminus LogicFormula$
grd10: $o\_AT \notin o\_maplets$

then

act1: $Property := Property \cup \{o\_lg\}$
act2: $LogicFormula := LogicFormula \cup \{o\_lg\}$
act3: $LogicFormula\_uses\_Operators(o\_lg) := \{1 \mapsto Equal2SetOf\_OP\}$
act4: $Constant\_isInvolvedIn\_LogicFormulas := Constant\_isInvolvedIn\_LogicFormulas \Leftarrow (\{o\_AT \mapsto (\{1 \mapsto o\_lg\} \cup Constant\_isInvolvedIn\_LogicFormulas(o\_AT))\} \cup \{co \mapsto lgs | co \in o\_maplets \wedge lgs = \{2 \mapsto o\_lg\} \cup Constant\_isInvolvedIn\_LogicFormulas(co)\})$
appearence order does not matter
act5: $LogicFormula\_involves\_Sets(o\_lg) := \emptyset$
act6: $LogicFormula\_definedIn\_Component(o\_lg) := DomainModel\_corresp\_Component(Attribute\_definedIn\_DomainModel(AT))$

end

Event rule_16_1 ⟨ordinary⟩ ≘
handling the transitivity of a constant relation

any RE o_lg1 o_lg2 o_RE composition

where

grd0: $(dom(Relation\_corresp\_Constant) \cap Relation\_isTransitive^{-1}[\{TRUE\}]) \neq \emptyset$
grd1: $RE \in (dom(Relation\_corresp\_Constant) \cap Relation\_isTransitive^{-1}[\{TRUE\}])$
grd2: $(\{RE \mapsto isTransitive\}) \notin dom(RelationCharacteristic\_corresp\_LogicFormula)$
grd3: $o\_RE = Relation\_corresp\_Constant(RE)$
grd4: $Relation\_definedIn\_DomainModel(RE) \in dom(DomainModel\_corresp\_Component)$
grd5: $LogicFormula\_Set \setminus LogicFormula \neq \emptyset$
grd6: $\{o\_lg1, o\_lg2\} \subseteq LogicFormula\_Set \setminus LogicFormula$
grd7: $partition(\{o\_lg1, o\_lg2\}, \{o\_lg1\}, \{o\_lg2\})$
grd8: $Constant\_Set \setminus Constant \neq \emptyset$
grd9: $composition \in Constant\_Set \setminus Constant$

then

act0: $Constant := Constant \cup \{composition\}$
act1: $Property := Property \cup \{o\_lg1, o\_lg2\}$
act2: $LogicFormula := LogicFormula \cup \{o\_lg1, o\_lg2\}$
act3: $Constant\_typing\_Property(composition) := o\_lg1$
act4: $RelationCharacteristic\_corresp\_LogicFormula(\{RE \mapsto isTransitive\}) := o\_lg2$
act5: $Constant\_isInvolvedIn\_LogicFormulas := Constant\_isInvolvedIn\_LogicFormulas \Leftarrow \{composition \mapsto \{1 \mapsto o\_lg1, 1 \mapsto o\_lg2\}, o\_RE \mapsto \{2 \mapsto o\_lg1, 3 \mapsto o\_lg1, 2 \mapsto o\_lg2\} \cup Constant\_isInvolvedIn\_LogicFormulas(o\_RE)\}$

act6: $LogicFormula\_uses\_Operators := LogicFormula\_uses\_Operators \cup \{o\_lg1 \mapsto \{1 \mapsto RelationComposition\_OP\}, o\_lg2 \mapsto \{1 \mapsto Inclusion\_OP\}\}$
act7: $LogicFormula\_involves\_Sets := LogicFormula\_involves\_Sets \cup \{o\_lg1 \mapsto \emptyset, o\_lg2 \mapsto \emptyset\}$
act8: $LogicFormula\_definedIn\_Component := LogicFormula\_definedIn\_Component \cup \{o\_lg1 \mapsto DomainModel\_corresp\_Component(Relation\_definedIn\_DomainModel(RE)), o\_lg2 \mapsto DomainModel\_corresp\_Component(Relation\_definedIn\_DomainModel(RE))\}$
act9: $Constant\_definedIn\_Component(composition) := DomainModel\_corresp\_Component(Relation\_definedIn\_DomainModel(RE))$

end

Event rule_16_2 ⟨ordinary⟩ ≙
handling the symmetrie of a constant relation
any RE o_lg1 o_lg2 o_RE inverse
where

grd0: $(dom(Relation\_corresp\_Constant) \cap Relation\_isSymmetric^{-1}[\{TRUE\}]) \neq \emptyset$
grd1: $RE \in (dom(Relation\_corresp\_Constant) \cap Relation\_isSymmetric^{-1}[\{TRUE\}])$
grd2: $(\{RE \mapsto isSymmetric\}) \notin dom(RelationCharacteristic\_corresp\_LogicFormula)$
grd3: $o\_RE = Relation\_corresp\_Constant(RE)$
grd4: $Relation\_definedIn\_DomainModel(RE) \in dom(DomainModel\_corresp\_Component)$
grd5: $LogicFormula\_Set \setminus LogicFormula \neq \emptyset$
grd6: $\{o\_lg1, o\_lg2\} \subseteq LogicFormula\_Set \setminus LogicFormula$
grd7: $partition(\{o\_lg1, o\_lg2\}, \{o\_lg1\}, \{o\_lg2\})$
grd8: $Constant\_Set \setminus Constant \neq \emptyset$
grd9: $inverse \in Constant\_Set \setminus Constant$

then

act0: $Constant := Constant \cup \{inverse\}$
act1: $Property := Property \cup \{o\_lg1, o\_lg2\}$
act2: $LogicFormula := LogicFormula \cup \{o\_lg1, o\_lg2\}$
act3: $Constant\_typing\_Property(inverse) := o\_lg1$
act4: $RelationCharacteristic\_corresp\_LogicFormula(\{RE \mapsto isSymmetric\}) := o\_lg2$
act5: $Constant\_isInvolvedIn\_LogicFormulas := Constant\_isInvolvedIn\_LogicFormulas \Leftarrow \{inverse \mapsto \{1 \mapsto o\_lg1, 1 \mapsto o\_lg2\}, o\_RE \mapsto \{2 \mapsto o\_lg1, 2 \mapsto o\_lg2\} \cup Constant\_isInvolvedIn\_LogicFormulas(o\_RE)\}$
act6: $LogicFormula\_uses\_Operators := LogicFormula\_uses\_Operators \cup \{o\_lg1 \mapsto \{1 \mapsto Inversion\_OP\}, o\_lg2 \mapsto \{1 \mapsto Equality\_OP\}\}$

act7: $LogicFormula\_involves\_Sets := LogicFormula\_involves\_Sets \cup \{o\_lg1 \mapsto \emptyset, o\_lg2 \mapsto \emptyset\}$
act8: $LogicFormula\_definedIn\_Component := LogicFormula\_definedIn\_Component$
$\cup \{o\_lg1 \mapsto DomainModel\_corresp\_Component(Relation\_definedIn\_DomainModel(RE)),$
$o\_lg2 \mapsto DomainModel\_corresp\_Component(Relation\_definedIn\_DomainModel(RE))\}$
act9: $Constant\_definedIn\_Component(inverse) := DomainModel\_corresp\_Component(Relation\_definedIn\_DomainModel(RE))$

end
END

A.3 Definition of the Back Propagation Rules

A.3.1 Informal Definition

The work done on case studies [58, 133] reveals that, very often, new elements need to be added to the structural part of the formal specification. These additions may be required during the specification of the body of events or during the verification and validation of the formal model (e.g. to define an invariant or a theorem required to discharge a proof obligation). These lead us to the definition of a set of rules allowing the back propagation, within the domain model, of the new elements introduced in the structural part of the *B System* specification.

Table A.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, *x* designates the domain model element corresponding to *b_x*. In addition, when used, qualifier *abstract* denotes "without parent".

Table A.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	CO	$CO \in$ Concept $Concept\_isVariable(CO) = FALSE$ Knowing that an abstract set introduced can correspond to a concept or to a custom data set, to avoid non-determinism, we choose to define *CO* as an instance of Concept. The user may subsequently change his type.
2	**Variable typed as subset of the correspondent of a concept**	x_CO b_CO	$x\_CO \in$ Variable $b\_CO \in ran(Concept\_corresp\_AbstractSet) \vee$ $b\_CO \in ran(Concept\_corresp\_Constant)$ $x\_CO \subseteq b\_CO$		$Concept\_isVariable(CO) = TRUE$

3	**Constant (resp. Variable) typed as a relation with the range corresponding to a data set**	b_AT b_CO b_DS	*b_AT* ∈ Constant (*resp.* Variable) *b_CO* ∈ *ran*(*Concept_corresp_AbstractSet*)∪ *ran*(*Concept_corresp_Constant*) *b_DS* ∈ *ran*(*DataSet_corresp_Set*) *b_AT* ∈ *b_CO* ↔ *b_DS*	AT	*AT* ∈ Attribute *Attribute_domain_Concept*(*AT*) = *CO* *Attribute_range_DataSet*(*AT*) = *DS* *Attribute_isVariable*(*AT*) = *FALSE* (The *isVariable* property is set to *TRUE* if *b_AT* ∈ Variable) The properties of *AT* such as *isFunctional* are set according to the type of *b_AT* (partial/total function, ...).
4	**Constant (resp. Variable) typed as a relation with the range corresponding to a concept**	b_RE b_CO1 b_CO2	*b_RE* ∈ Constant (*resp.* Variable) {*b_CO*1, *b_CO*2} ⊂ *ran*(*Concept_corresp_AbstractSet*) ∪ *ran*(*Concept_corresp_Constant*) *b_RE* ∈ *b_CO*1 ↔ *b_CO*2	RE	*RE* ∈ Relation *Relation_domain_Concept*(*RE*) = *CO*1 *Relation_range_Concept*(*RE*) = *CO*2 *Relation_isVariable*(*RE*) = *FALSE* (The *isVariable* property is set to *TRUE* if *b_RE* ∈ Variable) As usual, the cardinalities of *RE* are set according to the type of *b_RE* (*function*, *injection*, ...).
5	**Constant typed as subset of the correspondent of a concept**	b_CO b_PCO	*b_CO* ∈ Constant *b_PCO* ∈ *ran*(*Concept_corresp_AbstractSet*) ∨ *b_PCO* ∈ *ran*(*Concept_corresp_Constant*) *b_CO* ⊆ *b_PCO*	CO	*CO* ∈ Concept *Concept_parentConcept_Concept*(*CO*) = *PCO* *Concept_isVariable*(*CO*) = *FALSE*
6	**Set item**	b_elt b_ES	*b_elt* ∈ *SetItem* *b_ES* = *SetItem_itemOf_EnumeratedSet*(*b_elt*) *b_ES* has a domain model correspondent	elt	*elt* ∈ DataValue *DataValue_elements_-EnumeratedDataSet*(*elt*) = *ES*
7	**Constant typed as element of the correspondent of a concept**	b_ind b_CO	*b_ind* ∈ Constant *b_CO* ∈ *ran*(*Concept_corresp_AbstractSet*)∨ *b_CO* ∈ *ran*(*Concept_corresp_Constant*) *b_ind* ∈ *b_CO*	ind	*ind* ∈ Individual *Individual_individualOf_Concept*(*ind*) = *CO*
8	**Constant typed as element of the correspondent of a data set**	b_dva b_DS	*b_dva* ∈ Constant *b_DS* ∈ *ran*(*DataSet_corresp_Set*) *b_dva* ∈ *b_DS*	dva	*dva* ∈ DataValue *DataValue_valueOf_DataSet*(*dva*) = *DS*

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.

In what follows, we provide a description of some relevant rules. These rules have been chosen to make explicit the formalism used in Table A.2.

Addition of Abstract Sets

An abstract set *b_CO* (instance of class AbstractSet of the metamodel of Fig. 4.1) introduced in the *B System* specification gives a concept *CO* (instance of class Concept of the metamodel of Fig. 4.3) having its property *isVariable* set to *FALSE* (line 1 of Table A.2). If *b_CO* is set as the superset of a variable *x_CO*, then it is possible to dynamically add/remove individuals from concept *CO*: thus, property *isVariable* of *CO* must be set to *TRUE* (line 2 of Table A.2).

Addition of Constants or Variables typed as relations

The introduction in the *B System* specification of a constant typed as a relation can be back propagated, within the domain model, with the definition of a constant attribute (instance of class Attribute) or relation (instance of class Relation): (1) if the range of the constant is the correspondence of a data set (instance of class DataSet), then the element added within the domain model must be an attribute (line 3 of Table A.2); (2) however, if the range is the correspondence of a concept (instance of class Concept), then the element added within the domain model must be a relation (line 4 of Table A.2). When the *B System* relation is a variable, then property *isVariable* of the relation or attribute introduced in the domain model is set to *true*.

Addition of Subsets of Correspondences of concepts

A constant *b_CO* introduced in the *B System* specification and defined as a subset of *b_PCO*, the correspondent of a concept *PCO*, gives a concept *CO* having *PCO* as its parent concept (association parentConcept of the metamodel of Fig. 4.3) (line 5 of Table A.2). If *b_CO* is set as the superset of a variable *x_CO*, then it is possible to dynamically add/remove individuals from concept *CO*: thus, property *isVariable* of *CO* must be set to *TRUE* (line 2 of Table A.2).

Addition of Set Items

An item *b_elt* (instance of class SetItem of the metamodel of Fig. 4.1) added to a set *b_ES* gives a data value *elt* (instance of class DataValue of the metamodel of Fig. 4.3) linked to the enumerated dataset corresponding to *b_ES* with the association element (line 6 of Table A.2).

A.3.2 Event-B Specification

MACHINE event_b_specs_from_ontologies_ref_1
REFINES event_b_specs_from_ontologies
SEES EventB_Metamodel_Context,Domain_Metamodel_Context

A.3. Definition of the Back Propagation Rules

EVENTS

Event rule_b1 ⟨convergent⟩ ≘

handling the addition of a new abstract set (correspondence to a concept)

any CO o_CO

where

grd0: $AbstractSet \setminus (ran(Concept\_corresp\_AbstractSet) \cup ran(DataSet\_corresp\_Set)) \neq \emptyset$

grd1: $o\_CO \in AbstractSet \setminus (ran(Concept\_corresp\_AbstractSet) \cup ran(DataSet\_corresp\_Set))$

grd2: $Set\_definedIn\_Component(o\_CO) \in ran(DomainModel\_corresp\_Component)$

grd3: $Concept\_Set \setminus Concept \neq \emptyset$

grd4: $CO \in Concept\_Set \setminus Concept$

then

act1: $Concept := Concept \cup \{CO\}$

act2: $Concept\_corresp\_AbstractSet(CO) := o\_CO$

act3: $Concept\_definedIn\_DomainModel(CO) := DomainModel\_corresp\_Component^{-1}(Set\_definedIn\_Component(o\_CO))$

act4: $Concept\_isVariable(CO) := FALSE$

end

Event rule_b2 ⟨convergent⟩ ≘

handling the addition of a new abstract set (correspondence to a custom data set)

any DS o_DS

where

grd0: $AbstractSet \setminus (ran(Concept\_corresp\_AbstractSet) \cup ran(DataSet\_corresp\_Set)) \neq \emptyset$

grd1: $o\_DS \in AbstractSet \setminus (ran(Concept\_corresp\_AbstractSet) \cup ran(DataSet\_corresp\_Set))$

grd2: $Set\_definedIn\_Component(o\_DS) \in ran(DomainModel\_corresp\_Component)$

grd3: $DataSet\_Set \setminus DataSet \neq \emptyset$

grd4: $DS \in DataSet\_Set \setminus DataSet$

grd5: $DS \notin \{\_NATURAL, \_INTEGER, \_FLOAT, \_BOOL, \_STRING\}$

then

act1: $CustomDataSet := CustomDataSet \cup \{DS\}$

act2: $DataSet := DataSet \cup \{DS\}$

act3: $CustomDataSet\_corresp\_AbstractSet(DS) := o\_DS$

act4: $DataSet\_definedIn\_DomainModel(DS) := DomainModel\_corresp\_Component^{-1}(Set\_definedIn\_Component(o\_DS))$

act5: $DataSet\_corresp\_Set(DS) := o\_DS$

end

Event rule_b3 ⟨convergent⟩ ≘

handling the addition of an enumerated set

any EDS o_EDS elements o_elements mapping_elements_o_elements

where

grd0: $EnumeratedSet \setminus ran(DataSet\_corresp\_Set) \neq \emptyset$

grd1: $o\_EDS \in EnumeratedSet \setminus ran(DataSet\_corresp\_Set)$

grd2: $Set\_definedIn\_Component(o\_EDS) \in ran(DomainModel\_corresp\_Component)$

grd3: $DataSet\_Set \setminus DataSet \neq \emptyset$

grd4: $EDS \in DataSet\_Set \setminus DataSet$

grd5: $DataValue\_Set \setminus DataValue \neq \emptyset$

grd6: $elements \subseteq DataValue\_Set \setminus DataValue$

grd7: $o\_elements = SetItem\_itemOf\_EnumeratedSet^{-1}[\{o\_EDS\}]$

grd8: $card(o\_elements) = card(elements)$

grd9: $mapping\_elements\_o\_elements \in elements \rightarrowtail\!\!\!\!\!\rightarrow o\_elements$

grd10: $ran(DataValue\_corresp\_SetItem) \cap o\_elements = \emptyset$

grd11: $EDS \notin \{\_NATURAL, \_INTEGER, \_FLOAT, \_BOOL, \_STRING\}$

then

act1: $EnumeratedDataSet := EnumeratedDataSet \cup \{EDS\}$
act2: $DataSet := DataSet \cup \{EDS\}$
act3: $EnumeratedDataSet\_corresp\_EnumeratedSet(EDS) := o\_EDS$
act4: $DataSet\_definedIn\_DomainModel(EDS) := DomainModel\_corresp\_Component^{-1}(Set\_definedIn\_Component(o\_EDS))$

act5: $DataValue := DataValue \cup elements$
act6: $DataValue\_elements\_EnumeratedDataSet := DataValue\_elements\_EnumeratedDataSet \cup \{(xx \mapsto yy) | xx \in elements \wedge yy = EDS\}$
act7: $DataValue\_corresp\_SetItem := DataValue\_corresp\_SetItem \cup mapping\_elements\_o\_elements$
act8: $DataSet\_corresp\_Set := DataSet\_corresp\_Set \Leftarrow \{EDS \mapsto o\_EDS\}$
act9: $DataValue\_valueOf\_DataSet := DataValue\_valueOf\_DataSet \cup \{(xx \mapsto yy) | xx \in elements \wedge yy = EDS\}$
act10: $CustomDataSet := CustomDataSet \cup \{EDS\}$

end

Event rule_b4 ⟨convergent⟩ ≙
handling the addition of a new element in an existing enumerated set
any EDS o_EDS element o_element
where

grd0: $dom(SetItem\_itemOf\_EnumeratedSet) \setminus ran(DataValue\_corresp\_SetItem) \neq \emptyset$
grd1: $o\_element \in dom(SetItem\_itemOf\_EnumeratedSet) \setminus ran(DataValue\_corresp\_SetItem)$
grd2: $o\_EDS = SetItem\_itemOf\_EnumeratedSet(o\_element)$
grd3: $o\_EDS \in ran(EnumeratedDataSet\_corresp\_EnumeratedSet)$
grd4: $EDS = EnumeratedDataSet\_corresp\_EnumeratedSet^{-1}(o\_EDS)$
grd5: $DataValue\_Set \setminus DataValue \neq \emptyset$
grd6: $element \in DataValue\_Set \setminus DataValue$

then

act1: $DataValue := DataValue \cup \{element\}$
act2: $DataValue\_elements\_EnumeratedDataSet(element) := EDS$
act3: $DataValue\_corresp\_SetItem(element) := o\_element$
act4: $DataValue\_valueOf\_DataSet(element) := EDS$

end

Event rule_b5_1 ⟨convergent⟩ ≙
handling the addition of a constant, sub set of an instance of Concept (case where the concept corresponds to an abstract set)
any CO o_CO PCO o_lg o_PCO
where

grd0:
$dom(Constant\_typing\_Property) \setminus (ran(Concept\_corresp\_Constant)$
$\cup\ ran(Individual\_corresp\_Constant)$
$\cup\ ran(DataValue\_corresp\_Constant)$
$\cup\ ran(Relation\_corresp\_Constant)$
$\cup\ ran(Attribute\_corresp\_Constant)$
$\cup\ ran(RelationMaplet\_corresp\_Constant)$
$\cup\ ran(AttributeMaplet\_corresp\_Constant)$
$\cup\ ran(Attribute\_Type)$
$\cup\ ran(Relation\_Type)) \neq \emptyset$

grd1:
$o\_CO \in dom(Constant\_typing\_Property) \setminus (ran(Concept\_corresp\_Constant)$
$\cup\ ran(Individual\_corresp\_Constant)$
$\cup\ ran(DataValue\_corresp\_Constant)$
$\cup\ ran(Relation\_corresp\_Constant)$
$\cup\ ran(Attribute\_corresp\_Constant)$
$\cup\ ran(RelationMaplet\_corresp\_Constant)$
$\cup\ ran(AttributeMaplet\_corresp\_Constant)$
$\cup\ ran(Attribute\_Type)$
$\cup\ ran(Relation\_Type))$

grd2: $o\_lg = Constant\_typing\_Property(o\_CO)$
grd3: $LogicFormula\_uses\_Operators(o\_lg) = \{1 \mapsto Inclusion\_OP\}$
grd4: $LogicFormula\_involves\_Sets(o\_lg) \neq \emptyset$
grd5: $(2 \mapsto o\_PCO) \in LogicFormula\_involves\_Sets(o\_lg)$
grd6: $o\_PCO \in ran(Concept\_corresp\_AbstractSet)$
grd7: $PCO = Concept\_corresp\_AbstractSet^{-1}(o\_PCO)$
grd8: $Concept\_Set \setminus Concept \neq \emptyset$
grd9: $CO \in Concept\_Set \setminus Concept$
grd10: $Constant\_definedIn\_Component(o\_CO) \in ran(DomainModel\_corresp\_Component)$

then

act1: $Concept := Concept \cup \{CO\}$
act2: $Concept\_corresp\_Constant(CO) := o\_CO$
act3: $Concept\_definedIn\_DomainModel(CO) := DomainModel\_corresp\_Component^{-1}(Constant\_definedIn\_Component(o\_CO))$

act4: $Concept\_parentConcept\_Concept(CO) := PCO$
act5: $Concept\_isVariable(CO) := FALSE$

end

Event rule_b5_2 ⟨convergent⟩ ≙
handling the addition of a constant, sub set of an instance of Concept (case where the concept corresponds to a constant)

any CO o_CO PCO o_lg o_PCO
where

grd0:
$dom(Constant\_typing\_Property) \setminus (ran(Concept\_corresp\_Constant)$
$\cup\ ran(Individual\_corresp\_Constant)$
$\cup\ ran(DataValue\_corresp\_Constant)$
$\cup\ ran(Relation\_corresp\_Constant)$
$\cup\ ran(Attribute\_corresp\_Constant)$
$\cup\ ran(RelationMaplet\_corresp\_Constant)$
$\cup\ ran(AttributeMaplet\_corresp\_Constant)$
$\cup\ ran(Attribute\_Type)$
$\cup\ ran(Relation\_Type)) \neq \emptyset$

grd1:
$o\_CO \in dom(Constant\_typing\_Property) \setminus (ran(Concept\_corresp\_Constant)$
$\cup\ ran(Individual\_corresp\_Constant)$
$\cup\ ran(DataValue\_corresp\_Constant)$
$\cup\ ran(Relation\_corresp\_Constant)$
$\cup\ ran(Attribute\_corresp\_Constant)$
$\cup\ ran(RelationMaplet\_corresp\_Constant)$
$\cup\ ran(AttributeMaplet\_corresp\_Constant)$
$\cup\ ran(Attribute\_Type)$
$\cup\ ran(Relation\_Type))$

grd2: $o\_lg = Constant\_typing\_Property(o\_CO)$
grd3: $LogicFormula\_uses\_Operators(o\_lg) = \{1 \mapsto Inclusion\_OP\}$
grd4: $LogicFormula\_involves\_Sets(o\_lg) = \emptyset$
grd5: $o\_PCO \in dom(Constant\_isInvolvedIn\_LogicFormulas)$
grd6: $(2 \mapsto o\_lg) \in Constant\_isInvolvedIn\_LogicFormulas(o\_PCO)$
grd7: $o\_PCO \in ran(Concept\_corresp\_Constant)$
grd8: $PCO = Concept\_corresp\_Constant^{-1}(o\_PCO)$
grd9: $Concept\_Set \setminus Concept \neq \emptyset$
grd10: $CO \in Concept\_Set \setminus Concept$
grd11: $Constant\_definedIn\_Component(o\_CO) \in ran(DomainModel\_corresp\_Component)$

then

act1: $Concept := Concept \cup \{CO\}$

act2: $Concept\_corresp\_Constant(CO) := o\_CO$
act3: $Concept\_definedIn\_DomainModel(CO) := DomainModel\_corresp\_Component^{-1}(Constant\_definedIn\_Component(o\_CO))$

act4: $Concept\_parentConcept\_Concept(CO) := PCO$
act5: $Concept\_isVariable(CO) := FALSE$

end

Event rule_b6_1 ⟨convergent⟩ ≙
handling the addition of an individual (case where the concept corresponds to an abstract set)
any ind o_ind CO o_lg o_CO
where

grd0:
$dom(Constant\_typing\_Property) \setminus (ran(Concept\_corresp\_Constant)$
$\cup\ ran(Individual\_corresp\_Constant)$
$\cup\ ran(DataValue\_corresp\_Constant)$
$\cup\ ran(Relation\_corresp\_Constant)$
$\cup\ ran(Attribute\_corresp\_Constant)$
$\cup\ ran(RelationMaplet\_corresp\_Constant)$
$\cup\ ran(AttributeMaplet\_corresp\_Constant)$
$\cup\ ran(Attribute\_Type)$
$\cup\ ran(Relation\_Type)) \neq \emptyset$

grd1:
$o\_ind \in dom(Constant\_typing\_Property) \setminus (ran(Concept\_corresp\_Constant)$
$\cup\ ran(Individual\_corresp\_Constant)$
$\cup\ ran(DataValue\_corresp\_Constant)$
$\cup\ ran(Relation\_corresp\_Constant)$
$\cup\ ran(Attribute\_corresp\_Constant)$
$\cup\ ran(RelationMaplet\_corresp\_Constant)$
$\cup\ ran(AttributeMaplet\_corresp\_Constant)$
$\cup\ ran(Attribute\_Type)$
$\cup\ ran(Relation\_Type))$

grd2: $o\_lg = Constant\_typing\_Property(o\_ind)$
grd3: $LogicFormula\_uses\_Operators(o\_lg) = \{1 \mapsto Belonging\_OP\}$
grd4: $LogicFormula\_involves\_Sets(o\_lg) \neq \emptyset$
grd5: $(2 \mapsto o\_CO) \in LogicFormula\_involves\_Sets(o\_lg)$
grd6: $o\_CO \in ran(Concept\_corresp\_AbstractSet)$
grd7: $CO = Concept\_corresp\_AbstractSet^{-1}(o\_CO)$
grd8: $Individual\_Set \setminus Individual \neq \emptyset$
grd9: $ind \in Individual\_Set \setminus Individual$

then

act1: $Individual := Individual \cup \{ind\}$
act2: $Individual\_individualOf\_Concept(ind) := CO$
act3: $Individual\_corresp\_Constant(ind) := o\_ind$

end

Event rule_b6_2 ⟨convergent⟩ ≙
handling the addition of an individual (case where the concept corresponds to a constant)
any ind o_ind CO o_lg o_CO
where

grd0:
$dom(Constant\_typing\_Property) \setminus (ran(Concept\_corresp\_Constant)$
$\cup\ ran(Individual\_corresp\_Constant)$
$\cup\ ran(DataValue\_corresp\_Constant)$
$\cup\ ran(Relation\_corresp\_Constant)$
$\cup\ ran(Attribute\_corresp\_Constant)$
$\cup\ ran(RelationMaplet\_corresp\_Constant)$
$\cup\ ran(AttributeMaplet\_corresp\_Constant)$
$\cup\ ran(Attribute\_Type)$
$\cup\ ran(Relation\_Type)) \neq \emptyset$

grd1:
$o\_ind \in dom(Constant\_typing\_Property) \setminus (ran(Concept\_corresp\_Constant) \cup ran(Individual\_corresp\_Constant) \cup ran(DataValue\_corresp\_Constant) \cup ran(Relation\_corresp\_Constant) \cup ran(Attribute\_corresp\_Constant) \cup ran(RelationMaplet\_corresp\_Constant) \cup ran(AttributeMaplet\_corresp\_Constant) \cup ran(Attribute\_Type) \cup ran(Relation\_Type))$
grd2: $o\_lg = Constant\_typing\_Property(o\_ind)$
grd3: $LogicFormula\_uses\_Operators(o\_lg) = \{1 \mapsto Belonging\_OP\}$
grd4: $LogicFormula\_involves\_Sets(o\_lg) = \emptyset$
grd5: $o\_CO \in dom(Constant\_isInvolvedIn\_LogicFormulas)$
grd6: $(2 \mapsto o\_lg) \in Constant\_isInvolvedIn\_LogicFormulas(o\_CO)$
grd7: $o\_CO \in ran(Concept\_corresp\_Constant)$
grd8: $CO = Concept\_corresp\_Constant^{-1}(o\_CO)$
grd9: $Individual\_Set \setminus Individual \neq \emptyset$
grd10: $ind \in Individual\_Set \setminus Individual$

then

act1: $Individual := Individual \cup \{ind\}$
act2: $Individual\_individualOf\_Concept(ind) := CO$
act3: $Individual\_corresp\_Constant(ind) := o\_ind$

end

Event rule_b7 ⟨convergent⟩ ≙
handling the addition of a data value
any dva o_dva DS o_lg o_DS
where

grd0:
$dom(Constant\_typing\_Property) \setminus (ran(Concept\_corresp\_Constant) \cup ran(Individual\_corresp\_Constant) \cup ran(DataValue\_corresp\_Constant) \cup ran(Relation\_corresp\_Constant) \cup ran(Attribute\_corresp\_Constant) \cup ran(RelationMaplet\_corresp\_Constant) \cup ran(AttributeMaplet\_corresp\_Constant) \cup ran(Attribute\_Type) \cup ran(Relation\_Type)) \neq \emptyset$
grd1:
$o\_dva \in dom(Constant\_typing\_Property) \setminus (ran(Concept\_corresp\_Constant) \cup ran(Individual\_corresp\_Constant) \cup ran(DataValue\_corresp\_Constant) \cup ran(Relation\_corresp\_Constant) \cup ran(Attribute\_corresp\_Constant) \cup ran(RelationMaplet\_corresp\_Constant) \cup ran(AttributeMaplet\_corresp\_Constant) \cup ran(Attribute\_Type) \cup ran(Relation\_Type))$
grd2: $o\_lg = Constant\_typing\_Property(o\_dva)$
grd3: $LogicFormula\_uses\_Operators(o\_lg) = \{1 \mapsto Belonging\_OP\}$
grd4: $LogicFormula\_involves\_Sets(o\_lg) \neq \emptyset$
grd5: $(2 \mapsto o\_DS) \in LogicFormula\_involves\_Sets(o\_lg)$
grd6: $o\_DS \in ran(DataSet\_corresp\_Set)$
grd7: $DS = DataSet\_corresp\_Set^{-1}(o\_DS)$
grd8: $DataValue\_Set \setminus DataValue \neq \emptyset$
grd9: $dva \in DataValue\_Set \setminus DataValue$

then

act1: $DataValue := DataValue \cup \{dva\}$
act2: $DataValue\_valueOf\_DataSet(dva) := DS$
act3: $DataValue\_corresp\_Constant(dva) := o\_dva$

end

Event rule_b8_1 ⟨convergent⟩ $\widehat{=}$
handling the addition of a variable, sub set of an instance of Concept (case where the concept corresponds to an abstract set)

any x_CO CO o_lg o_CO

where

grd0:
$dom(Variable\_typing\_Invariant) \setminus (ran(Concept\_corresp\_Variable) \cup ran(Relation\_corresp\_Variable) \cup ran(Attribute\_corresp\_Variable)) \neq \emptyset$
grd1:
$x\_CO \in dom(Variable\_typing\_Invariant) \setminus (ran(Concept\_corresp\_Variable) \cup ran(Relation\_corresp\_Variable) \cup ran(Attribute\_corresp\_Variable))$
grd2: $o\_lg = Variable\_typing\_Invariant(x\_CO)$
grd3: $LogicFormula\_uses\_Operators(o\_lg) = \{1 \mapsto Inclusion\_OP\}$
grd4: $LogicFormula\_involves\_Sets(o\_lg) \neq \emptyset$
grd5: $(2 \mapsto o\_CO) \in LogicFormula\_involves\_Sets(o\_lg)$
grd6: $o\_CO \in ran(Concept\_corresp\_AbstractSet)$
grd7: $CO = Concept\_corresp\_AbstractSet^{-1}(o\_CO)$
grd8: $CO \notin dom(Concept\_corresp\_Variable)$

then

act1: $Concept\_isVariable(CO) := TRUE$
act2: $Concept\_corresp\_Variable(CO) := x\_CO$

end

Event rule_b8_2 ⟨convergent⟩ $\widehat{=}$
handling the addition of a variable, sub set of an instance of Concept (case where the concept corresponds to a constant)

any x_CO CO o_lg o_CO

where

grd0:
$dom(Variable\_typing\_Invariant) \setminus (ran(Concept\_corresp\_Variable) \cup ran(Relation\_corresp\_Variable) \cup ran(Attribute\_corresp\_Variable)) \neq \emptyset$
grd1:
$x\_CO \in dom(Variable\_typing\_Invariant) \setminus (ran(Concept\_corresp\_Variable) \cup ran(Relation\_corresp\_Variable) \cup ran(Attribute\_corresp\_Variable))$
grd2: $o\_lg = Variable\_typing\_Invariant(x\_CO)$
grd3: $LogicFormula\_uses\_Operators(o\_lg) = \{1 \mapsto Inclusion\_OP\}$
grd4: $LogicFormula\_involves\_Sets(o\_lg) = \emptyset$
grd5: $o\_CO \in dom(Constant\_isInvolvedIn\_LogicFormulas)$
grd6: $(2 \mapsto o\_lg) \in Constant\_isInvolvedIn\_LogicFormulas(o\_CO)$
grd7: $o\_CO \in ran(Concept\_corresp\_Constant)$
grd8: $CO = Concept\_corresp\_Constant^{-1}(o\_CO)$
grd9: $CO \notin dom(Concept\_corresp\_Variable)$

then

act1: $Concept\_isVariable(CO) := TRUE$
act2: $Concept\_corresp\_Variable(CO) := x\_CO$

end

Event rule_b9_1 ⟨convergent⟩ $\widehat{=}$
handling the addition of a constant, defined as a maplet, element of a relation (case where the relation corresponds to a constant relation)

any o_maplet maplet o_RE RE o_lg1 o_lg2 antecedent image o_antecedent o_image

where

grd0:
$dom(Constant\_typing\_Property) \setminus (ran(Concept\_corresp\_Constant) \cup ran(Individual\_corresp\_Constant) \cup ran(DataValue\_corresp\_Constant) \cup ran(Relation\_corresp\_Constant) \cup ran(Attribute\_corresp\_Constant) \cup ran(RelationMaplet\_corresp\_Constant) \cup ran(AttributeMaplet\_corresp\_Constant) \cup ran(Attribute\_Type) \cup ran(Relation\_Type)) \neq \emptyset$

grd1:
$o\_maplet \in dom(Constant\_typing\_Property) \setminus (ran(Concept\_corresp\_Constant) \cup ran(Individual\_corresp\_Constant) \cup ran(DataValue\_corresp\_Constant) \cup ran(Relation\_corresp\_Constant) \cup ran(Attribute\_corresp\_Constant) \cup ran(RelationMaplet\_corresp\_Constant) \cup ran(AttributeMaplet\_corresp\_Constant) \cup ran(Attribute\_Type) \cup ran(Relation\_Type))$

grd2: $o\_lg1 = Constant\_typing\_Property(o\_maplet)$

grd3: $LogicFormula\_uses\_Operators(o\_lg1) = \{1 \mapsto Maplet\_OP\}$

grd4: $\{o\_antecedent, o\_image\} \subseteq (dom(Constant\_isInvolvedIn\_LogicFormulas) \cap ran(Individual\_corresp\_Constant))$

grd5: $(2 \mapsto o\_lg1) \in Constant\_isInvolvedIn\_LogicFormulas(o\_antecedent)$

grd6: $(3 \mapsto o\_lg1) \in Constant\_isInvolvedIn\_LogicFormulas(o\_image)$

grd7: $antecedent = Individual\_corresp\_Constant^{-1}(o\_antecedent)$

grd8: $image = Individual\_corresp\_Constant^{-1}(o\_image)$

grd9: $o\_lg2 \in LogicFormula$

grd10: $LogicFormula\_uses\_Operators(o\_lg2) = \{1 \mapsto Equal2SetOf\_OP\}$

grd11: $(2 \mapsto o\_lg2) \in Constant\_isInvolvedIn\_LogicFormulas(o\_maplet)$

grd12: $o\_RE \in ran(Relation\_corresp\_Constant)$

grd13: $(1 \mapsto o\_lg2) \in Constant\_isInvolvedIn\_LogicFormulas(o\_RE)$

grd14: $RE = Relation\_corresp\_Constant^{-1}(o\_RE)$

grd15: $Relation\_Maplet\_Set \setminus RelationMaplet \neq \emptyset$

grd16: $maplet \in Relation\_Maplet\_Set \setminus RelationMaplet$

grd17: $Individual\_individualOf\_Concept(antecedent) = Relation\_domain\_Concept(RE)$

grd18: $Individual\_individualOf\_Concept(image) = Relation\_range\_Concept(RE)$

then

act1: $RelationMaplet := RelationMaplet \cup \{maplet\}$

act2: $RelationMaplet\_corresp\_Constant(maplet) := o\_maplet$

act3: $RelationMaplet\_mapletOf\_Relation(maplet) := RE$

act4: $RelationMaplet\_antecedent\_Individual(maplet) := antecedent$

act5: $RelationMaplet\_image\_Individual(maplet) := image$

end

END

Appendix B

Comprehensive Definition of the Adjusted Domain Modeling Language and Correspondence Rules

B.1 Summary of the Event-B Specification of the Adjusted Language and Rules

Table B.1 summarises the key characteristics of the Rodin project corresponding to the *Event-B* specification of the adjusted language and rules (translation and back propagation rules). The specification includes three refinement levels.

Table B.1 – Key characteristics of the *Event-B* specification of the adjusted language and rules

Characteristics	Root level	First refinement level	Second refinement level
Events	3	24	1
Invariants	11	129	5
Proof Obligations (PO)	37	1351	28
Automatically Discharged POs	27	1094	20
Interactively Discharged POs	10	257	8

Validating the consistency of the formal specification required the discharge of *1416* proof obligations of which only *275* (19.42 %) have required manual proofs. Thus, proving the new specification required less manual effort. This is due to the simplification of rules, whose number has also been reduced, introduced by the defined adjustments, and to the definition of better auto/post proof tactics.

B.2 Event-B Specification of the Adjusted SysML/KAOS Domain Modeling Language

B.2.1 Informal Definition

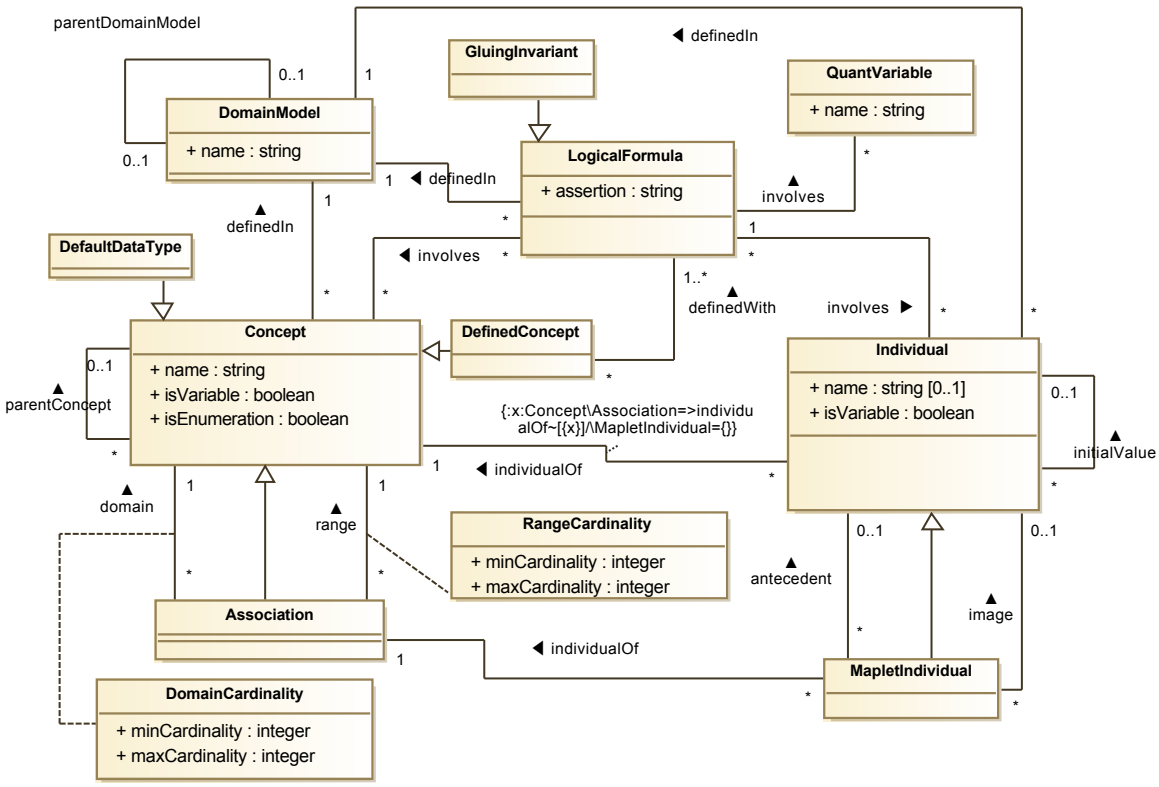


Figure B.1 – The revised SysML/KAOS domain metamodel

Figure B.1 represents the revised SysML/KAOS domain metamodel.

Description

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`.

Logical formulas (instances of LogicalFormula) are used to represent constraints between different elements of the domain model.*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. *Defined concepts* (instances of DefinedConcept) are concepts built on existing elements of the domain model using logical formulas.

Additional Constraints

This section defines the most relevant 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 [8]. For the complete list of constraints, please refer to invariants defined within the specification of Section B.2.2.

- $x \in \mathsf{Concept} \setminus \mathsf{Association}$
  $\Rightarrow \mathsf{Individual\_individualOf\_Concept}^{-1}[\{x\}] \cap \mathsf{MapletIndividual} = \emptyset$:
  if concept `x` is not an association, then no individual of `x` can be a maplet individual.
- $x \in \mathsf{MapletIndividual} \cap dom(\mathsf{MapletIndividual\_antecedent\_Individual})$
  $\Rightarrow \mathsf{MapletIndividual\_antecedent\_Individual}(x)$
  $\in \mathsf{Association\_domain\_Concept}(\mathsf{Individual\_individualOf\_Concept}(x))$:
  if maplet individual `x` has an antecedent, then the antecedent is an individual of the domain of its association.
- $x \in \mathsf{MapletIndividual} \cap dom(\mathsf{MapletIndividual\_image\_Individual})$
  $\Rightarrow \mathsf{MapletIndividual\_image\_Individual}(x)$
  $\in \mathsf{Association\_range\_Concept}(\mathsf{Individual\_individualOf\_Concept}(x))$:

if maplet individual x has an image, then the image is an individual of the range of its association.

- $ind \in \mathsf{Individual} \setminus \mathsf{MapletIndividual} \Rightarrow ind \in dom(\mathsf{Individual\_name})$:
every individual which is not a maplet individual must be named.
- $ind \in \mathsf{Individual} \setminus dom(\mathsf{Individual\_name}) \Rightarrow \mathsf{Individual\_isVariable}(ind) = FALSE$:
every unnamed individual must be constant.
- $ind \in \mathsf{MapletIndividual} \cap dom(\mathsf{MapletIndividual\_antecedent\_Individual})$
$\cap dom(\mathsf{MapletIndividual\_image\_Individual}) \Rightarrow (\mathsf{MapletIndividual\_antecedent\_Individual}(ind) \in$
$dom(\mathsf{Individual\_name})$
$\wedge\ \mathsf{MapletIndividual\_image\_Individual}(ind) \in dom(\mathsf{Individual\_name}))$:
antecedents and images of maplet individuals must be named.
- $ind \in \mathsf{MapletIndividual} \setminus dom(\mathsf{Individual\_name})$
$\Rightarrow ind \in dom(\mathsf{MapletIndividual\_antecedent\_Individual}) \cap dom(\mathsf{MapletIndividual\_image\_Individual})$:
every unnamed maplet individual must have an antecedent and an image.
- $x \in \mathsf{Concept} \setminus (\mathsf{Association} \cup \mathsf{DefinedConcept} \cup dom(\mathsf{Concept\_parent\_Concept}))$
$\Rightarrow \mathsf{Concept\_isVariable}(x) = FALSE$:
every abstract concept (that has no parent concept) that is not an association must be constant.
- $x \in \mathsf{Concept} \wedge \mathsf{Concept\_isEnumeration}(x) = TRUE \Rightarrow \mathsf{Concept\_isVariable}(x) = FALSE$:
every concept that is an enumeration must be constant.
- $(ind \in \mathsf{MapletIndividual} \cap dom(\mathsf{MapletIndividual\_antecedent\_Individual})$
$\cap\ dom(\mathsf{MapletIndividual\_image\_Individual}) \wedge \mathsf{Individual\_isVariable}(ind) = FALSE)$
$\Rightarrow (\mathsf{Individual\_isVariable}(\mathsf{MapletIndividual\_antecedent\_Individual}(ind)) = FALSE$
$\wedge\ \mathsf{Individual\_isVariable}(\mathsf{MapletIndividual\_image\_Individual}(ind)) = FALSE)$:
antecedents and images of constant maplet individuals must be constant.
- $(x \in \mathsf{Association} \wedge \mathsf{Concept\_isVariable}(x) = FALSE)$
$\Rightarrow (\mathsf{Concept\_isVariable}(\mathsf{Association\_domain\_Concept}(x)) = FALSE$
$\wedge\ \mathsf{Concept\_isVariable}(\mathsf{Association\_range\_Concept}(x)) = FALSE)$:
domains and ranges of constant associations must be constant.

B.2.2 Event-B Specification

CONTEXT Domain_Metamodel_Context
SETS DomainModel_Set Concept_Set Individual_Set RelationCharacteristics_Set
CONSTANTS DefaultDataType _NATURAL _INTEGER _FLOAT _BOOL _STRING isTransitive isSymmetric _TRUE _FALSE
AXIOMS

axiom1: $DefaultDataType \subseteq Concept\_Set$
axiom2: $partition(DefaultDataType, \{\_NATURAL\}, \{\_INTEGER\}, \{\_FLOAT\}, \{\_BOOL\}, \{\_STRING\})$
axiom3: $partition(RelationCharacteristics\_Set, \{isTransitive\}, \{isSymmetric\})$
axiom4: $finite(DomainModel\_Set) \wedge finite(Concept\_Set) \wedge finite(Individual\_Set)$
axiom5: $\{\_TRUE, \_FALSE\} \subseteq Individual\_Set \wedge \_TRUE \neq \_FALSE$

END

MACHINE event_b_specs_from_ontologies_ref_1
REFINES event_b_specs_from_ontologies
SEES EventB_Metamodel_Context,Domain_Metamodel_Context
VARIABLES DomainModel_corresp_Component Concept Individual DefinedConcept Association MapletIndividual Concept_isVariable Concept_isEnumeration Individual_isNamed Individual_isVariable Association_isTransitive Association_isSymmetric Association_isASymmetric Association_isReflexive Association_isIrreflexive
Domain Model links
Concept_definedIn_DomainModel Individual_definedIn_DomainModel Concept_parentConcept_Concept

Individual_individualOf_Concept Individual_initialValue_Individual Association_domain_Concept
Association_range_Concept Association_DomainCardinality_minCardinality Association_DomainCardinality_maxCardinality
Association_RangeCardinality_minCardinality Association_RangeCardinality_maxCardinality
MapletIndividual_antecedent_Individual MapletIndividual_image_Individual LogicFormula_defines_DefinedConcept

Correspondence links
Concept_corresp_Set Concept_corresp_Constant Concept_corresp_Variable Individual_corresp_Constant
Individual_corresp_Variable Individual_corresp_SetItem Association_Type_Constant Association_Type_Variable
AssociationCharacteristic_corresp_LogicFormula ConcreteEnumeration_corresp_IndividualSetLogicalFormula
UnnamedMapletIndividual_corresp_LogicalFormula

INVARIANTS

dm_elt_typ_inv1_1: $Concept \subseteq Concept\_Set$
dm_elt_typ_inv1_2: $Individual \subseteq Individual\_Set$
dm_elt_typ_inv1_3: $DefinedConcept \subseteq Concept$
dm_elt_typ_inv1_4: $Association \subseteq Concept$
dm_elt_typ_inv1_5: $DefaultDataType \subseteq Concept$
dm_elt_typ_inv1_6: $partition(DefaultDataType \cup DefinedConcept \cup Association, DefaultDataType, DefinedConcept, Association)$

dm_elt_typ_inv1_7: $MapletIndividual \subseteq Individual$

Domain Model properties : typing invariants

dm_prop_typ_inv1_1: $Concept\_isVariable \in Concept \rightarrow BOOL$
dm_prop_typ_inv1_2: $Concept\_isEnumeration \in Concept \rightarrow BOOL$
dm_prop_typ_inv1_3: $Individual\_isNamed \in Individual \rightarrow BOOL$
dm_prop_typ_inv1_4: $Individual\_isVariable \in Individual \rightarrow BOOL$
dm_prop_typ_inv1_5: $Association\_isTransitive \in Association \nrightarrow BOOL$
dm_prop_typ_inv1_6: $Association\_isSymmetric \in Association \nrightarrow BOOL$
dm_prop_typ_inv1_7: $Association\_isASymmetric \in Association \nrightarrow BOOL$
dm_prop_typ_inv1_8: $Association\_isReflexive \in Association \nrightarrow BOOL$
dm_prop_typ_inv1_9: $Association\_isIrreflexive \in Association \nrightarrow BOOL$

Domain Model links : typing invariants

dm_link_typ_inv1_1: $Concept\_definedIn\_DomainModel \in Concept \nrightarrow DomainModel$
dm_link_typ_inv1_2: $Individual\_definedIn\_DomainModel \in Individual \nrightarrow DomainModel$
dm_link_typ_inv1_3: $Concept\_parentConcept\_Concept \in Concept \nrightarrow Concept$
dm_link_typ_inv1_4: $Individual\_individualOf\_Concept \in Individual \rightarrow Concept$
dm_link_typ_inv1_5: $Individual\_initialValue\_Individual \in Individual \nrightarrow Individual$
dm_link_typ_inv1_6: $Association\_domain\_Concept \in Association \rightarrow Concept$
dm_link_typ_inv1_7: $Association\_range\_Concept \in Association \rightarrow Concept$
dm_link_typ_inv1_8: $Association\_DomainCardinality\_minCardinality \in Association \rightarrow \mathbb{N}$
dm_link_typ_inv1_9: $Association\_DomainCardinality\_maxCardinality \in Association \rightarrow (\mathbb{N} \cup \{-1\})$
dm_link_typ_inv1_10: $Association\_RangeCardinality\_minCardinality \in Association \rightarrow \mathbb{N}$
dm_link_typ_inv1_11: $Association\_RangeCardinality\_maxCardinality \in Association \rightarrow (\mathbb{N} \cup \{-1\})$
dm_link_typ_inv1_12: $MapletIndividual\_antecedent\_Individual \in MapletIndividual \nrightarrow Individual$
dm_link_typ_inv1_13: $MapletIndividual\_image\_Individual \in MapletIndividual \nrightarrow Individual$
dm_link_typ_inv1_14: $LogicFormula\_defines\_DefinedConcept \in LogicFormula \nrightarrow\!\!\!\!\!\rightarrow DefinedConcept$

Domain Model : various constraints

dm_constr_inv1_1: $MapletIndividual \lhd Individual\_individualOf\_Concept \in MapletIndividual \rightarrow Association$
dm_constr_inv1_2: $\forall ass \cdot (ass \in Association \Rightarrow (Association\_DomainCardinality\_maxCardinality(ass) = -1 \lor Association\_DomainCardinality\_minCardinality(ass) \leq Association\_DomainCardinality\_maxCardinality(ass)))$
dm_constr_inv1_3: $\forall ass \cdot (ass \in Association \Rightarrow (Association\_RangeCardinality\_maxCardinality(ass) = -1 \lor Association\_RangeCardinality\_minCardinality(ass) \leq Association\_RangeCardinality\_maxCardinality(ass)))$
dm_constr_inv1_4: $dom(MapletIndividual\_antecedent\_Individual) = dom(MapletIndividual\_image\_Individual)$
dm_constr_inv1_5: $\forall mi \cdot (mi \in dom(MapletIndividual\_antecedent\_Individual) \Rightarrow Individual\_individualOf\_Concept(MapletIndividual\_antecedent\_Individual(mi)) = Association\_domain\_Concept(Individual\_individualOf\_Concept(mi)))$
dm_constr_inv1_6: $\forall mi \cdot (mi \in dom(MapletIndividual\_image\_Individual) \Rightarrow Individual\_individualOf\_Concept(MapletIndividual\_image\_Individual(mi)) = Association\_range\_Concept(Individual\_individualOf\_Concept(mi)))$

dm_constr_inv1_7: $Concept\_isEnumeration(\_BOOL) = TRUE$

dm_constr_inv1_8: $\{\_TRUE, \_FALSE\} \subseteq Individual$

dm_constr_inv1_9: $Individual\_individualOf\_Concept(\_TRUE) = \_BOOL$

dm_constr_inv1_10: $Individual\_individualOf\_Concept(\_FALSE) = \_BOOL$

dm_constr_inv1_11: $DefaultDataType \lhd Concept\_definedIn\_DomainModel \in (Concept \setminus DefaultDataType) \rightarrow DomainModel$

dm_constr_inv1_12: $\{\_TRUE, \_FALSE\} \lhd Individual\_definedIn\_DomainModel \in (Individual \setminus \{\_TRUE, \_FALSE\}) \rightarrow DomainModel$

dm_constr_inv1_13: $\forall ind \cdot (ind \in Individual \setminus MapletIndividual \Rightarrow Individual\_isNamed(ind) = TRUE)$

dm_constr_inv1_14: $\forall ind \cdot ((ind \in Individual \wedge Individual\_isNamed(ind) = FALSE) \Rightarrow Individual\_isVariable(ind) = FALSE)$

dm_constr_inv1_15: $\forall ind \cdot (ind \in dom(MapletIndividual\_antecedent\_Individual) \cap dom(MapletIndividual\_image\_Individual) \Rightarrow (Individual\_isNamed(MapletIndividual\_antecedent\_Individual(ind)) = TRUE \wedge Individual\_isNamed(MapletIndividual\_image\_Individual(ind)) = TRUE))$

dm_constr_inv1_16: $\forall ind \cdot ((ind \in MapletIndividual \wedge Individual\_isNamed(ind) = FALSE) \Rightarrow ind \in dom(MapletIndividual\_antecedent\_Individual) \cap dom(MapletIndividual\_image\_Individual))$

dm_constr_inv1_17: $\forall co \cdot (co \in Concept \setminus (Association \cup DefinedConcept \cup dom(Concept\_parentConcept\_Concept)) \Rightarrow Concept\_isVariable(co) = FALSE)$

dm_constr_inv1_18: $\forall co \cdot ((co \in Concept \wedge Concept\_isEnumeration(co) = TRUE) \Rightarrow Concept\_isVariable(co) = FALSE)$

dm_constr_inv1_19: $\forall ind \cdot ((ind \in dom(MapletIndividual\_antecedent\_Individual) \cap dom(MapletIndividual\_image\_Individual) \wedge Individual\_isVariable(ind) = FALSE) \Rightarrow (Individual\_isVariable(MapletIndividual\_antecedent\_Individual(ind)) = FALSE \wedge Individual\_isVariable(MapletIndividual\_image\_Individual(ind)) = FALSE))$

dm_constr_inv1_20: $\forall co \cdot ((co \in Association \wedge Concept\_isVariable(co) = FALSE) \Rightarrow (Concept\_isVariable(Association\_domain\_Concept(co)) = FALSE \wedge Concept\_isVariable(Association\_range\_Concept(co)) = FALSE))$

dm_constr_inv1_21: $Concept\_parentConcept\_Concept \cap (Concept \lhd id) = \emptyset$ added to discharge a proof

dm_constr_inv1_22: $Association\_domain\_Concept \cap (Concept \lhd id) = \emptyset$ added to discharge a proof

dm_constr_inv1_23: $Association\_range\_Concept \cap (Concept \lhd id) = \emptyset$ added to discharge a proof

dm_constr_inv1_24: $Individual\_initialValue\_Individual \cap (Individual \lhd id) = \emptyset$ added to discharge a proof

dm_constr_inv1_25: $MapletIndividual\_antecedent\_Individual \cap (Individual \lhd id) = \emptyset$ added to discharge a proof

dm_constr_inv1_26: $MapletIndividual\_image\_Individual \cap (Individual \lhd id) = \emptyset$ added to discharge a proof

dm_constr_inv1_27: $\forall co \cdot ((co \in Concept \wedge Concept\_isEnumeration(co) = TRUE) \Rightarrow Individual\_individualOf\_Concept^{-1}[\{co\}] \cap Individual\_isVariable^{-1}[\{FALSE\}] \neq \emptyset)$ added to discharge a proof

Correspondence links : typing invariants

corr_link_typ_inv1_1: $Concept\_corresp\_Set \in Concept\_isVariable^{-1}[\{FALSE\}] \rightarrowtail\!\!\!\!\!\!\rightarrow Set$

corr_link_typ_inv1_2: $Association\_Type\_Constant \in Association \rightarrowtail\!\!\!\!\!\!\rightarrow Constant$

contient les types des associations; eg: LgOfLs_Type = LandingSet –> LandingGear

corr_link_typ_inv1_3: $Concept\_corresp\_Constant \in Concept\_isVariable^{-1}[\{FALSE\}] \rightarrowtail\!\!\!\!\!\!\rightarrow Constant$

corr_link_typ_inv1_4: $Concept\_corresp\_Variable \in Concept\_isVariable^{-1}[\{TRUE\}] \rightarrowtail\!\!\!\!\!\!\rightarrow Variable$

corr_link_typ_inv1_5: $Individual\_corresp\_Constant \in Individual\_isVariable^{-1}[\{FALSE\}] \rightarrowtail\!\!\!\!\!\!\rightarrow Constant$

corr_link_typ_inv1_6: $Individual\_corresp\_Variable \in Individual\_isVariable^{-1}[\{TRUE\}] \rightarrowtail\!\!\!\!\!\!\rightarrow Variable$

corr_link_typ_inv1_7: $Individual\_corresp\_SetItem \in (Individual\_individualOf\_Concept^{-1}[Concept\_isEnumeration^{-1}[\{TRUE\}]] \cap Individual\_isVariable^{-1}[\{FALSE\}]) \rightarrowtail\!\!\!\!\!\!\rightarrow SetItem$

corr_link_typ_inv1_8: $AssociationCharacteristic\_corresp\_LogicFormula \in (Association \nrightarrow RelationCharacteristics\_Set) \rightarrowtail\!\!\!\!\!\!\rightarrow LogicFormula$

corr_link_typ_inv1_9: $Association\_Type\_Variable \in Association \rightarrowtail\!\!\!\!\!\!\rightarrow Variable$

corr_link_typ_inv1_10: $ConcreteEnumeration\_corresp\_IndividualSetLogicalFormula \in (dom(Concept\_parentConcept\_Concept) \cap Concept\_isEnumeration^{-1}[\{TRUE\}] \cap dom(Concept\_corresp\_Constant)) \rightarrowtail\!\!\!\!\!\!\rightarrow LogicFormula$

corr_link_typ_inv1_11: $UnnamedMapletIndividual\_corresp\_LogicalFormula \in (MapletIndividual \cap Individual\_isNamed^{-1}[\{FALSE\}]) \rightarrowtail\!\!\!\!\!\!\rightarrow LogicFormula$

Correspondence links : various constraints

corr_link_constr_inv1_1: $Concept\_isEnumeration^{-1}[\{TRUE\}] \lhd Concept\_corresp\_Set \in Concept \rightarrowtail\!\!\!\!\!\!\rightarrow EnumeratedSet$

corr_link_constr_inv1_2: $Concept\_isEnumeration^{-1}[\{FALSE\}] \lhd Concept\_corresp\_Set \in Concept \rightarrowtail\!\!\!\!\!\!\rightarrow AbstractSet$

corr_link_constr_inv1_3: $partition(dom(Concept\_corresp\_Set) \cup dom(Concept\_corresp\_Constant) \cup dom(Concept\_corresp\_Variable), dom(Concept\_corresp\_Set), dom(Concept\_corresp\_Constant), dom(Concept\_corresp\_Variable))$

corr_link_constr_inv1_5: $dom(Individual\_corresp\_Variable) \cap Individual\_isNamed^{-1}[\{FALSE\}] = \emptyset$

corr_link_constr_inv1_6: $partition(dom(Individual\_corresp\_Constant) \cup dom(Individual\_corresp\_Variable) \cup dom(Individual\_corresp\_SetItem), dom(Individual\_corresp\_Constant), dom(Individual\_corresp\_Variable), dom(Individual\_corresp\_SetItem))$

corr_link_constr_inv1_7: $\forall xx \cdot ((xx \in Concept\_isEnumeration^{-1}[\{TRUE\}] \land xx \notin dom(Concept\_corresp\_Set \rhd EnumeratedSet)) \Rightarrow Individual\_individualOf\_Concept^{-1}[\{xx\}] \cap dom(Individual\_corresp\_SetItem) = \emptyset)$

corr_link_constr_inv1_8: $\{\_NATURAL \mapsto B\_NATURAL, \_INTEGER \mapsto B\_INTEGER, \_FLOAT \mapsto B\_FLOAT, \_BOOL \mapsto B\_BOOL, \_STRING \mapsto B\_STRING\} \subseteq Concept\_corresp\_Set$

corr_link_constr_inv1_9: $\{\_TRUE \mapsto B\_TRUE, \_FALSE \mapsto B\_FALSE\} \subseteq Individual\_corresp\_SetItem$

corr_link_constr_inv1_10: $\forall co \cdot (co \in Concept\_isEnumeration^{-1}[\{TRUE\}] \setminus (dom(Concept\_corresp\_Set) \cup dom(Concept\_parentConcept\_Concept) \cup Association \cup DefinedConcept \cup DefaultDataType) \Rightarrow (Individual\_individualOf\_Concept^{-1}[\{co\}] \cap Individual\_isVariable^{-1}[\{FALSE\}]) \cap (dom(Individual\_corresp\_SetItem) \cup dom(Individual\_corresp\_Constant)) = \emptyset)$ added to discharge a proof

corr_link_constr_inv1_11: $dom(Concept\_parentConcept\_Concept) \cap dom(Concept\_corresp\_Set) = \emptyset$ added to discharge a proof

corr_link_constr_inv1_12: $partition(ran(Concept\_corresp\_Constant) \cup ran(Individual\_corresp\_Constant) \cup ran(Association\_Type\_Constant), ran(Concept\_corresp\_Constant), ran(Individual\_corresp\_Constant), ran(Association\_Type\_Constant))$

corr_link_constr_inv1_13: $partition(ran(Concept\_corresp\_Variable) \cup ran(Individual\_corresp\_Variable) \cup ran(Association\_Type\_Variable), ran(Concept\_corresp\_Variable), ran(Individual\_corresp\_Variable), ran(Association\_Type\_Variable))$

corr_link_constr_inv1_14: $partition(dom(Association\_Type\_Constant) \cup dom(Association\_Type\_Variable), dom(Association\_Type\_Constant), dom(Association\_Type\_Variable))$

corr_link_constr_inv1_15: $(Concept \setminus (dom(Concept\_parentConcept\_Concept) \cup Association \cup DefinedConcept)) \cap (dom(Concept\_corresp\_Constant) \cup dom(Concept\_corresp\_Variable)) = \emptyset$

corr_link_constr_inv1_16: $\forall xx \cdot (xx \in (Association \cap (dom(Concept\_corresp\_Constant) \cup dom(Concept\_corresp\_Variable))) \setminus dom(Concept\_parentConcept\_Concept) \Rightarrow xx \in dom(Association\_Type\_Constant) \cup dom(Association\_Type\_Variable))$

corr_link_constr_inv1_17: $Association \cap dom(Concept\_corresp\_Set) = \emptyset$

corr_link_constr_inv1_18: $partition(ran(AssociationCharacteristic\_corresp\_LogicFormula) \cup ran(ConcreteEnumeration\_corresp\_IndividualSetLogicalFormula) \cup ran(UnnamedMapletIndividual\_corresp\_LogicalFormula), ran(AssociationCharacteristic\_corresp\_LogicFormula), ran(ConcreteEnumeration\_corresp\_IndividualSetLogicalFormula), ran(UnnamedMapletIndividual\_corresp\_LogicalFormula))$ added to discharge a proof

isomorphisms

constant subconcept linked to its abstract parent concept

isom_inv1_1: $\forall xx, pxx, o\_lg \cdot (\ xx \in dom(Concept\_parentConcept\_Concept) \land pxx = Concept\_parentConcept\_Concept(xx) \land xx \in dom(Concept\_corresp\_Constant) \land pxx \in dom(Concept\_corresp\_Set) \land o\_lg = Constant\_typing\_Property(Concept\_corresp\_Constant(xx))\ ) \Rightarrow (\ LogicFormula\_uses\_Operators(o\_lg) = \{1 \mapsto Inclusion\_OP\} \land (2 \mapsto Concept\_corresp\_Set(pxx)) \in LogicFormula\_involves\_Sets(o\_lg)\ )\ )$

Variable subconcept linked to its abstract parent concept

isom_inv1_1_2: $\forall xx, pxx, o\_lg \cdot (\ xx \in dom(Concept\_parentConcept\_Concept) \land pxx = Concept\_parentConcept\_Concept(xx) \land xx \in dom(Concept\_corresp\_Variable) \land pxx \in dom(Concept\_corresp\_Set) \land o\_lg = Variable\_typing\_Invariant(Concept\_corresp\_Variable(xx))\ ) \Rightarrow (\ LogicFormula\_uses\_Operators(o\_lg) = \{1 \mapsto Inclusion\_OP\} \land (2 \mapsto Concept\_corresp\_Set(pxx)) \in LogicFormula\_involves\_Sets(o\_lg)\ )\ )$

Constant subconcept linked to its concrete constant parent concept

isom_inv1_1_3: $\forall xx, pxx, o\_lg \cdot (\ xx \in dom(Concept\_parentConcept\_Concept) \land pxx = Concept\_parentConcept\_Concept(xx) \land xx \in dom(Concept\_corresp\_Constant) \land pxx \in dom(Concept\_corresp\_Constant) \land o\_lg = Constant\_typing\_Property(Concept\_corresp\_Constant(xx))\ ) \Rightarrow (\ LogicFormula\_uses\_Operators(o\_lg) = \{1 \mapsto Inclusion\_OP\} \land (2 \mapsto o\_lg) \in Constant\_isInvolvedIn\_LogicFormulas(Concept\_corresp\_Constant(pxx))\ )\ )$

Variable subconcept linked to its concrete constant parent concept

isom_inv1_1_4: $\forall xx, pxx, o\_lg \cdot (\ xx \in dom(Concept\_parentConcept\_Concept) \land pxx = Concept\_parentConcept\_Concept(xx) \land xx \in dom(Concept\_corresp\_Variable) \land pxx \in dom(Concept\_corresp\_Constant) \land o\_lg = Variable\_typing\_Invariant(Concept\_corresp\_Variable(xx))\ ) \Rightarrow (\ LogicFormula\_uses\_Operators(o\_lg) = \{1 \mapsto Inclusion\_OP\} \land (2 \mapsto o\_lg) \in Constant\_isInvolvedIn\_LogicFormulas(Concept\_corresp\_Constant(pxx))\ )\ )$

Constant subconcept linked to its concrete variable parent concept

isom_inv1_1_5: $\forall xx, pxx, o\_lg \cdot (\ xx \in dom(Concept\_parentConcept\_Concept)$
$\land\ pxx = Concept\_parentConcept\_Concept(xx)\ \land\ xx \in dom(Concept\_corresp\_Constant)$
$\land\ pxx \in dom(Concept\_corresp\_Variable)\ \land\ o\_lg = Constant\_typing\_Property(Concept\_corresp\_Constant(xx))\ )$
$\Rightarrow (\ LogicFormula\_uses\_Operators(o\_lg) = \{1 \mapsto Inclusion\_OP\}$
$\land\ \exists ppxx, o\_lg\_i \cdot (\ ppxx \in ran(Concept\_parentConcept\_Concept)$
$\land\ (\ (ppxx \in dom(Concept\_corresp\_Constant)$
$\land\ (2 \mapsto o\_lg) \in Constant\_isInvolvedIn\_LogicFormulas(Concept\_corresp\_Constant(ppxx)))$
$\lor\ (ppxx \in dom(Concept\_corresp\_Set) \land (2 \mapsto Concept\_corresp\_Set(ppxx)) \in LogicFormula\_involves\_Sets(o\_lg))\ )$
$\land\ o\_lg\_i \in Invariant\ \land\ LogicFormula\_uses\_Operators(o\_lg\_i) = \{1 \mapsto Inclusion\_OP\}$
$\land\ (1 \mapsto o\_lg\_i) \in Constant\_isInvolvedIn\_LogicFormulas(Concept\_corresp\_Constant(xx))$
$\land\ (2 \mapsto Concept\_corresp\_Variable(pxx)) \in Invariant\_involves\_Variables(o\_lg\_i)\ )\ )\ )$
$ppxx \in\ cls(Concept\_parentConcept\_Concept)[\{pxx\}]$
Variable subconcept linked to its concrete variable parent concept

isom_inv1_1_6: $\forall xx, pxx, o\_lg \cdot (\ xx \in dom(Concept\_parentConcept\_Concept)$
$\land\ pxx = Concept\_parentConcept\_Concept(xx)\ \land\ xx \in dom(Concept\_corresp\_Variable)$
$\land\ pxx \in dom(Concept\_corresp\_Variable)\ \land\ o\_lg = Variable\_typing\_Invariant(Concept\_corresp\_Variable(xx))\ )$
$\Rightarrow (\ LogicFormula\_uses\_Operators(o\_lg) = \{1 \mapsto Inclusion\_OP\}$
$\land\ (2 \mapsto Concept\_corresp\_Variable(pxx)) \in Invariant\_involves\_Variables(o\_lg)\ )\ )$
Constant included in an abstract set (the correspondence of the parent concept or of its ancestor)

isom_inv1_2: $\forall o\_xx, o\_pxx, o\_lg \cdot (\ o\_xx \in ran(Concept\_corresp\_Constant)$
$\land\ o\_lg = Constant\_typing\_Property(o\_xx)\ \land\ LogicFormula\_uses\_Operators(o\_lg) = \{1 \mapsto Inclusion\_OP\}$
$\land\ o\_pxx \in ran(Concept\_corresp\_Set)\ \land\ (2 \mapsto o\_pxx) \in LogicFormula\_involves\_Sets(o\_lg)\ )$
$\Rightarrow (\ Concept\_corresp\_Constant^{-1}(o\_xx) \in dom(Concept\_parentConcept\_Concept)$
$\land\ (\ Concept\_corresp\_Set^{-1}(o\_pxx) = Concept\_parentConcept\_Concept(Concept\_corresp\_Constant^{-1}(o\_xx))$
$\lor\ (\exists o\_lg\_i, o\_pxx\_v \cdot (\ o\_lg\_i \in Invariant$
$\land\ o\_pxx\_v \in ran(Concept\_corresp\_Variable)$
$\land\ Concept\_corresp\_Variable^{-1}(o\_pxx\_v) = Concept\_parentConcept\_Concept(Concept\_corresp\_Constant^{-1}(o\_xx))$
$\land\ Concept\_corresp\_Set^{-1}(o\_pxx) \in ran(Concept\_parentConcept\_Concept)$
$\land\ LogicFormula\_uses\_Operators(o\_lg\_i) = \{1 \mapsto Inclusion\_OP\}$
$\land\ (1 \mapsto o\_lg\_i) \in Constant\_isInvolvedIn\_LogicFormulas(o\_xx)$
$\land\ (2 \mapsto o\_pxx\_v) \in Invariant\_involves\_Variables(o\_lg\_i)\ )\ )\ )\ )$
$Concept\_corresp\_Set^{-1}(o\_pxx) \in\ cls(Concept\_parentConcept\_Concept)[\{Concept\_corresp\_Variable^{-1}(o\_pxx\_v)\}]$
Constant included in another constant (the correspondence of the parent concept or of its ancestor)

isom_inv1_2_2: $\forall o\_xx, o\_pxx, o\_lg \cdot (\ o\_xx \in ran(Concept\_corresp\_Constant)$
$\land\ o\_lg = Constant\_typing\_Property(o\_xx)\ \land\ LogicFormula\_uses\_Operators(o\_lg) = \{1 \mapsto Inclusion\_OP\}$
$\land\ o\_pxx \in ran(Concept\_corresp\_Constant)\ \land\ (2 \mapsto o\_lg) \in Constant\_isInvolvedIn\_LogicFormulas(o\_pxx)\ )$
$\Rightarrow (\ Concept\_corresp\_Constant^{-1}(o\_xx) \in dom(Concept\_parentConcept\_Concept)$
$\land\ (\ Concept\_corresp\_Constant^{-1}(o\_pxx) = Concept\_parentConcept\_Concept(Concept\_corresp\_Constant^{-1}(o\_xx))$
$\lor\ (\exists o\_lg\_i, o\_pxx\_v \cdot (\ o\_lg\_i \in Invariant\ \land\ o\_pxx\_v \in ran(Concept\_corresp\_Variable)$
$\land\ Concept\_corresp\_Variable^{-1}(o\_pxx\_v) = Concept\_parentConcept\_Concept(Concept\_corresp\_Constant^{-1}(o\_xx))$
$\land\ Concept\_corresp\_Constant^{-1}(o\_pxx) \in ran(Concept\_parentConcept\_Concept)$
$\land\ LogicFormula\_uses\_Operators(o\_lg\_i) = \{1 \mapsto Inclusion\_OP\}$
$\land\ (1 \mapsto o\_lg\_i) \in Constant\_isInvolvedIn\_LogicFormulas(o\_xx)$
$\land\ (2 \mapsto o\_pxx\_v) \in Invariant\_involves\_Variables(o\_lg\_i)\ )\ )\ )\ )$
$\land Concept\_corresp\_Constant^{-1}(o\_pxx) \in\ cls(Concept\_parentConcept\_Concept)[\{Concept\_corresp\_Variable^{-1}(o\_pxx\_v)\}]$
Variable included in an abstract set

isom_inv1_2_3: $\forall o\_xx, o\_pxx, o\_lg \cdot (\ o\_xx \in ran(Concept\_corresp\_Variable)$
$\land\ o\_lg = Variable\_typing\_Invariant(o\_xx)\ \land\ LogicFormula\_uses\_Operators(o\_lg) = \{1 \mapsto Inclusion\_OP\}$
$\land\ o\_pxx \in ran(Concept\_corresp\_Set)\ \land\ (2 \mapsto o\_pxx) \in LogicFormula\_involves\_Sets(o\_lg)\ )$
$\Rightarrow (\ Concept\_corresp\_Variable^{-1}(o\_xx) \in dom(Concept\_parentConcept\_Concept)$
$\land\ Concept\_corresp\_Set^{-1}(o\_pxx) = Concept\_parentConcept\_Concept(Concept\_corresp\_Variable^{-1}(o\_xx)))\ )$
Variable included in a constant

isom_inv1_2_4: $\forall o\_xx, o\_pxx, o\_lg \cdot (\ o\_xx \in ran(Concept\_corresp\_Variable)$
$\land\ o\_lg = Variable\_typing\_Invariant(o\_xx)\ \land\ LogicFormula\_uses\_Operators(o\_lg) = \{1 \mapsto Inclusion\_OP\}$
$\land\ o\_pxx \in ran(Concept\_corresp\_Constant)\ \land\ (2 \mapsto o\_lg) \in Constant\_isInvolvedIn\_LogicFormulas(o\_pxx)\ )$
$\Rightarrow (\ Concept\_corresp\_Variable^{-1}(o\_xx) \in dom(Concept\_parentConcept\_Concept)$
$\land\ Concept\_corresp\_Constant^{-1}(o\_pxx) = Concept\_parentConcept\_Concept(Concept\_corresp\_Variable^{-1}(o\_xx)))\ )$
Variable included in a variable

isom_inv1_2_5: $\forall o\_xx, o\_pxx, o\_lg \cdot ((o\_xx \in ran(Concept\_corresp\_Variable)$
$\land o\_lg = Variable\_typing\_Invariant(o\_xx) \land LogicFormula\_uses\_Operators(o\_lg) = \{1 \mapsto Inclusion\_OP\}$
$\land o\_pxx \in ran(Concept\_corresp\_Variable) \land (2 \mapsto o\_pxx) \in Invariant\_involves\_Variables(o\_lg))$
$\Rightarrow (Concept\_corresp\_Variable^{-1}(o\_xx) \in dom(Concept\_parentConcept\_Concept)$
$\land Concept\_corresp\_Variable^{-1}(o\_pxx) = Concept\_parentConcept\_Concept(Concept\_corresp\_Variable^{-1}(o\_xx))))$
invariants required to discharge isomorphisms
each concept is translated after its parent concept

isom_inv1_3: $\forall xx, pxx \cdot ((xx \in dom(Concept\_parentConcept\_Concept)$
$\land pxx = Concept\_parentConcept\_Concept(xx) \land xx \in (dom(Concept\_corresp\_Constant) \cup dom(Concept\_corresp\_Variable)))$
$\Rightarrow pxx \in (dom(Concept\_corresp\_Set) \cup dom(Concept\_corresp\_Constant) \cup dom(Concept\_corresp\_Variable)))$
each constant is back propagated after its type (abstract set)

isom_inv1_4: $\forall o\_xx, o\_pxx, o\_lg \cdot ((o\_xx \in Constant \land o\_lg = Constant\_typing\_Property(o\_xx)$
$\land LogicFormula\_uses\_Operators(o\_lg) = \{1 \mapsto Inclusion\_OP\} \land (2 \mapsto o\_pxx) \in LogicFormula\_involves\_Sets(o\_lg)$
$\land o\_xx \in ran(Concept\_corresp\_Constant))$
$\Rightarrow o\_pxx \in ran(Concept\_corresp\_Set))$
each constant is back propagated after its type (constant)

isom_inv1_4_2: $\forall o\_xx, o\_pxx, o\_lg \cdot ((o\_xx \in Constant \land o\_lg = Constant\_typing\_Property(o\_xx)$
$\land LogicFormula\_uses\_Operators(o\_lg) = \{1 \mapsto Inclusion\_OP\} \land o\_pxx \in Constant$
$\land (2 \mapsto o\_lg) \in Constant\_isInvolvedIn\_LogicFormulas(o\_pxx) \land o\_xx \in ran(Concept\_corresp\_Constant))$
$\Rightarrow o\_pxx \in ran(Concept\_corresp\_Constant))$
each variable is back propagated after its type (abstract set)

isom_inv1_4_3: $\forall o\_xx, o\_pxx, o\_lg \cdot ((o\_xx \in Variable \land o\_lg = Variable\_typing\_Invariant(o\_xx)$
$\land LogicFormula\_uses\_Operators(o\_lg) = \{1 \mapsto Inclusion\_OP\}$
$\land (2 \mapsto o\_pxx) \in LogicFormula\_involves\_Sets(o\_lg)$
$\land o\_xx \in ran(Concept\_corresp\_Variable))$
$\Rightarrow o\_pxx \in ran(Concept\_corresp\_Set))$
each variable is back propagated after its type (constant)

isom_inv1_4_4: $\forall o\_xx, o\_pxx, o\_lg \cdot ((o\_xx \in Variable \land o\_lg = Variable\_typing\_Invariant(o\_xx)$
$\land LogicFormula\_uses\_Operators(o\_lg) = \{1 \mapsto Inclusion\_OP\}$
$\land o\_pxx \in Constant \land (2 \mapsto o\_lg) \in Constant\_isInvolvedIn\_LogicFormulas(o\_pxx)$
$\land o\_xx \in ran(Concept\_corresp\_Variable))$
$\Rightarrow o\_pxx \in ran(Concept\_corresp\_Constant))$
each variable is back propagated after its type (variable)

isom_inv1_4_5: $\forall o\_xx, o\_pxx, o\_lg \cdot ((o\_xx \in Variable \land o\_lg = Variable\_typing\_Invariant(o\_xx)$
$\land LogicFormula\_uses\_Operators(o\_lg) = \{1 \mapsto Inclusion\_OP\}$
$\land (2 \mapsto o\_pxx) \in Invariant\_involves\_Variables(o\_lg)$
$\land o\_xx \in ran(Concept\_corresp\_Variable))$
$\Rightarrow o\_pxx \in ran(Concept\_corresp\_Variable))$

theo_var_nat: ⟨theorem⟩ $card(Concept \setminus (dom(Concept\_corresp\_Set)$
$\cup dom(Concept\_corresp\_Constant) \cup dom(Concept\_corresp\_Variable)))$
$+ card(Individual \setminus (dom(Individual\_corresp\_Constant)$
$\cup dom(Individual\_corresp\_Variable) \cup dom(Individual\_corresp\_SetItem)))$
$+ card(Set \setminus ran(Concept\_corresp\_Set))$
$+ card(SetItem \setminus ran(Individual\_corresp\_SetItem))$
$+ card(Constant \setminus (ran(Concept\_corresp\_Constant)$
$\cup ran(Individual\_corresp\_Constant) \cup ran(Association\_Type\_Constant)))$
$+ card(Variable \setminus (ran(Concept\_corresp\_Variable)$
$\cup ran(Individual\_corresp\_Variable) \cup ran(Association\_Type\_Variable)))$
$+ card(Association \setminus (dom(Association\_Type\_Constant) \cup dom(Association\_Type\_Variable)))$
$+ card((dom(Concept\_parentConcept\_Concept)$
$\cap Concept\_isEnumeration^{-1}[\{TRUE\}]) \setminus dom(Concrete\_Enumeration\_corresp\_IndividualSetLogicalFormula))$
$+ card(MapletIndividual \setminus dom(UnnamedMapletIndividual\_corresp\_LogicalFormula))$
$+ card((Association \nrightarrow RelationCharacteristics\_Set) \setminus dom(AssociationCharacteristic\_corresp\_LogicFormula)) \in \mathbb{N}$

VARIANT

$card(Concept \setminus (dom(Concept\_corresp\_Set)$
$\cup dom(Concept\_corresp\_Constant) \cup dom(Concept\_corresp\_Variable)))$
$+ card(Individual \setminus (dom(Individual\_corresp\_Constant)$
$\cup dom(Individual\_corresp\_Variable) \cup dom(Individual\_corresp\_SetItem)))$
$+ card(Set \setminus ran(Concept\_corresp\_Set))$

$+ card(SetItem \setminus ran(Individual\_corresp\_SetItem))$
$+ card(Constant \setminus (ran(Concept\_corresp\_Constant)$
$\cup\ ran(Individual\_corresp\_Constant) \cup ran(Association\_Type\_Constant)))$
$+ card(Variable \setminus (ran(Concept\_corresp\_Variable)$
$\cup\ ran(Individual\_corresp\_Variable) \cup ran(Association\_Type\_Variable)))$
$+ card(Association \setminus (dom(Association\_Type\_Constant) \cup dom(Association\_Type\_Variable)))$
$+ card((dom(Concept\_parentConcept\_Concept)$
$\cap\ Concept\_isEnumeration^{-1}[\{TRUE\}]) \setminus dom(Concrete\_Enumeration\_corresp\_IndividualSetLogicalFormula))$
$+ card(MapletIndividual \setminus dom(UnnamedMapletIndividual\_corresp\_LogicalFormula))$
$+ card((Association \nrightarrow RelationCharacteristics\_Set) \setminus dom(AssociationCharacteristic\_corresp\_LogicFormula))$

END

MACHINE event_b_specs_from_ontologies_ref_2
REFINES event_b_specs_from_ontologies_ref_1
SEES EventB_Metamodel_Context,Domain_Metamodel_Context
VARIABLES InitialisationAction InitialisationAction_uses_Operators InitialisationAction_inits_Variable
InitialisationAction_involves_Constants
INVARIANTS

bs_elt_typ_inv2_1: $InitialisationAction \subseteq InitialisationAction\_Set$
B System links : typing invariants
bs_link_typ_inv2_1: $InitialisationAction\_uses\_Operators \in InitialisationAction \rightarrow (\mathbb{N}_1 \nrightarrow Operator)$
bs_link_typ_inv2_2: $InitialisationAction\_inits\_Variable \in InitialisationAction \rightarrowtail Variable$
for initialisation actions, the assigned operand is the involved variable.
bs_link_typ_inv2_3: $InitialisationAction\_involves\_Constants \in InitialisationAction \rightarrow (\mathbb{N}_1 \nrightarrow Constant)$
theo_var_nat: ⟨theorem⟩ $card((ran(Concept\_corresp\_Variable)$
$\cup\ ran(Individual\_corresp\_Variable)) \setminus ran(InitialisationAction\_inits\_Variable)) \in \mathbb{N}$

VARIANT

$card((ran(Concept\_corresp\_Variable)$
$\cup\ ran(Individual\_corresp\_Variable)) \setminus ran(InitialisationAction\_inits\_Variable))$

END

B.3 Definition of the Adjusted Translation Rules

B.3.1 Informal Definition

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

Table B.2 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* [56] built on top of *Jetbrains MPS* [87] and PlantUML [116] to provide a proof of concept of the SysML/KAOS Domain Modeling Language. They have also been implemented within the *Openflexo* platform [109] which federates the various contributions of *FORMOSE* project partners [17]. Rules *3*, *4*, *6*..*8*, and *12*..*16* have undergone significant updates to the previously defined translation rules (see annex A).

Table B.2 – The adjusted 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))[$\{PCO\}$] [1] $o\_PPCO \in$ 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 [2] $\forall j \in 1..n, o\_I_j \in o\_CO$		**Property:** $o\_CO = (o\_I_j)_{j\in 1..n}$

1. *closure1(Concept_parent_Concept)* designates the transitive closure of relation Concept_parent_Concept

2. Every concrete enumeration is a constant

(9)	**Association or defined concept without parent**	CO	$CO \in$ (DefinedConcept $\cup$ Association) $CO \notin dom$(Concept_parent_Concept) [3] 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) da = Association_DomainCardinality_maxCardinality(AS) di = Association_DomainCardinality_minCardinality(AS) ra = Association_RangeCardinality_maxCardinality(AS) ri = Association_RangeCardinality_minCardinality(AS) $o\_AS \in$ Constant $\cup$ Variable $\{o\_CO1, o\_CO2\} \subseteq$ (Set $\cup$ Constant $\cup$ Variable)	T_o_AS	**IF** Concept_isVariable($CO1$) = $FALSE$ $\wedge$ 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:** $T\_o\_AS = o\_CO1 \twoheadrightarrow\!\!\!\!\!\twoheadrightarrow o\_CO2$ **ELSE IF** $\{ra, da\} = \{1\}$ **THEN** **LogicFormula:** $T\_o\_AS = o\_CO1 \rightarrowtail\!\!\!\!\!\rightarrowtail o\_CO2$ **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:** $o\_AS \in T\_o\_AS$
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 $\cup$ Constant	o_Ind	**IF** Individual_isVariable(Ind) = $TRUE$ **THEN** $o\_Ind \in$ Variable **ELSE** $o\_Ind \in$ Constant **LogicFormula:** $o\_Ind \in o\_CO$

3. 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.

12	**Constant individual of a variable concept**	Ind CO PPCO	*Ind* ∈ Individual \ MapletIndividual Individual_isVariable(*Ind*) = *FALSE* *CO* = Individual_individualOf_Concept(*Ind*) *o_CO* ∈ Variable *PPCO* ∈ Concept *PPCO* ∈ (*closure*1(Concept_parent_Concept))[{*CO*}] *o_PPCO* ∈ Set ∪ Constant	o_Ind	*o_Ind* ∈ Constant **Property:** *o_Ind* ∈ *o_PPCO* **Invariant:** *o_Ind* ∈ *o_CO*
13	**Variable individual of a variable concept**	Ind CO	*Ind* ∈ Individual \ MapletIndividual Individual_isVariable(*Ind*) = *TRUE* *CO* = Individual_individualOf_Concept(*Ind*) *o_CO* ∈ Variable	o_Ind	*o_Ind* ∈ Variable **Invariant:** *o_Ind* ∈ *o_CO*
14	**Variable individual of a concept that is an abstract enumeration**	Ind CO	*Ind* ∈ Individual \ MapletIndividual Individual_isVariable(*Ind*) = *TRUE* *CO* = Individual_individualOf_Concept(*Ind*) Concept_isEnumeration(*CO*) = *TRUE* *CO* ∉ *dom*(Concept_parent_Concept) *o_CO* ∈ EnumeratedSet	o_Ind	*o_Ind* ∈ Variable **Invariant:** *o_Ind* ∈ *o_CO*
15	**Maplet individual**	Ind AS Ant Im PPCO1 PPCO2	*Ind* ∈ MapletIndividual *AS* = Individual_individualOf_Concept(*Ind*)[4] *o_AS* ∈ Constant ∪ Variable *Ind* ∈ *dom*(MapletIndividual_antecedent_Individual) ⇒ *Ant* = MapletIndividual_antecedent_Individual(*Ind*) *o_Ant* ∈ Constant ∪ Variable *Ind* ∈ *dom*(MapletIndividual_image_Individual) ⇒ *Im* = MapletIndividual_image_Individual(*Ind*) *o_Im* ∈ Constant ∪ Variable {*PPCO*1, *PPCO*2} ⊆ Concept *PPCO*1 ∈ (*closure*1(Concept_parent_Concept))[{Association_domain_Concept(*AS*)}] *PPCO*2 ∈ (*closure*1(Concept_parent_Concept))[{MapletIndividual_range_Individual(*AS*)}] {*o_PPCO*1, *o_PPCO*2} ⊆ Set ∪ Constant	o_Ind	**IF** *Ind* ∈ *dom*(Individual_name) **THEN** **IF** Individual_isVariable(*Ind*) = *TRUE* **THEN** *o_Ind* ∈ Variable **Invariant:** *o_Ind* ∈ *o_AS* **IF** *Ind* ∈ *dom*(MapletIndividual_antecedent_Individual) ∩*dom*(MapletIndividual_image_Individual) **THEN Invariant:** *o_Ind* = *o_Ant* ↦ *o_Im* **ELSE** *o_Ind* ∈ Constant **IF** *o_AS* ∈ Constant **THEN Property:** *o_Ind* ∈ *o_AS* **ELSE** **Property:** *o_Ind* ∈ *o_PPCO*1 ↔ *o_PPCO*2 **Invariant:** *o_Ind* ∈ *o_AS* **IF** *Ind* ∈ *dom*(MapletIndividual_antecedent_Individual) ∩*dom*(MapletIndividual_image_Individual) **THEN Property:** *o_Ind* = *o_Ant* ↦ *o_Im* **ELSE LogicFormula:** *o_Ant* ↦ *o_Im* ∈ *o_AS*[5]

4. *AS* must be an association
5. Following the variability status of *o_AS*, this predicate can be a property or an invariant

16	**Variable individual initialisation**	Ind Init CO Init_ant Init_im	$Ind \in$ Individual $\cap$ dom(*Individual_name*) *Individual_isVariable*(*Ind*) = *TRUE* *o_Ind* $\in$ Variable *CO* = Individual_individualOf_Concept(*Ind*) *o_CO* $\in$ Set $\cup$ Constant $\cup$ Variable *Ind* $\notin$ *dom*(Individual_initialValue_individual) $\vee$(Individual_initialValue_individual(*Ind*) = *Init* $\wedge$ ((*Init* $\notin$ *dom*(Individual_name) $\wedge$ *Init_ant* = MapletIndividual_antecedent_Individual(*Init*) $\wedge$ *Init_im* = MapletIndividual_image_Individual(*Init*) $\wedge$ {*Init_ant*, *Init_im*} $\subseteq$ Constant $\cup$ Variable) $\vee$ *o_Init* $\in$ Constant $\cup$ 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 $\wedge$ Individual_individualOf_Concept(I_j) = *CO* $\wedge$ Individual_isVariable(I_j) = *FALSE* *o_CO* $\in$ Variable $\forall j \in 1..n, o\_I_j \in o\_CO$		**Initialisation:** $o\_CO := (o\_I_j)_{j\in 1..n}$ [6]
18	**Association transitivity**	AS	*AS* $\in$ Association Association_isTransitive(*AS*) = *TRUE* *o_AS* $\in$ Constant $\cup$ Variable		**LogicFormula:** (*o_AS* ; *o_AS*) $\subseteq$ *o_AS*
19	**Association symmetry**	AS	*AS* $\in$ Association Association_isSymmetric(*AS*) = *TRUE* *o_AS* $\in$ Constant $\cup$ Variable		**LogicFormula:** $o\_AS^{-1} = o\_AS$
20	**Association asymmetry**	AS CO	*AS* $\in$ Association Association_isSymmetric(*AS*) = *TRUE* *o_AS* $\in$ Constant $\cup$ Variable Association_domain_Concept(*AS*) = *CO* *o_CO* $\in$ Set $\cup$ Constant $\cup$ 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 $\cup$ Variable Association_domain_Concept(*AS*) = *CO* *o_CO* $\in$ Set $\cup$ Constant $\cup$ Variable		**LogicFormula:** $id(o\_CO) \subseteq o\_AS$
22	**Association irreflexivity**	AS CO	*AS* $\in$ Association Association_isIrreflexive(*AS*) = *TRUE* *o_AS* $\in$ Constant $\cup$ Variable Association_domain_Concept(*AS*) = *CO* *o_CO* $\in$ Set $\cup$ Constant $\cup$ Variable		**LogicFormula:** $id(o\_CO) \cap o\_AS = \emptyset$

6. 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

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.

B.3.2 Event-B Specification

MACHINE event_b_specs_from_ontologies_ref_1
REFINES event_b_specs_from_ontologies
SEES EventB_Metamodel_Context,Domain_Metamodel_Context
EVENTS
Event rule_3 ⟨convergent⟩ $\widehat{=}$

Abstract concept that is not an enumeration

any CO o_CO
where

grd0: $Concept\_isEnumeration^{-1}[\{FALSE\}]\setminus(dom(Concept\_corresp\_Set)\cup dom(Concept\_parentConcept\_Concept)\cup Association \cup DefinedConcept \cup DefaultDataType) \neq \emptyset$

grd1: $CO \in Concept\_isEnumeration^{-1}[\{FALSE\}]\setminus(dom(Concept\_corresp\_Set)\cup dom(Concept\_parentConcept\_Concept)\cup Association \cup DefinedConcept \cup DefaultDataType)$

grd2: $Concept\_definedIn\_DomainModel(CO) \in dom(DomainModel\_corresp\_Component)$

grd3: $Set\_Set \setminus Set \neq \emptyset$

grd4: $o\_CO \in Set\_Set \setminus Set$

then

act1: $AbstractSet := AbstractSet \cup \{o\_CO\}$

act2: $Set := Set \cup \{o\_CO\}$

act3: $Concept\_corresp\_Set(CO) := o\_CO$

act4: $Set\_definedIn\_Component(o\_CO) := DomainModel\_corresp\_Component(Concept\_definedIn\_DomainModel(CO))$

end

Event rule_4 ⟨convergent⟩ $\widehat{=}$

Abstract concept that is an enumeration

any CO o_CO elements o_elements mapping_elements_o_elements
where

grd0: $Concept\_isEnumeration^{-1}[\{TRUE\}]\setminus(dom(Concept\_corresp\_Set)\cup dom(Concept\_parentConcept\_Concept)\cup Association \cup DefinedConcept \cup DefaultDataType) \neq \emptyset$

grd1: $CO \in Concept\_isEnumeration^{-1}[\{TRUE\}] \setminus (dom(Concept\_corresp\_Set) \cup dom(Concept\_parentConcept\_Concept) \cup Association \cup DefinedConcept \cup DefaultDataType)$

grd2: $Concept\_definedIn\_DomainModel(CO) \in dom(DomainModel\_corresp\_Component)$

grd3: $Set\_Set \setminus Set \neq \emptyset$

grd4: $o\_CO \in Set\_Set \setminus Set$

grd5: $o\_elements \subseteq SetItem\_Set$

grd6: $o\_elements \cap SetItem = \emptyset$

grd7: $elements = (Individual\_individualOf\_Concept^{-1}[\{CO\}] \cap Individual\_isVariable^{-1}[\{FALSE\}])$

grd8: $card(o\_elements) = card(elements)$

grd9: $mapping\_elements\_o\_elements \in elements \rightarrowtail\!\!\!\!\!\!\rightarrow o\_elements$

then

act1: $EnumeratedSet := EnumeratedSet \cup \{o\_CO\}$

act2: $Set := Set \cup \{o\_CO\}$

act3: $Concept\_corresp\_Set(CO) := o\_CO$

act4: $Set\_definedIn\_Component(o\_CO) := DomainModel\_corresp\_Component(Concept\_definedIn\_DomainModel(CO))$

act5: $SetItem := SetItem \cup o\_elements$

@act6 SetItem_itemOf_EnumeratedSet := SetItem_itemOf_EnumeratedSet ∪ {(xx↦ yy) |xx∈ o_elements ∧yy=o_CO}

act6: $SetItem\_itemOf\_EnumeratedSet := SetItem\_itemOf\_EnumeratedSet \cup (\lambda xx \cdot xx \in o\_elements | o\_CO)$

act7: $Individual\_corresp\_SetItem := Individual\_corresp\_SetItem \cup mapping\_elements\_o\_elements$

end

Event rule_5 ⟨convergent⟩ $\widehat{=}$

Concept with constant parent

any CO o_CO_c o_CO_v PCO o_lg o_PCO_s o_PCO_c

where

grd0: $dom(Concept\_parentConcept\_Concept) \setminus (DefaultDataType \cup dom(Concept\_corresp\_Constant) \cup dom(Concept\_corresp\_Variable)) \neq \emptyset$

grd1: $CO \in dom(Concept\_parentConcept\_Concept) \setminus (DefaultDataType \cup dom(Concept\_corresp\_Constant) \cup dom(Concept\_corresp\_Variable))$

grd2: $dom(Concept\_corresp\_Set) \cup dom(Concept\_corresp\_Constant) \neq \emptyset$

grd3: $PCO \in dom(Concept\_corresp\_Set) \cup dom(Concept\_corresp\_Constant)$

grd4: $Concept\_parentConcept\_Concept(CO) = PCO$

grd5: $Concept\_definedIn\_DomainModel(CO) \in dom(DomainModel\_corresp\_Component)$

grd6: $Concept\_isVariable(CO) = FALSE \Rightarrow ((Constant\_Set \setminus Constant \neq \emptyset) \wedge (o\_CO\_c \in Constant\_Set \setminus Constant))$

grd7: $Concept\_isVariable(CO) = TRUE \Rightarrow ((Variable\_Set \setminus Variable \neq \emptyset) \wedge (o\_CO\_v \in Variable\_Set \setminus Variable))$

grd8: $LogicFormula\_Set \setminus LogicFormula \neq \emptyset$

grd9: $o\_lg \in LogicFormula\_Set \setminus LogicFormula$

grd10: $o\_PCO\_s \in Set \wedge (PCO \in dom(Concept\_corresp\_Set) \Rightarrow o\_PCO\_s = Concept\_corresp\_Set(PCO))$

grd11: $o\_PCO\_c \in Constant \wedge (PCO \in dom(Concept\_corresp\_Constant) \Rightarrow o\_PCO\_c = Concept\_corresp\_Constant(PCO))$

then

act1: $Constant := \{TRUE \mapsto Constant, FALSE \mapsto Constant \cup \{o\_CO\_c\}\}(bool(Concept\_isVariable(CO) = TRUE))$

act2: $Variable := \{TRUE \mapsto Variable \cup \{o\_CO\_v\}, FALSE \mapsto Variable\}(bool(Concept\_isVariable(CO) = TRUE))$

act3: $Concept\_corresp\_Constant := \{TRUE \mapsto Concept\_corresp\_Constant, FALSE \mapsto Concept\_corresp\_Constant \cup \{CO \mapsto o\_CO\_c\}\}(bool(Concept\_isVariable(CO) = TRUE))$

act4: $Concept\_corresp\_Variable := \{TRUE \mapsto Concept\_corresp\_Variable \cup \{CO \mapsto o\_CO\_v\}, FALSE \mapsto Concept\_corresp\_Variable\}(bool(Concept\_isVariable(CO) = TRUE))$

act5: $Constant\_definedIn\_Component := \{TRUE \mapsto Constant\_definedIn\_Component, FALSE \mapsto Constant\_definedIn\_Component \cup \{o\_CO\_c \mapsto DomainModel\_corresp\_Component(Concept\_definedIn\_DomainModel(CO))\}\}(bool(Concept\_isVariable(CO) = TRUE))$

act6: $Variable\_definedIn\_Component := \{TRUE \mapsto Variable\_definedIn\_Component \cup \{o\_CO\_v \mapsto DomainModel\_corresp\_Component(Concept\_definedIn\_DomainModel(CO))\}, FALSE \mapsto Variable\_definedIn\_Component\}(bool(Concept\_isVariable(CO) = TRUE))$

act7: $Property := \{TRUE \mapsto Property, FALSE \mapsto Property \cup \{o\_lg\}\}(bool(Concept\_isVariable(CO) = TRUE))$

act8: $Invariant := \{TRUE \mapsto Invariant \cup \{o\_lg\}, FALSE \mapsto Invariant\}(bool(Concept\_isVariable(CO) = TRUE))$

act9: $LogicFormula := LogicFormula \cup \{o\_lg\}$

act10: $LogicFormula\_uses\_Operators(o\_lg) := \{1 \mapsto Inclusion\_OP\}$

act11:
$Constant\_isInvolvedIn\_LogicFormulas := \{TRUE \mapsto (\{TRUE \mapsto Constant\_isInvolvedIn\_LogicFormulas, FALSE \mapsto Constant\_isInvolvedIn\_LogicFormulas \Leftarrow \{o\_PCO\_c \mapsto Constant\_isInvolvedIn\_LogicFormulas(o\_PCO\_c) \cup \{2 \mapsto o\_lg\}\}\}(bool(PCO \in dom(Concept\_corresp\_Set)))),$
$FALSE \mapsto (\{TRUE \mapsto Constant\_isInvolvedIn\_LogicFormulas \cup \{o\_CO\_c \mapsto \{1 \mapsto o\_lg\}\},$
$FALSE \mapsto Constant\_isInvolvedIn\_LogicFormulas \Leftarrow \{(o\_CO\_c \mapsto \{1 \mapsto o\_lg\}),$
$o\_PCO\_c \mapsto Constant\_isInvolvedIn\_LogicFormulas(o\_PCO\_c) \cup \{2 \mapsto o\_lg\}\}\}(bool(PCO \in dom(Concept\_corresp\_Set)))))\}$
$(bool(Concept\_isVariable(CO) = TRUE))$

act12: $Invariant\_involves\_Variables := \{TRUE \mapsto Invariant\_involves\_Variables \cup \{o\_lg \mapsto \{1 \mapsto o\_CO\_v\}\}, FALSE \mapsto Invariant\_involves\_Variables\}(bool(Concept\_isVariable(CO) = TRUE))$

act13: $LogicFormula\_involves\_Sets := \{TRUE \mapsto LogicFormula\_involves\_Sets \cup \{o\_lg \mapsto \{2 \mapsto o\_PCO\_s\}\}, FALSE \mapsto LogicFormula\_involves\_Sets \cup \{o\_lg \mapsto \emptyset\}\}(bool(PCO \in dom(Concept\_corresp\_Set)))$

act14: $LogicFormula\_definedIn\_Component(o\_lg) := DomainModel\_corresp\_Component(Concept\_definedIn\_DomainModel(CO))$

act15: $Constant\_typing\_Property := \{TRUE \mapsto Constant\_typing\_Property, FALSE \mapsto Constant\_typing\_Property \cup \{o\_CO\_c \mapsto o\_lg\}\}(bool(Concept\_isVariable(CO) = TRUE))$

act16: $Variable\_typing\_Invariant := \{TRUE \mapsto Variable\_typing\_Invariant \cup \{o\_CO\_v \mapsto o\_lg\}, FALSE \mapsto Variable\_typing\_Invariant\}(bool(Concept\_isVariable(CO) = TRUE))$

end

Event rule_6 ⟨convergent⟩ $\widehat{=}$

Constant concept with variable parent

any CO o_CO PCO o_PCO PPCO o_lg_p o_lg_i o_PPCO_s o_PPCO_c

where

grd0: $(dom(Concept\_parentConcept\_Concept) \cap Concept\_isVariable^{-1}[\{FALSE\}]) \setminus (DefaultDataType \cup dom(Concept\_corresp\_Constant)) \neq \emptyset$

grd1: $CO \in (dom(Concept\_parentConcept\_Concept) \cap Concept\_isVariable^{-1}[\{FALSE\}]) \setminus (DefaultDataType \cup dom(Concept\_corresp\_Constant))$

grd2: $Concept\_parentConcept\_Concept(CO) = PCO$

grd3: $PCO \in dom(Concept\_corresp\_Variable)$

grd4: $o\_PCO = Concept\_corresp\_Variable(PCO)$

grd5: $dom(Concept\_corresp\_Set) \cup dom(Concept\_corresp\_Constant) \neq \emptyset$

grd6: $PPCO \in dom(Concept\_corresp\_Set) \cup dom(Concept\_corresp\_Constant)$

grd7: $PPCO \in cls(Concept\_parentConcept\_Concept)[\{PCO\}]$

grd7_1: $PPCO \in ran(Concept\_parentConcept\_Concept)$

grd8: $Concept\_definedIn\_DomainModel(CO) \in dom(DomainModel\_corresp\_Component)$

grd9: $(Constant\_Set \setminus Constant \neq \emptyset) \wedge (o\_CO \in Constant\_Set \setminus Constant)$

grd10: $LogicFormula\_Set \setminus LogicFormula \neq \emptyset$

grd11: $\{o\_lg\_p, o\_lg\_i\} \subseteq LogicFormula\_Set \setminus LogicFormula$

grd12: $o\_lg\_p \neq o\_lg\_i$

grd13: $o\_PPCO\_s \in Set \wedge (PPCO \in dom(Concept\_corresp\_Set) \Rightarrow o\_PPCO\_s = Concept\_corresp\_Set(PPCO))$

grd14: $o\_PPCO\_c \in Constant \wedge (PPCO \in dom(Concept\_corresp\_Constant) \Rightarrow o\_PPCO\_c = Concept\_corresp\_Constant(PPCO))$

then

act1: $Constant := Constant \cup \{o\_CO\}$

act2: $Concept\_corresp\_Constant(CO) := o\_CO$

act3: $Constant\_definedIn\_Component(o\_CO) := DomainModel\_corresp\_Component(Concept\_definedIn\_DomainModel(CO))$

act4: $Property := Property \cup \{o\_lg\_p\}$

act5: $Invariant := Invariant \cup \{o\_lg\_i\}$

act6: $LogicFormula := LogicFormula \cup \{o\_lg\_p, o\_lg\_i\}$

act7: $LogicFormula\_uses\_Operators := LogicFormula\_uses\_Operators \cup (\{o\_lg\_p, o\_lg\_i\} \times \{\{1 \mapsto Inclusion\_OP\}\})$

act8: $Constant\_isInvolvedIn\_LogicFormulas := Constant\_isInvolvedIn\_LogicFormulas \Leftarrow (\{o\_CO \mapsto \{1 \mapsto o\_lg\_p, 1 \mapsto o\_lg\_i\}\} \cup (\{TRUE \mapsto \emptyset, FALSE \mapsto \{o\_PPCO\_c \mapsto Constant\_isInvolvedIn\_LogicFormulas(o\_PPCO\_c) \cup \{2 \mapsto o\_lg\_p\}\}\}(bool(PPCO \in dom(Concept\_corresp\_Set)))))$

act9: $Invariant\_involves\_Variables(o\_lg\_i) := \{2 \mapsto o\_PCO\}$

act10: $LogicFormula\_involves\_Sets := LogicFormula\_involves\_Sets \cup \{o\_lg\_i \mapsto \emptyset\} \cup \{TRUE \mapsto \{o\_lg\_p \mapsto \{2 \mapsto o\_PPCO\_s\}\}, FALSE \mapsto \{o\_lg\_p \mapsto \emptyset\}\}(bool(PPCO \in dom(Concept\_corresp\_Set)))$

act11: $LogicFormula\_definedIn\_Component := LogicFormula\_definedIn\_Component \cup (\{o\_lg\_p, o\_lg\_i\} \times \{DomainModel\_corresp\_Component(Concept\_definedIn\_DomainModel(CO))\})$

act12: $Constant\_typing\_Property(o\_CO) := o\_lg\_p$

end

Event rule_7 ⟨convergent⟩ $\widehat{=}$

Variable concept with variable parent

any CO o_CO PCO o_lg o_PCO

where

grd0: $(dom(Concept\_parentConcept\_Concept) \cap Concept\_isVariable^{-1}[\{TRUE\}]) \setminus (DefaultDataType \cup dom(Concept\_corresp\_Variable)) \neq \emptyset$

grd1: $CO \in (dom(Concept\_parentConcept\_Concept) \cap Concept\_isVariable^{-1}[\{TRUE\}]) \setminus (DefaultDataType \cup dom(Concept\_corresp\_Variable))$

grd2: $Concept\_parentConcept\_Concept(CO) = PCO$

grd3: $PCO \in dom(Concept\_corresp\_Variable)$

grd4: $o\_PCO = Concept\_corresp\_Variable(PCO)$

grd5: $Concept\_definedIn\_DomainModel(CO) \in dom(DomainModel\_corresp\_Component)$

grd6: $Variable\_Set \setminus Variable \neq \emptyset$

grd7: $o\_CO \in Variable\_Set \setminus Variable$

grd8: $LogicFormula\_Set \setminus LogicFormula \neq \emptyset$

grd9: $o\_lg \in LogicFormula\_Set \setminus LogicFormula$

then

act1: $Variable := Variable \cup \{o\_CO\}$

act2: $Concept\_corresp\_Variable(CO) := o\_CO$

act3: $Variable\_definedIn\_Component(o\_CO) := DomainModel\_corresp\_Component(Concept\_definedIn\_DomainModel(CO))$

act4: $Invariant := Invariant \cup \{o\_lg\}$

act5: $LogicFormula := LogicFormula \cup \{o\_lg\}$

act6: $LogicFormula\_uses\_Operators(o\_lg) := \{1 \mapsto Inclusion\_OP\}$

act7: $Invariant\_involves\_Variables(o\_lg) := \{1 \mapsto o\_CO, 2 \mapsto o\_PCO\}$

act8: $LogicFormula\_definedIn\_Component(o\_lg) := DomainModel\_corresp\_Component(Concept\_definedIn\_DomainModel(CO))$

act9: $Variable\_typing\_Invariant(o\_CO) := o\_lg$

act10: $LogicFormula\_involves\_Sets(o\_lg) := \emptyset$

end

Event rule_8 ⟨convergent⟩ $\widehat{=}$

Enumerated concept with parent

any CO o_CO o_lg inds o_inds bij_o_inds

where

grd0: $(dom(Concept\_parentConcept\_Concept) \cap (Concept\_isEnumeration^{-1}[\{TRUE\}]) \cap dom(Concept\_corresp\_Constant)) \setminus (DefaultDataType \cup dom(ConcreteEnumeration\_corresp\_IndividualSetLogicalFormula)) \neq \emptyset$

grd1: $CO \in (dom(Concept\_parentConcept\_Concept) \cap Concept\_isEnumeration^{-1}[\{TRUE\}] \cap dom(Concept\_corresp\_Constant)) \setminus (DefaultDataType \cup dom(ConcreteEnumeration\_corresp\_IndividualSetLogicalFormula))$

grd2: $Concept\_definedIn\_DomainModel(CO) \in dom(DomainModel\_corresp\_Component)$

grd3: $o\_CO = Concept\_corresp\_Constant(CO)$

grd4: $LogicFormula\_Set \setminus LogicFormula \neq \emptyset$

grd5: $o\_lg \in LogicFormula\_Set \setminus LogicFormula$

grd6: $inds = Individual\_individualOf\_Concept^{-1}[\{CO\}] \cap Individual\_isVariable^{-1}[\{FALSE\}]$

grd7: $inds \subseteq dom(Individual\_corresp\_Constant)$

grd8: $o\_inds = Individual\_corresp\_Constant[inds]$

grd9: $card(o\_inds) = card(inds)$

grd10: $bij\_o\_inds \in o\_inds \rightarrowtail\!\!\!\!\!\rightarrow 2 \,..\, (card(o\_inds) + 1)$

then

act0: $ConcreteEnumeration\_corresp\_IndividualSetLogicalFormula(CO) := o\_lg$

act1: $Property := Property \cup \{o\_lg\}$

act2: $LogicFormula := LogicFormula \cup \{o\_lg\}$

act3: $LogicFormula\_uses\_Operators(o\_lg) := \{1 \mapsto Equal2SetOf\_OP\}$

act4: $Constant\_isInvolvedIn\_LogicFormulas := Constant\_isInvolvedIn\_LogicFormulas \Leftarrow ((\{o\_CO \mapsto Constant\_isInvolvedIn\_LogicFormulas(o\_CO) \cup \{1 \mapsto o\_lg\}\}) \cup (\lambda o\_ind \cdot o\_ind \in o\_inds | Constant\_isInvolvedIn\_LogicFormulas(o\_ind) \cup \{bij\_o\_inds(o\_ind) \mapsto o\_lg\}))$

act5: $LogicFormula\_involves\_Sets(o\_lg) := \emptyset$

act6: $LogicFormula\_definedIn\_Component(o\_lg) := DomainModel\_corresp\_Component(Concept\_definedIn\_DomainModel(CO))$

end

Event rule_9_10 ⟨convergent⟩ $\widehat{=}$

correspondence of an instance of association

any AS o_AS_c o_AS_v T_AS_c T_AS_v CO1 o_CO1_s o_CO1_c o_CO1_v CO2 o_CO2_s o_CO2_c o_CO2_v o_lg_type o_lg_item o_DM

where

grd0: $((Association \setminus dom(Concept\_parentConcept\_Concept)) \cup (Association \cap dom(Concept\_parentConcept\_Concept) \cap (dom(Concept\_corresp\_Constant) \cup dom(Concept\_corresp\_Variable)))) \setminus (dom(Association\_Type\_Constant) \cup dom(Association\_Type\_Variable)) \neq \emptyset$

grd1: $AS \in ((Association \setminus dom(Concept\_parentConcept\_Concept)) \cup (Association \cap dom(Concept\_parentConcept\_Concept) \cap (dom(Concept\_corresp\_Constant) \cup dom(Concept\_corresp\_Variable)))) \setminus (dom(Association\_Type\_Constant) \cup dom(Association\_Type\_Variable))$

grd2: $AS \in dom(Concept\_corresp\_Constant) \Rightarrow o\_AS\_c = Concept\_corresp\_Constant(AS)$

grd3: $AS \in dom(Concept\_corresp\_Variable) \Rightarrow o\_AS\_v = Concept\_corresp\_Variable(AS)$

grd4: $AS \notin (dom(Concept\_corresp\_Constant) \cup dom(Concept\_corresp\_Variable)) \Rightarrow ($
$(Concept\_isVariable(AS) = FALSE \Rightarrow ((Constant\_Set \setminus Constant \neq \emptyset) \wedge (o\_AS\_c \in Constant\_Set \setminus Constant)))$
$\wedge (Concept\_isVariable(AS) = TRUE \Rightarrow ((Variable\_Set \setminus Variable \neq \emptyset) \wedge (o\_AS\_v \in Variable\_Set \setminus Variable)))$
$)$

grd5: $CO1 = Association\_domain\_Concept(AS)$

grd6: $CO2 = Association\_range\_Concept(AS)$

grd7: $(Concept\_isVariable(CO1) = FALSE \wedge Concept\_isVariable(CO2) = FALSE)$
$\Rightarrow ((Constant\_Set \setminus Constant \neq \emptyset) \wedge (T\_AS\_c \in Constant\_Set \setminus Constant))$

grd8: $(Concept\_isVariable(CO1) = TRUE \vee Concept\_isVariable(CO2) = TRUE)$
$\Rightarrow ((Variable\_Set \setminus Variable \neq \emptyset) \wedge (T\_AS\_v \in Variable\_Set \setminus Variable))$

grd9: $dom(Concept\_corresp\_Set) \cup dom(Concept\_corresp\_Constant) \cup dom(Concept\_corresp\_Variable) \neq \emptyset$

grd10: $\{CO1, CO2\} \subseteq dom(Concept\_corresp\_Set) \cup dom(Concept\_corresp\_Constant)$
$\cup dom(Concept\_corresp\_Variable)$

grd11: $o\_CO1\_s \in Set \wedge (CO1 \in dom(Concept\_corresp\_Set) \Rightarrow o\_CO1\_s = Concept\_corresp\_Set(CO1))$

grd12: $o\_CO1\_c \in Constant \wedge (CO1 \in dom(Concept\_corresp\_Constant)$
$\Rightarrow o\_CO1\_c = Concept\_corresp\_Constant(CO1))$

grd13: $o\_CO1\_v \in Variable \wedge (CO1 \in dom(Concept\_corresp\_Variable)$
$\Rightarrow o\_CO1\_v = Concept\_corresp\_Variable(CO1))$

grd14: $o\_CO2\_s \in Set \wedge (CO2 \in dom(Concept\_corresp\_Set) \Rightarrow o\_CO2\_s = Concept\_corresp\_Set(CO2))$

grd15: $o\_CO2\_c \in Constant \land (CO2 \in dom(Concept\_corresp\_Constant) \Rightarrow o\_CO2\_c = Concept\_corresp\_Constant(CO2))$

grd16: $o\_CO2\_v \in Variable \land (CO2 \in dom(Concept\_corresp\_Variable) \Rightarrow o\_CO2\_v = Concept\_corresp\_Variable(CO2))$

grd17: $Concept\_definedIn\_DomainModel(AS) \in dom(DomainModel\_corresp\_Component)$

grd18: $LogicFormula\_Set \setminus LogicFormula \neq \emptyset$

grd19: $\{o\_lg\_type, o\_lg\_item\} \subseteq LogicFormula\_Set \setminus LogicFormula$

grd20: $o\_DM = DomainModel\_corresp\_Component(Concept\_definedIn\_DomainModel(AS))$

grd21: $partition(\{T\_AS\_c, o\_AS\_c, o\_CO1\_c, o\_CO2\_c\}, \{T\_AS\_c\}, \{o\_AS\_c\}, \{o\_CO1\_c, o\_CO2\_c\})$

grd22: $partition(\{T\_AS\_v, o\_AS\_v, o\_CO1\_v, o\_CO2\_v\}, \{T\_AS\_v\}, \{o\_AS\_v\}, \{o\_CO1\_v, o\_CO2\_v\})$

grd23: $o\_lg\_type \neq o\_lg\_item$

then

act1: $Constant := Constant \cup (\{TRUE \mapsto \{T\_AS\_c\}, FALSE \mapsto \emptyset\}(bool(Concept\_isVariable(CO1) = FALSE \land Concept\_isVariable(CO2) = FALSE))) \cup (\{TRUE \mapsto (\{TRUE \mapsto \{o\_AS\_c\}, FALSE \mapsto \emptyset\}(bool(Concept\_isVariable(AS) = FALSE))), FALSE \mapsto \emptyset\}(bool(AS \notin (dom(Concept\_corresp\_Constant) \cup dom(Concept\_corresp\_Variable)))))$

act2: $Variable := Variable \cup (\{TRUE \mapsto \emptyset, FALSE \mapsto \{T\_AS\_v\}\}(bool(Concept\_isVariable(CO1) = FALSE \land Concept\_isVariable(CO2) = FALSE))) \cup (\{TRUE \mapsto (\{TRUE \mapsto \emptyset, FALSE \mapsto \{o\_AS\_v\}\}(bool(Concept\_isVariable(AS) = FALSE))), FALSE \mapsto \emptyset\}(bool(AS \notin (dom(Concept\_corresp\_Constant) \cup dom(Concept\_corresp\_Variable)))))$

act3: $Concept\_corresp\_Constant := Concept\_corresp\_Constant \cup (\{TRUE \mapsto (\{TRUE \mapsto \{AS \mapsto o\_AS\_c\}, FALSE \mapsto \emptyset\}(bool(Concept\_isVariable(AS) = FALSE))), FALSE \mapsto \emptyset\}(bool(AS \notin (dom(Concept\_corresp\_Constant) \cup dom(Concept\_corresp\_Variable)))))$

act4: $Concept\_corresp\_Variable := Concept\_corresp\_Variable \cup (\{TRUE \mapsto (\{TRUE \mapsto \emptyset, FALSE \mapsto \{AS \mapsto o\_AS\_v\}\}(bool(Concept\_isVariable(AS) = FALSE))), FALSE \mapsto \emptyset\}(bool(AS \notin (dom(Concept\_corresp\_Constant) \cup dom(Concept\_corresp\_Variable)))))$

act5: $Association\_Type\_Constant := Association\_Type\_Constant \cup (\{TRUE \mapsto \{AS \mapsto T\_AS\_c\}, FALSE \mapsto \emptyset\}(bool(Concept\_isVariable(CO1) = FALSE \land Concept\_isVariable(CO2) = FALSE)))$

act6: $Association\_Type\_Variable := Association\_Type\_Variable \cup (\{TRUE \mapsto \emptyset, FALSE \mapsto \{AS \mapsto T\_AS\_v\}\}(bool(Concept\_isVariable(CO1) = FALSE \land Concept\_isVariable(CO2) = FALSE)))$

act7: $Constant\_definedIn\_Component := Constant\_definedIn\_Component \cup (\{TRUE \mapsto \{T\_AS\_c \mapsto o\_DM\}, FALSE \mapsto \emptyset\}(bool(Concept\_isVariable(CO1) = FALSE \land Concept\_isVariable(CO2) = FALSE))) \cup (\{TRUE \mapsto (\{TRUE \mapsto \{o\_AS\_c \mapsto o\_DM\}, FALSE \mapsto \emptyset\}(bool(Concept\_isVariable(AS) = FALSE))), FALSE \mapsto \emptyset\}(bool(AS \notin (dom(Concept\_corresp\_Constant) \cup dom(Concept\_corresp\_Variable)))))$

act8: $Variable\_definedIn\_Component := Variable\_definedIn\_Component \cup (\{TRUE \mapsto \emptyset, FALSE \mapsto \{T\_AS\_v \mapsto o\_DM\}\}(bool(Concept\_isVariable(CO1) = FALSE \land Concept\_isVariable(CO2) = FALSE))) \cup (\{TRUE \mapsto (\{TRUE \mapsto \emptyset, FALSE \mapsto \{o\_AS\_v \mapsto o\_DM\}\}(bool(Concept\_isVariable(AS) = FALSE))), FALSE \mapsto \emptyset\}(bool(AS \notin (dom(Concept\_corresp\_Constant) \cup dom(Concept\_corresp\_Variable)))))$

B.3. Definition of the Adjusted Translation Rules

act9: $Property := Property \cup (\{TRUE \mapsto \{o\_lg\_type, o\_lg\_item\}, FALSE \mapsto (\{TRUE \mapsto \{o\_lg\_type\}, FALSE \mapsto \emptyset\}(bool(Concept\_isVariable(CO1) = FALSE \wedge Concept\_isVariable(CO2) = FALSE)))\}(bool(Concept\_isVariable(AS) = FALSE)))$

act10: $Invariant := Invariant \cup (\{TRUE \mapsto (\{TRUE \mapsto \emptyset, FALSE \mapsto \{o\_lg\_item\}\}(bool(Concept\_isVariable(AS) = FALSE))), FALSE \mapsto \{o\_lg\_type, o\_lg\_item\}\}(bool(Concept\_isVariable(CO1) = FALSE \wedge Concept\_isVariable(CO2) = FALSE)))$

act11: $LogicFormula := LogicFormula \cup \{o\_lg\_type, o\_lg\_item\}$

act12: $Constant\_typing\_Property := Constant\_typing\_Property \cup (\{TRUE \mapsto \{T\_AS\_c \mapsto o\_lg\_type\}, FALSE \mapsto \emptyset\}(bool(Concept\_isVariable(CO1) = FALSE \wedge Concept\_isVariable(CO2) = FALSE))) \cup (\{TRUE \mapsto (\{TRUE \mapsto \{o\_AS\_c \mapsto o\_lg\_item\}, FALSE \mapsto \emptyset\}(bool(Concept\_isVariable(AS) = FALSE))), FALSE \mapsto \emptyset\}(bool(AS \notin (dom(Concept\_corresp\_Constant) \cup dom(Concept\_corresp\_Variable)))))$

act13: $Variable\_typing\_Invariant := Variable\_typing\_Invariant \cup (\{TRUE \mapsto \emptyset, FALSE \mapsto \{T\_AS\_v \mapsto o\_lg\_type\}\}(bool(Concept\_isVariable(CO1) = FALSE \wedge Concept\_isVariable(CO2) = FALSE))) \cup (\{TRUE \mapsto (\{TRUE \mapsto \emptyset, FALSE \mapsto \{o\_AS\_v \mapsto o\_lg\_item\}\}(bool(Concept\_isVariable(AS) = FALSE))), FALSE \mapsto \emptyset\}(bool(AS \notin (dom(Concept\_corresp\_Constant) \cup dom(Concept\_corresp\_Variable)))))$

act14: $LogicFormula\_uses\_Operators := LogicFormula\_uses\_Operators \cup \{o\_lg\_type \mapsto \{1 \mapsto RelationSet\_OP\}, o\_lg\_item \mapsto \{1 \mapsto Belonging\_OP\}\}$

act15: $Constant\_isInvolvedIn\_LogicFormulas := Constant\_isInvolvedIn\_LogicFormulas \Leftarrow ((\{TRUE \mapsto \{T\_AS\_c \mapsto \{1 \mapsto o\_lg\_type, 2 \mapsto o\_lg\_item\}\}, FALSE \mapsto \emptyset\}(bool(Concept\_isVariable(CO1) = FALSE \wedge Concept\_isVariable(CO2) = FALSE))) \cup (\{TRUE \mapsto (\{TRUE \mapsto \{o\_AS\_c \mapsto \{1 \mapsto o\_lg\_item\}\}, FALSE \mapsto \emptyset\}(bool(Concept\_isVariable(AS) = FALSE))), FALSE \mapsto (\{TRUE \mapsto \{o\_AS\_c \mapsto union(Constant\_isInvolvedIn\_LogicFormulas[\{o\_AS\_c\}]) \cup \{1 \mapsto o\_lg\_item\}\}, FALSE \mapsto \emptyset\}(bool(AS \in dom(Concept\_corresp\_Constant))))\}(bool(AS \notin (dom(Concept\_corresp\_Constant) \cup dom(Concept\_corresp\_Variable))))) \cup (\{TRUE \mapsto (\{TRUE \mapsto (\{TRUE \mapsto \{o\_CO1\_c \mapsto Constant\_isInvolvedIn\_LogicFormulas(o\_CO1\_c) \cup (\{1,2\} \times \{o\_lg\_type\})\}, FALSE \mapsto \{o\_CO1\_c \mapsto Constant\_isInvolvedIn\_LogicFormulas(o\_CO1\_c) \cup \{1 \mapsto o\_lg\_type\}, o\_CO2\_c \mapsto Constant\_isInvolvedIn\_LogicFormulas(o\_CO2\_c) \cup \{2 \mapsto o\_lg\_type\}\}\}(bool(CO1 = CO2))), FALSE \mapsto \{o\_CO1\_c \mapsto Constant\_isInvolvedIn\_LogicFormulas(o\_CO1\_c) \cup \{1 \mapsto o\_lg\_type\}\}\}(bool(CO2 \in dom(Concept\_corresp\_Constant)))), FALSE \mapsto (\{TRUE \mapsto \{o\_CO2\_c \mapsto Constant\_isInvolvedIn\_LogicFormulas(o\_CO2\_c) \cup \{2 \mapsto o\_lg\_type\}\}, FALSE \mapsto \emptyset\}(bool(CO2 \in dom(Concept\_corresp\_Constant))))\}(bool(CO1 \in dom(Concept\_corresp\_Constant)))))$

act16: $Invariant\_involves\_Variables := Invariant\_involves\_Variables \cup (\{TRUE \mapsto (\{TRUE \mapsto \emptyset, FALSE \mapsto \{o\_lg\_item \mapsto \{1 \mapsto o\_AS\_v\}\}\}(bool(Concept\_isVariable(AS) = FALSE))), FALSE \mapsto \{ o\_lg\_item \mapsto \{1 \mapsto o\_AS\_v, 2 \mapsto T\_AS\_v\}, o\_lg\_type \mapsto (\{1 \mapsto T\_AS\_v\} \cup (\{TRUE \mapsto \{2 \mapsto o\_CO1\_v\}, FALSE \mapsto \emptyset\}(bool(CO1 \in dom(Concept\_corresp\_Variable)))) \cup (\{TRUE \mapsto \{3 \mapsto o\_CO2\_v\}, FALSE \mapsto \emptyset\}(bool(CO2 \in dom(Concept\_corresp\_Variable))))) \}\}(bool(Concept\_isVariable(CO1) = FALSE \wedge Concept\_isVariable(CO2) = FALSE)))$

act17: $LogicFormula\_involves\_Sets := LogicFormula\_involves\_Sets \cup \{o\_lg\_item \mapsto \emptyset\} \cup \{o\_lg\_type \mapsto ((\{TRUE \mapsto \{2 \mapsto o\_CO1\_s\}, FALSE \mapsto \emptyset\}(bool(CO1 \in dom(Concept\_corresp\_Set)))) \cup (\{TRUE \mapsto \{3 \mapsto o\_CO2\_s\}, FALSE \mapsto \emptyset\}(bool(CO2 \in dom(Concept\_corresp\_Set)))))\}$

act18: $LogicFormula\_definedIn\_Component := LogicFormula\_definedIn\_Component \cup (\{o\_lg\_type, o\_lg\_item\} \times \{o\_DM\})$

end

Event rule_11 ⟨convergent⟩ $\widehat{=}$

Individual of a constant concept that is not an abstract enumeration

any ind o_ind_c o_ind_v CO o_lg o_CO_s o_CO_c

where

grd0: $Individual \setminus (MapletIndividual \cup dom(Individual\_corresp\_Constant) \cup dom(Individual\_corresp\_Variable)) \neq \emptyset$

grd1: $ind \in Individual \setminus (MapletIndividual \cup dom(Individual\_corresp\_Constant) \cup dom(Individual\_corresp\_Variable))$

grd2: $dom(Concept\_corresp\_Set \rhd AbstractSet) \cup dom(Concept\_corresp\_Constant) \neq \emptyset$

grd3: $CO \in dom(Concept\_corresp\_Set \rhd AbstractSet) \cup dom(Concept\_corresp\_Constant)$

grd4: $Individual\_individualOf\_Concept(ind) = CO$

grd5: $Individual\_definedIn\_DomainModel(ind) \in dom(DomainModel\_corresp\_Component)$

grd6: $Individual\_isVariable(ind) = FALSE \Rightarrow ((Constant\_Set \setminus Constant \neq \emptyset) \wedge (o\_ind\_c \in Constant\_Set \setminus Constant))$

grd7: $Individual\_isVariable(ind) = TRUE \Rightarrow ((Variable\_Set \setminus Variable \neq \emptyset) \wedge (o\_ind\_v \in Variable\_Set \setminus Variable))$

grd8: $LogicFormula\_Set \setminus LogicFormula \neq \emptyset$

grd9: $o\_lg \in LogicFormula\_Set \setminus LogicFormula$

grd10: $o\_CO\_s \in Set \wedge (CO \in dom(Concept\_corresp\_Set) \Rightarrow o\_CO\_s = Concept\_corresp\_Set(CO))$

grd11: $o\_CO\_c \in Constant \wedge (CO \in dom(Concept\_corresp\_Constant) \Rightarrow o\_CO\_c = Concept\_corresp\_Constant(CO))$

then

act1: $Constant := \{TRUE \mapsto Constant, FALSE \mapsto Constant \cup \{o\_ind\_c\}\}(bool(Individual\_isVariable(ind) = TRUE))$

act2: $Variable := \{TRUE \mapsto Variable \cup \{o\_ind\_v\}, FALSE \mapsto Variable\}(bool(Individual\_isVariable(ind) = TRUE))$

act3: $Individual\_corresp\_Constant := \{TRUE \mapsto Individual\_corresp\_Constant, FALSE \mapsto Individual\_corresp\_Constant \cup \{ind \mapsto o\_ind\_c\}\}(bool(Individual\_isVariable(ind) = TRUE))$

act4: $Individual\_corresp\_Variable := \{TRUE \mapsto Individual\_corresp\_Variable \cup \{ind \mapsto o\_ind\_v\}, FALSE \mapsto Individual\_corresp\_Variable\}(bool(Individual\_isVariable(ind) = TRUE))$

act5: $Constant\_definedIn\_Component := \{TRUE \mapsto Constant\_definedIn\_Component, FALSE \mapsto Constant\_definedIn\_Component \cup \{o\_ind\_c \mapsto DomainModel\_corresp\_Component(Individual\_definedIn\_DomainModel(ind))\}\}(bool(Individual\_isVariable(ind) = TRUE))$

act6: $Variable\_definedIn\_Component := \{TRUE \mapsto Variable\_definedIn\_Component \cup \{o\_ind\_v \mapsto DomainModel\_corresp\_Component(Individual\_definedIn\_DomainModel(ind))\}, FALSE \mapsto Variable\_definedIn\_Component\}(bool(Individual\_isVariable(ind) = TRUE))$

act7: $Property := \{TRUE \mapsto Property, FALSE \mapsto Property \cup \{o\_lg\}\}(bool(Individual\_isVariable(ind) = TRUE))$

act8: $Invariant := \{TRUE \mapsto Invariant \cup \{o\_lg\}, FALSE \mapsto Invariant\}(bool(Individual\_isVariable(ind) = TRUE))$

act9: $LogicFormula := LogicFormula \cup \{o\_lg\}$

act10: $LogicFormula\_uses\_Operators(o\_lg) := \{1 \mapsto Belonging\_OP\}$

act11:

$Constant\_isInvolvedIn\_LogicFormulas := \{TRUE \mapsto (\{TRUE \mapsto Constant\_isInvolvedIn\_LogicFormulas, FALSE \mapsto Constant\_isInvolvedIn\_LogicFormulas \Leftarrow \{o\_CO\_c \mapsto Constant\_isInvolvedIn\_LogicFormulas(o\_CO\_c) \cup \{2 \mapsto o\_lg\}\}\}(bool(CO \in dom(Concept\_corresp\_Set)))), FALSE \mapsto (\{TRUE \mapsto Constant\_isInvolvedIn\_LogicFormulas \cup \{o\_ind\_c \mapsto \{1 \mapsto o\_lg\}\}, FALSE \mapsto Constant\_isInvolvedIn\_LogicFormulas \Leftarrow \{(o\_ind\_c \mapsto \{1 \mapsto o\_lg\}), o\_CO\_c \mapsto Constant\_isInvolvedIn\_LogicFormulas(o\_CO\_c) \cup \{2 \mapsto o\_lg\}\}\}(bool(CO \in dom(Concept\_corresp\_Set)))))\}(bool(Individual\_isVariable(ind) = TRUE))$

act12: $Invariant\_involves\_Variables := \{TRUE \mapsto Invariant\_involves\_Variables \cup \{o\_lg \mapsto \{1 \mapsto o\_ind\_v\}\}, FALSE \mapsto Invariant\_involves\_Variables\}(bool(Individual\_isVariable(ind) = TRUE))$

act13: $LogicFormula\_involves\_Sets := \{TRUE \mapsto LogicFormula\_involves\_Sets \cup \{o\_lg \mapsto \{2 \mapsto o\_CO\_s\}\}, FALSE \mapsto LogicFormula\_involves\_Sets \cup \{o\_lg \mapsto \emptyset\}\}(bool(CO \in dom(Concept\_corresp\_Set)))$

act14: $LogicFormula\_definedIn\_Component(o\_lg) := DomainModel\_corresp\_Component(Individual\_definedIn\_DomainModel(ind))$

act15: $Constant\_typing\_Property := \{TRUE \mapsto Constant\_typing\_Property, FALSE \mapsto Constant\_typing\_Property \cup \{o\_ind\_c \mapsto o\_lg\}\}(bool(Individual\_isVariable(ind) = TRUE))$

act16: $Variable\_typing\_Invariant := \{TRUE \mapsto Variable\_typing\_Invariant \cup \{o\_ind\_v \mapsto o\_lg\}, FALSE \mapsto Variable\_typing\_Invariant\}(bool(Individual\_isVariable(ind) = TRUE))$

end

Event rule_12 ⟨convergent⟩ $\widehat{=}$

Constant individual of a variable concept

any ind o_ind CO o_CO PPCO o_lg_p o_lg_i o_PPCO_s o_PPCO_c

where

grd0: $(Individual \cap Individual\_isVariable^{-1}[\{FALSE\}]) \setminus (MapletIndividual \cup dom(Individual\_corresp\_Constant)) \neq \emptyset$

grd1: $ind \in (Individual \cap Individual\_isVariable^{-1}[\{FALSE\}]) \setminus (MapletIndividual \cup dom(Individual\_corresp\_Constant))$

grd2: $Individual\_individualOf\_Concept(ind) = CO$

grd3: $CO \in dom(Concept\_corresp\_Variable)$

grd4: $o\_CO = Concept\_corresp\_Variable(CO)$

grd5: $dom(Concept\_corresp\_Set) \cup dom(Concept\_corresp\_Constant) \neq \emptyset$

grd6: $PPCO \in dom(Concept\_corresp\_Set) \cup dom(Concept\_corresp\_Constant)$

grd7: $PPCO \in cls(Concept\_parentConcept\_Concept)[\{CO\}]$

grd7_1: $PPCO \in ran(Concept\_parentConcept\_Concept)$

grd8: $Individual\_definedIn\_DomainModel(ind) \in dom(DomainModel\_corresp\_Component)$

grd9: $(Constant\_Set \setminus Constant \neq \emptyset) \wedge (o\_ind \in Constant\_Set \setminus Constant)$

grd10: $LogicFormula\_Set \setminus LogicFormula \neq \emptyset$

grd11: $\{o\_lg\_p, o\_lg\_i\} \subseteq LogicFormula\_Set \setminus LogicFormula$

grd12: $o\_lg\_p \neq o\_lg\_i$

grd13: $o\_PPCO\_s \in Set \wedge (PPCO \in dom(Concept\_corresp\_Set) \Rightarrow o\_PPCO\_s = Concept\_corresp\_Set(PPCO))$

grd14: $o\_PPCO\_c \in Constant \wedge (PPCO \in dom(Concept\_corresp\_Constant) \Rightarrow o\_PPCO\_c = Concept\_corresp\_Constant(PPCO))$

then

act1: $Constant := Constant \cup \{o\_ind\}$

act2: $Individual\_corresp\_Constant(ind) := o\_ind$

act3: $Constant\_definedIn\_Component(o\_ind) := DomainModel\_corresp\_Component(Individual\_definedIn\_DomainModel(ind))$

act4: $Property := Property \cup \{o\_lg\_p\}$

act5: $Invariant := Invariant \cup \{o\_lg\_i\}$

act6: $LogicFormula := LogicFormula \cup \{o\_lg\_p, o\_lg\_i\}$

act7: $LogicFormula\_uses\_Operators := LogicFormula\_uses\_Operators \cup (\{o\_lg\_p, o\_lg\_i\} \times \{\{1 \mapsto Belonging\_OP\}\})$

act8: $Constant\_isInvolvedIn\_LogicFormulas := Constant\_isInvolvedIn\_LogicFormulas \lhd (\{o\_ind \mapsto \{1 \mapsto o\_lg\_p, 1 \mapsto o\_lg\_i\}\} \cup (\{TRUE \mapsto \emptyset, FALSE \mapsto \{o\_PPCO\_c \mapsto Constant\_isInvolvedIn\_LogicFormulas(o\_PPCO\_c) \cup \{2 \mapsto o\_lg\_p\}\}\}(bool(PPCO \in dom(Concept\_corresp\_Set)))))$

act9: $Invariant\_involves\_Variables(o\_lg\_i) := \{2 \mapsto o\_CO\}$

act10: $LogicFormula\_involves\_Sets := LogicFormula\_involves\_Sets \cup \{o\_lg\_i \mapsto \emptyset\} \cup \{TRUE \mapsto \{o\_lg\_p \mapsto \{2 \mapsto o\_PPCO\_s\}\}, FALSE \mapsto \{o\_lg\_p \mapsto \emptyset\}\}(bool(PPCO \in dom(Concept\_corresp\_Set)))$

act11: $LogicFormula\_definedIn\_Component := LogicFormula\_definedIn\_Component \cup (\{o\_lg\_p, o\_lg\_i\} \times \{DomainModel\_corresp\_Component(Individual\_definedIn\_DomainModel(ind))\})$

act12: $Constant\_typing\_Property(o\_ind) := o\_lg\_p$

end

Event rule_13 ⟨convergent⟩ $\widehat{=}$

Variable individual of a variable concept

any ind o_ind CO o_lg o_CO
where

grd0: $(Individual \cap Individual\_isVariable^{-1}[\{TRUE\}]) \setminus (MapletIndividual \cup dom(Individual\_corresp\_Variable)) \neq \emptyset$

grd1: $ind \in (Individual \cap Individual\_isVariable^{-1}[\{TRUE\}]) \setminus (MapletIndividual \cup dom(Individual\_corresp\_Variable))$

grd2: $Individual\_individualOf\_Concept(ind) = CO$

grd3: $CO \in dom(Concept\_corresp\_Variable)$

grd4: $o\_CO = Concept\_corresp\_Variable(CO)$

grd5: $Individual\_definedIn\_DomainModel(ind) \in dom(DomainModel\_corresp\_Component)$

grd6: $Variable\_Set \setminus Variable \neq \emptyset$

grd7: $o\_ind \in Variable\_Set \setminus Variable$

grd8: $LogicFormula\_Set \setminus LogicFormula \neq \emptyset$

grd9: $o\_lg \in LogicFormula\_Set \setminus LogicFormula$

then

act1: $Variable := Variable \cup \{o\_ind\}$

act2: $Individual\_corresp\_Variable(ind) := o\_ind$

act3: $Variable\_definedIn\_Component(o\_ind) := DomainModel\_corresp\_Component(Individual\_definedIn\_DomainModel(ind))$

act4: $Invariant := Invariant \cup \{o\_lg\}$

act5: $LogicFormula := LogicFormula \cup \{o\_lg\}$

act6: $LogicFormula\_uses\_Operators(o\_lg) := \{1 \mapsto Belonging\_OP\}$

act7: $Invariant\_involves\_Variables(o\_lg) := \{1 \mapsto o\_ind, 2 \mapsto o\_CO\}$

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

act9: $LogicFormula\_definedIn\_Component(o\_lg) := DomainModel\_corresp\_Component(Individual\_definedIn\_DomainModel(ind))$

act10: $Variable\_typing\_Invariant(o\_ind) := o\_lg$

end

Event rule_14 ⟨convergent⟩ $\widehat{=}$

Variable individual of a concept that is an abstract enumeration

any ind o_ind CO o_lg o_CO
where

grd0: $(Individual \cap Individual\_isVariable^{-1}[\{TRUE\}]) \setminus (MapletIndividual \cup dom(Individual\_corresp\_Variable)) \neq \emptyset$

grd1: $ind \in (Individual \cap Individual\_isVariable^{-1}[\{TRUE\}]) \setminus (MapletIndividual \cup dom(Individual\_corresp\_Variable))$

grd2: $Individual\_individualOf\_Concept(ind) = CO$

grd3: $CO \in dom(Concept\_isEnumeration^{-1}[\{TRUE\}] \lhd Concept\_corresp\_Set) \setminus dom(Concept\_parentConcept\_Concept)$

grd4: $o\_CO = Concept\_corresp\_Set(CO)$

grd5: $Individual\_definedIn\_DomainModel(ind) \in dom(DomainModel\_corresp\_Component)$

grd6: $Variable\_Set \setminus Variable \neq \emptyset$

grd7: $o\_ind \in Variable\_Set \setminus Variable$

grd8: $LogicFormula\_Set \setminus LogicFormula \neq \emptyset$

grd9: $o\_lg \in LogicFormula\_Set \setminus LogicFormula$

then

act1: $Variable := Variable \cup \{o\_ind\}$

act2: $Individual\_corresp\_Variable(ind) := o\_ind$

act3: $Variable\_definedIn\_Component(o\_ind) := DomainModel\_corresp\_Component(Individual\_definedIn\_DomainModel(ind))$

act4: $Invariant := Invariant \cup \{o\_lg\}$

act5: $LogicFormula := LogicFormula \cup \{o\_lg\}$

act6: $LogicFormula\_uses\_Operators(o\_lg) := \{1 \mapsto Belonging\_OP\}$

act7: $Invariant\_involves\_Variables(o\_lg) := \{1 \mapsto o\_ind\}$

act8: $LogicFormula\_involves\_Sets(o\_lg) := \{2 \mapsto o\_CO\}$

act9: $LogicFormula\_definedIn\_Component(o\_lg) := DomainModel\_corresp\_Component(Individual\_definedIn\_DomainModel(ind))$

act10: $Variable\_typing\_Invariant(o\_ind) := o\_lg$

end

Event rule_15_0 ⟨convergent⟩ $\widehat{=}$

correspondence of an unnamed maplet individual

any ind AS o_AS_c o_AS_v ant o_ant_c o_ant_v im o_im_c o_im_v o_lg o_DM

where

grd0: $(MapletIndividual \cap Individual\_isNamed^{-1}[\{FALSE\}]) \setminus dom(UnnamedMapletIndividual\_corresp\_LogicalFormula) \neq \emptyset$

grd1: $ind \in (MapletIndividual \cap Individual\_isNamed^{-1}[\{FALSE\}]) \setminus dom(UnnamedMapletIndividual\_corresp\_LogicalFormula)$

grd2: $Association \cap (dom(Concept\_corresp\_Variable) \cup dom(Concept\_corresp\_Constant)) \neq \emptyset$

grd3: $AS \in Association \cap (dom(Concept\_corresp\_Variable) \cup dom(Concept\_corresp\_Constant))$

grd4: $Individual\_individualOf\_Concept(ind) = AS$

grd5: $o\_AS\_c \in Constant \land (AS \in dom(Concept\_corresp\_Constant) \Rightarrow o\_AS\_c = Concept\_corresp\_Constant(AS))$

grd6: $o\_AS\_v \in Variable \land (AS \in dom(Concept\_corresp\_Variable) \Rightarrow o\_AS\_v = Concept\_corresp\_Variable(AS))$

grd7: $Individual\_definedIn\_DomainModel(ind) \in dom(DomainModel\_corresp\_Component)$

grd8: $ant = MapletIndividual\_antecedent\_Individual(ind)$

grd9: $im = MapletIndividual\_image\_Individual(ind)$

grd10: $dom(Individual\_corresp\_Constant) \cup dom(Individual\_corresp\_Variable) \neq \emptyset$

grd11: $\{ant, im\} \subseteq dom(Individual\_corresp\_Constant) \cup dom(Individual\_corresp\_Variable)$

grd12: $o\_ant\_c \in Constant \land (ant \in dom(Individual\_corresp\_Constant)$
$\Rightarrow o\_ant\_c = Individual\_corresp\_Constant(ant))$

grd13: $o\_ant\_v \in Variable \land (ant \notin dom(Individual\_corresp\_Constant)$
$\Rightarrow o\_ant\_v = Individual\_corresp\_Variable(ant))$

grd14: $o\_im\_c \in Constant \land (im \in dom(Individual\_corresp\_Constant)$
$\Rightarrow o\_im\_c = Individual\_corresp\_Constant(im))$

grd15: $o\_im\_v \in Variable \land (im \notin dom(Individual\_corresp\_Constant)$
$\Rightarrow o\_im\_v = Individual\_corresp\_Variable(im))$

grd16: $LogicFormula\_Set \setminus LogicFormula \neq \emptyset$

grd17: $o\_lg \in LogicFormula\_Set \setminus LogicFormula$

grd18: $o\_DM = DomainModel\_corresp\_Component(Individual\_definedIn\_DomainModel(ind))$

grd19: $partition(\{o\_AS\_c, o\_ant\_c, o\_im\_c\}, \{o\_AS\_c\}, \{o\_ant\_c, o\_im\_c\})$

grd20: $partition(\{o\_AS\_v, o\_ant\_v, o\_im\_v\}, \{o\_AS\_v\}, \{o\_ant\_v, o\_im\_v\})$

then

act1: $Property := Property \cup (\{TRUE \mapsto \{o\_lg\}, FALSE \mapsto \emptyset\}(bool(ant \in dom(Individual\_corresp\_Constant) \land$
$im \in dom(Individual\_corresp\_Constant) \land AS \in dom(Concept\_corresp\_Constant))))$

act2: $Invariant := Invariant \cup (\{TRUE \mapsto \emptyset, FALSE \mapsto \{o\_lg\}\}(bool(ant \in dom(Individual\_corresp\_Constant) \land$
$im \in dom(Individual\_corresp\_Constant) \land AS \in dom(Concept\_corresp\_Constant))))$

act3: $LogicFormula := LogicFormula \cup \{o\_lg\}$

act4: $LogicFormula\_uses\_Operators(o\_lg) := \{1 \mapsto Maplet\_OP, 2 \mapsto Belonging\_OP\}$

act5: $Constant\_isInvolvedIn\_LogicFormulas := Constant\_isInvolvedIn\_LogicFormulas \Leftarrow ($
$(\{TRUE \mapsto \{o\_AS\_c \mapsto Constant\_isInvolvedIn\_LogicFormulas(o\_AS\_c) \cup \{3 \mapsto o\_lg\}\}, FALSE \mapsto \emptyset\}(bool(AS \in dom(Concept\_corresp\_Constant)))) \cup$
$(\{TRUE \mapsto (\{TRUE \mapsto (\{TRUE \mapsto \{o\_ant\_c \mapsto Constant\_isInvolvedIn\_LogicFormulas(o\_ant\_c) \cup (\{1, 2\} \times$
$\{o\_lg\})\}, \quad FALSE \mapsto \{o\_ant\_c \mapsto Constant\_isInvolvedIn\_LogicFormulas(o\_ant\_c) \cup \{1 \mapsto o\_lg\}, o\_im\_c \mapsto$
$Constant\_isInvolvedIn\_LogicFormulas(o\_im\_c) \cup \{2 \mapsto o\_lg\}\}\}(bool(o\_ant\_c = o\_im\_c))), \quad FALSE \mapsto \{o\_ant\_c \mapsto$
$Constant\_isInvolvedIn\_LogicFormulas(o\_ant\_c) \cup \{1 \mapsto o\_lg\}\}\}(bool(im \in dom(Individual\_corresp\_Constant)))), \quad FALSE \mapsto$
$(\{TRUE \mapsto \{o\_im\_c \mapsto Constant\_isInvolvedIn\_LogicFormulas(o\_im\_c) \cup \{2 \mapsto o\_lg\}\}, FALSE \mapsto \emptyset\}(bool(im \in$
$dom(Individual\_corresp\_Constant)))))\}(bool(ant \in dom(Individual\_corresp\_Constant)))) \quad)$

act6: $Invariant\_involves\_Variables := Invariant\_involves\_Variables$
$\cup$ ({$TRUE \mapsto \emptyset, FALSE \mapsto \{o\_lg \mapsto ($
$(\{TRUE \mapsto \emptyset, FALSE \mapsto \{1 \mapsto o\_ant\_v\}\})(bool(ant \in dom(Individual\_corresp\_Constant))))$
$\cup$ ({$TRUE \mapsto \emptyset, FALSE \mapsto \{2 \mapsto o\_im\_v\}\})(bool(im \in dom(Individual\_corresp\_Constant))))$
$\cup$ ({$TRUE \mapsto \emptyset, FALSE \mapsto \{3 \mapsto o\_AS\_v\}\})(bool(AS \in dom(Concept\_corresp\_Constant))))$
$)$ $\}\})(bool(ant \in dom(Individual\_corresp\_Constant)$
$\wedge\ im \in dom(Individual\_corresp\_Constant) \wedge AS \in dom(Concept\_corresp\_Constant))))$

act7: $LogicFormula\_involves\_Sets(o\_lg) := \emptyset$

act8: $LogicFormula\_definedIn\_Component(o\_lg) := o\_DM$

act9: $UnnamedMapletIndividual\_corresp\_LogicalFormula(ind) := o\_lg$

end

Event rule_15_1 ⟨convergent⟩ $\widehat{=}$

correspondence of a named variable maplet individual

any ind o_ind AS o_AS_c o_AS_v ant o_ant_c o_ant_v im o_im_c o_im_v o_lg_type o_lg_item o_DM

where

grd0: $(MapletIndividual \cap Individual\_isNamed^{-1}[\{TRUE\}]$
$\cap\ Individual\_isVariable^{-1}[\{TRUE\}]) \setminus dom(Individual\_corresp\_Variable) \neq \emptyset$

grd1: $ind \in (MapletIndividual \cap Individual\_isNamed^{-1}[\{TRUE\}]$
$\cap\ Individual\_isVariable^{-1}[\{TRUE\}]) \setminus dom(Individual\_corresp\_Variable)$

grd2: $Association \cap (dom(Concept\_corresp\_Variable) \cup dom(Concept\_corresp\_Constant)) \neq \emptyset$

grd3: $AS \in Association \cap (dom(Concept\_corresp\_Variable) \cup dom(Concept\_corresp\_Constant))$

grd4: $Individual\_individualOf\_Concept(ind) = AS$

grd5: $o\_AS\_c \in Constant \wedge (AS \in dom(Concept\_corresp\_Constant) \Rightarrow o\_AS\_c = Concept\_corresp\_Constant(AS))$

grd6: $o\_AS\_v \in Variable \wedge (AS \in dom(Concept\_corresp\_Variable) \Rightarrow o\_AS\_v = Concept\_corresp\_Variable(AS))$

grd7: $Individual\_definedIn\_DomainModel(ind) \in dom(DomainModel\_corresp\_Component)$

grd8: $Variable\_Set \setminus Variable \neq \emptyset$

grd9: $o\_ind \in Variable\_Set \setminus Variable$

grd10: $ant \in Individual \wedge (ind \in dom(MapletIndividual\_antecedent\_Individual)$
$\Rightarrow ant = MapletIndividual\_antecedent\_Individual(ind))$

grd11: $im \in Individual \wedge (ind \in dom(MapletIndividual\_antecedent\_Individual)$
$\Rightarrow im = MapletIndividual\_image\_Individual(ind))$

grd12: $ind \in dom(MapletIndividual\_antecedent\_Individual) \Rightarrow (dom(Individual\_corresp\_Constant) \cup dom(Individual\_corresp\_Variable) \neq \emptyset \wedge \{ant, im\} \subseteq dom(Individual\_corresp\_Constant) \cup dom(Individual\_corresp\_Variable))$

grd13: $o\_ant\_c \in Constant \wedge (ant \in dom(Individual\_corresp\_Constant)$
$\Rightarrow o\_ant\_c = Individual\_corresp\_Constant(ant))$

grd14: $o\_ant\_v \in Variable \land (ant \in dom(Individual\_corresp\_Variable) \Rightarrow o\_ant\_v = Individual\_corresp\_Variable(ant))$

grd15: $o\_im\_c \in Constant \land (im \in dom(Individual\_corresp\_Constant) \Rightarrow o\_im\_c = Individual\_corresp\_Constant(im))$

grd16: $o\_im\_v \in Variable \land (im \in dom(Individual\_corresp\_Variable) \Rightarrow o\_im\_v = Individual\_corresp\_Variable(im))$

grd17: $LogicFormula\_Set \setminus LogicFormula \neq \emptyset$

grd18: $\{o\_lg\_type, o\_lg\_item\} \subseteq LogicFormula\_Set \setminus LogicFormula$

grd19: $o\_DM = DomainModel\_corresp\_Component(Individual\_definedIn\_DomainModel(ind))$

grd20: $partition(\{o\_AS\_c, o\_ant\_c, o\_im\_c\}, \{o\_AS\_c\}, \{o\_ant\_c, o\_im\_c\})$

grd21: $partition(\{o\_ind, o\_AS\_v, o\_ant\_v, o\_im\_v\}, \{o\_ind\}, \{o\_AS\_v\}, \{o\_ant\_v, o\_im\_v\})$

grd22: $o\_lg\_type \neq o\_lg\_item$

then

act1: $Variable := Variable \cup \{o\_ind\}$

act2: $Individual\_corresp\_Variable(ind) := o\_ind$

act3: $Variable\_definedIn\_Component(o\_ind) := o\_DM$

act4: $Invariant := Invariant \cup \{o\_lg\_item\} \cup (\{TRUE \mapsto \{o\_lg\_type\}, FALSE \mapsto \emptyset\}(bool(ind \in dom(MapletIndividual\_antecedent\_Individual))))$

act5: $LogicFormula := LogicFormula \cup \{o\_lg\_item\} \cup (\{TRUE \mapsto \{o\_lg\_type\}, FALSE \mapsto \emptyset\}(bool(ind \in dom(MapletIndividual\_antecedent\_Individual))))$

act6: $LogicFormula\_uses\_Operators := LogicFormula\_uses\_Operators \cup \{o\_lg\_item \mapsto \{1 \mapsto Belonging\_OP\}\} \cup (\{TRUE \mapsto \{o\_lg\_type \mapsto \{1 \mapsto Equality\_OP, 2 \mapsto Maplet\_OP\}\}, FALSE \mapsto \emptyset\}(bool(ind \in dom(MapletIndividual\_antecedent\_Individual))))$

act7: $Constant\_isInvolvedIn\_LogicFormulas := Constant\_isInvolvedIn\_LogicFormulas \Leftarrow ((\{TRUE \mapsto \{o\_AS\_c \mapsto Constant\_isInvolvedIn\_LogicFormulas(o\_AS\_c) \cup \{2 \mapsto o\_lg\_item\}\}, FALSE \mapsto \emptyset\}(bool(AS \in dom(Concept\_corresp\_Constant)))) \cup (\{TRUE \mapsto (\{TRUE \mapsto (\{TRUE \mapsto (\{TRUE \mapsto \{o\_ant\_c \mapsto Constant\_isInvolvedIn\_LogicFormulas(o\_ant\_c) \cup (\{2, 3\} \times \{o\_lg\_type\})\}, FALSE \mapsto \{o\_ant\_c \mapsto Constant\_isInvolvedIn\_LogicFormulas(o\_ant\_c) \cup \{2 \mapsto o\_lg\_type\}, o\_im\_c \mapsto Constant\_isInvolvedIn\_LogicFormulas(o\_im\_c) \cup \{3 \mapsto o\_lg\_type\}\}\}(bool(o\_ant\_c = o\_im\_c))), FALSE \mapsto \{o\_ant\_c \mapsto Constant\_isInvolvedIn\_LogicFormulas(o\_ant\_c) \cup \{2 \mapsto o\_lg\_type\}\}\}(bool(im \in dom(Individual\_corresp\_Constant)))), FALSE \mapsto (\{TRUE \mapsto \{o\_im\_c \mapsto Constant\_isInvolvedIn\_LogicFormulas(o\_im\_c) \cup \{3 \mapsto o\_lg\_type\}\}, FALSE \mapsto \emptyset\}(bool(im \in dom(Individual\_corresp\_Constant)))))\}(bool(ant \in dom(Individual\_corresp\_Constant)))) , FALSE \mapsto \emptyset\}(bool(ind \in dom(MapletIndividual\_antecedent\_Individual)))))$

act8: $Invariant\_involves\_Variables := Invariant\_involves\_Variables \cup \{o\_lg\_item \mapsto (\{1 \mapsto o\_ind\} \cup (\{TRUE \mapsto \emptyset, FALSE \mapsto \{2 \mapsto o\_AS\_v\}\}(bool(AS \in dom(Concept\_corresp\_Constant))))) \} \cup (\{TRUE \mapsto \{o\_lg\_type \mapsto \{1 \mapsto o\_ind\} \cup (\{TRUE \mapsto \emptyset, FALSE \mapsto \{2 \mapsto o\_ant\_v\}\}(bool(ant \in dom(Individual\_corresp\_Constant)))) \cup (\{TRUE \mapsto \emptyset, FALSE \mapsto \{3 \mapsto o\_im\_v\}\}(bool(im \in dom(Individual\_corresp\_Constant))))) \}, FALSE \mapsto \emptyset\}(bool(ind \in dom(MapletIndividual\_antecedent\_Individual))))$

act9: $LogicFormula\_involves\_Sets := LogicFormula\_involves\_Sets \cup \{o\_lg\_item \mapsto \emptyset\}$
$\cup (\{TRUE \mapsto \{o\_lg\_type \mapsto \emptyset\}, FALSE \mapsto \emptyset\}(bool(ind \in dom(MapletIndividual\_antecedent\_Individual))))$

act10: $LogicFormula\_definedIn\_Component := LogicFormula\_definedIn\_Component \cup \{o\_lg\_item \mapsto o\_DM\} \cup$
$(\{TRUE \mapsto \{o\_lg\_type \mapsto o\_DM\}, FALSE \mapsto \emptyset\}(bool(ind \in dom(MapletIndividual\_antecedent\_Individual))))$

act11: $Variable\_typing\_Invariant(o\_ind) := o\_lg\_item$

end

Event rule_15_2 ⟨convergent⟩ $\widehat{=}$

correspondence of a named constant maplet individual (constant association)

any ind o_ind AS o_AS ant o_ant im o_im o_lg_type o_lg_item o_DM

where

grd0: $(MapletIndividual \cap Individual\_isNamed^{-1}[\{TRUE\}]$
$\cap Individual\_isVariable^{-1}[\{FALSE\}]) \setminus dom(Individual\_corresp\_Constant) \neq \emptyset$

grd1: $ind \in (MapletIndividual \cap Individual\_isNamed^{-1}[\{TRUE\}]$
$\cap Individual\_isVariable^{-1}[\{FALSE\}]) \setminus dom(Individual\_corresp\_Constant)$

grd2: $Association \cap dom(Concept\_corresp\_Constant) \neq \emptyset$

grd3: $AS \in Association \cap dom(Concept\_corresp\_Constant)$

grd4: $Individual\_individualOf\_Concept(ind) = AS$

grd5: $o\_AS = Concept\_corresp\_Constant(AS)$

grd6: $Individual\_definedIn\_DomainModel(ind) \in dom(DomainModel\_corresp\_Component)$

grd7: $Constant\_Set \setminus Constant \neq \emptyset$

grd8: $o\_ind \in Constant\_Set \setminus Constant$

grd9: $ant \in Individual \wedge (ind \in dom(MapletIndividual\_antecedent\_Individual)$
$\Rightarrow ant = MapletIndividual\_antecedent\_Individual(ind))$

grd10: $im \in Individual \wedge (ind \in dom(MapletIndividual\_antecedent\_Individual)$
$\Rightarrow im = MapletIndividual\_image\_Individual(ind))$

grd11: $ind \in dom(MapletIndividual\_antecedent\_Individual)$
$\Rightarrow (dom(Individual\_corresp\_Constant) \neq \emptyset \wedge \{ant, im\} \subseteq dom(Individual\_corresp\_Constant))$

grd12: $o\_ant \in Constant \wedge (ind \in dom(MapletIndividual\_antecedent\_Individual)$
$\Rightarrow o\_ant = Individual\_corresp\_Constant(ant))$

grd13: $o\_im \in Constant \wedge (ind \in dom(MapletIndividual\_antecedent\_Individual)$
$\Rightarrow o\_im = Individual\_corresp\_Constant(im))$

grd14: $LogicFormula\_Set \setminus LogicFormula \neq \emptyset$

grd15: $\{o\_lg\_type, o\_lg\_item\} \subseteq LogicFormula\_Set \setminus LogicFormula$

grd16: $o\_DM = DomainModel\_corresp\_Component(Individual\_definedIn\_DomainModel(ind))$

grd17: $partition(\{o\_ind, o\_AS, o\_ant, o\_im\}, \{o\_ind\}, \{o\_AS\}, \{o\_ant, o\_im\})$

grd18: $o\_lg\_type \neq o\_lg\_item$

then

act1: $Constant := Constant \cup \{o\_ind\}$

act2: $Individual\_corresp\_Constant(ind) := o\_ind$

act3: $Constant\_definedIn\_Component(o\_ind) := o\_DM$

act4: $Property := Property \cup \{o\_lg\_item\} \cup (\{TRUE \mapsto \{o\_lg\_type\}, FALSE \mapsto \emptyset\}(bool(ind \in dom(MapletIndividual\_antecedent\_Individual))))$

act5: $LogicFormula := LogicFormula \cup \{o\_lg\_item\} \cup (\{TRUE \mapsto \{o\_lg\_type\}, FALSE \mapsto \emptyset\}(bool(ind \in dom(MapletIndividual\_antecedent\_Individual))))$

act6: $LogicFormula\_uses\_Operators := LogicFormula\_uses\_Operators \cup \{o\_lg\_item \mapsto \{1 \mapsto Belonging\_OP\}\} \cup (\{TRUE \mapsto \{o\_lg\_type \mapsto \{1 \mapsto Equality\_OP, 2 \mapsto Maplet\_OP\}\}, FALSE \mapsto \emptyset\}(bool(ind \in dom(MapletIndividual\_antecedent\_Individual))))$

act7: $Constant\_isInvolvedIn\_LogicFormulas := Constant\_isInvolvedIn\_LogicFormulas \Leftarrow (\{o\_ind \mapsto (\{1 \mapsto o\_lg\_item\} \cup (\{TRUE \mapsto \{1 \mapsto o\_lg\_type\}, FALSE \mapsto \emptyset\}(bool(ind \in dom(MapletIndividual\_antecedent\_Individual))))))\} \cup \{o\_AS \mapsto Constant\_isInvolvedIn\_LogicFormulas(o\_AS) \cup \{2 \mapsto o\_lg\_item\}\} \cup (\{TRUE \mapsto (\{TRUE \mapsto \{o\_ant \mapsto Constant\_isInvolvedIn\_LogicFormulas(o\_ant) \cup (\{2,3\} \times \{o\_lg\_type\})\}, FALSE \mapsto \{o\_ant \mapsto Constant\_isInvolvedIn\_LogicFormulas(o\_ant) \cup \{2 \mapsto o\_lg\_type\}, o\_im \mapsto Constant\_isInvolvedIn\_LogicFormulas(o\_im) \cup \{3 \mapsto o\_lg\_type\}\}\}(bool(o\_ant = o\_im))), FALSE \mapsto \emptyset\}(bool(ind \in dom(MapletIndividual\_antecedent\_Individual))))$

act8: $LogicFormula\_involves\_Sets := LogicFormula\_involves\_Sets \cup \{o\_lg\_item \mapsto \emptyset\} \cup (\{TRUE \mapsto \{o\_lg\_type \mapsto \emptyset\}, FALSE \mapsto \emptyset\}(bool(ind \in dom(MapletIndividual\_antecedent\_Individual))))$

act9: $LogicFormula\_definedIn\_Component := LogicFormula\_definedIn\_Component \cup \{o\_lg\_item \mapsto o\_DM\} \cup (\{TRUE \mapsto \{o\_lg\_type \mapsto o\_DM\}, FALSE \mapsto \emptyset\}(bool(ind \in dom(MapletIndividual\_antecedent\_Individual))))$

act10: $Constant\_typing\_Property(o\_ind) := o\_lg\_item$

end

Event rule_15_3 ⟨convergent⟩ $\widehat{=}$

correspondence of a named constant maplet individual (variable association)

any ind o_ind AS o_AS ant o_ant im o_im PPCO1 o_PPCO1_c o_PPCO1_s PPCO2 o_PPCO2_c o_PPCO2_s o_lg_type o_lg_item_i o_lg_item_p o_DM

where

grd0: $(MapletIndividual \cap Individual\_isNamed^{-1}[\{TRUE\}] \cap Individual\_isVariable^{-1}[\{FALSE\}]) \setminus dom(Individual\_corresp\_Constant) \neq \emptyset$

grd1: $ind \in (MapletIndividual \cap Individual\_isNamed^{-1}[\{TRUE\}] \cap Individual\_isVariable^{-1}[\{FALSE\}]) \setminus dom(Individual\_corresp\_Constant)$

grd2: $Association \cap dom(Concept\_corresp\_Variable) \neq \emptyset$

grd3: $AS \in Association \cap dom(Concept\_corresp\_Variable)$

grd4: $Individual\_individualOf\_Concept(ind) = AS$

grd5: $o\_AS = Concept\_corresp\_Variable(AS)$

grd6: $Individual\_definedIn\_DomainModel(ind) \in dom(DomainModel\_corresp\_Component)$

grd7: $Constant\_Set \setminus Constant \neq \emptyset$

grd8: $o\_ind \in Constant\_Set \setminus Constant$

grd9: $ant \in Individual \wedge (ind \in dom(MapletIndividual\_antecedent\_Individual)$
$\Rightarrow ant = MapletIndividual\_antecedent\_Individual(ind))$

grd10: $im \in Individual \wedge (ind \in dom(MapletIndividual\_antecedent\_Individual)$
$\Rightarrow im = MapletIndividual\_image\_Individual(ind))$

grd11: $ind \in dom(MapletIndividual\_antecedent\_Individual)$
$\Rightarrow (dom(Individual\_corresp\_Constant) \neq \emptyset \wedge \{ant, im\} \subseteq dom(Individual\_corresp\_Constant))$

grd12: $o\_ant \in Constant \wedge (ind \in dom(MapletIndividual\_antecedent\_Individual)$
$\Rightarrow o\_ant = Individual\_corresp\_Constant(ant))$

grd13: $o\_im \in Constant \wedge (ind \in dom(MapletIndividual\_antecedent\_Individual)$
$\Rightarrow o\_im = Individual\_corresp\_Constant(im))$

grd14: $LogicFormula\_Set \setminus LogicFormula \neq \emptyset$

grd15: $\{o\_lg\_type, o\_lg\_item\_i, o\_lg\_item\_p\} \subseteq LogicFormula\_Set \setminus LogicFormula$

grd16: $o\_DM = DomainModel\_corresp\_Component(Individual\_definedIn\_DomainModel(ind))$

grd17: $\{PPCO1, PPCO2\} \subseteq (dom(Concept\_corresp\_Set)$
$\cup\, dom(Concept\_corresp\_Constant)) \cap ran(Concept\_parentConcept\_Concept)$

grd18: $PPCO1 \in cls(Concept\_parentConcept\_Concept)[\{Association\_domain\_Concept(AS)\}]$

grd19: $PPCO2 \in cls(Concept\_parentConcept\_Concept)[\{Association\_range\_Concept(AS)\}]$

grd20: $o\_PPCO1\_s \in Set \wedge (PPCO1 \in dom(Concept\_corresp\_Set) \Rightarrow o\_PPCO1\_s = Concept\_corresp\_Set(PPCO1))$

grd21: $o\_PPCO1\_c \in Constant \wedge (PPCO1 \in dom(Concept\_corresp\_Constant)$
$\Rightarrow o\_PPCO1\_c = Concept\_corresp\_Constant(PPCO1))$

grd22: $o\_PPCO2\_s \in Set \wedge (PPCO2 \in dom(Concept\_corresp\_Set) \Rightarrow o\_PPCO2\_s = Concept\_corresp\_Set(PPCO2))$

grd23: $o\_PPCO2\_c \in Constant \wedge (PPCO2 \in dom(Concept\_corresp\_Constant)$
$\Rightarrow o\_PPCO2\_c = Concept\_corresp\_Constant(PPCO2))$

grd24: $partition(\{o\_ind, o\_ant, o\_im, o\_PPCO1\_c, o\_PPCO2\_c\}, \{o\_ind\}, \{o\_ant, o\_im\}, \{o\_PPCO1\_c, o\_PPCO2\_c\})$

grd25: $partition(\{o\_lg\_type, o\_lg\_item\_i, o\_lg\_item\_p\}, \{o\_lg\_type\}, \{o\_lg\_item\_i\}, \{o\_lg\_item\_p\})$

then

act1: $Constant := Constant \cup \{o\_ind\}$

act2: $Individual\_corresp\_Constant(ind) := o\_ind$

act3: $Constant\_definedIn\_Component(o\_ind) := o\_DM$

act4: $Property := Property \cup \{o\_lg\_item\_p\} \cup (\{TRUE \mapsto \{o\_lg\_type\},$
$FALSE \mapsto \emptyset\}(bool(ind \in dom(MapletIndividual\_antecedent\_Individual))))$

act5: $Invariant := Invariant \cup \{o\_lg\_item\_i\}$

act6: $LogicFormula := LogicFormula \cup \{o\_lg\_item\_p, o\_lg\_item\_i\} \cup (\{TRUE \mapsto \{o\_lg\_type\}, FALSE \mapsto \emptyset\}(bool(ind \in dom(MapletIndividual\_antecedent\_Individual))))$

act7: $LogicFormula\_uses\_Operators := LogicFormula\_uses\_Operators \cup \{o\_lg\_item\_i \mapsto \{1 \mapsto Belonging\_OP\}, o\_lg\_item\_p \mapsto \{1 \mapsto Belonging\_OP, 2 \mapsto RelationSet\_OP\}\} \cup (\{TRUE \mapsto \{o\_lg\_type \mapsto \{1 \mapsto Equality\_OP, 2 \mapsto Maplet\_OP\}\}, FALSE \mapsto \emptyset\}(bool(ind \in dom(MapletIndividual\_antecedent\_Individual))))$

act8: $Constant\_isInvolvedIn\_LogicFormulas := Constant\_isInvolvedIn\_LogicFormulas \Leftarrow (\{o\_ind \mapsto ((\{1\} \times \{o\_lg\_item\_p, o\_lg\_item\_i\}) \cup (\{TRUE \mapsto \{1 \mapsto o\_lg\_type\}, FALSE \mapsto \emptyset\}(bool(ind \in dom(MapletIndividual\_antecedent\_Individual))))))\} \cup (\{TRUE \mapsto (\{TRUE \mapsto (\{TRUE \mapsto \{o\_PPCO1\_c \mapsto Constant\_isInvolvedIn\_LogicFormulas(o\_PPCO1\_c) \cup (\{2,3\} \times \{o\_lg\_item\_p\})\}, FALSE \mapsto \{o\_PPCO1\_c \mapsto Constant\_isInvolvedIn\_LogicFormulas(o\_PPCO1\_c) \cup \{2 \mapsto o\_lg\_item\_p\}, o\_PPCO2\_c \mapsto Constant\_isInvolvedIn\_LogicFormulas(o\_PPCO2\_c) \cup \{3 \mapsto o\_lg\_item\_p\}\}\}(bool(o\_PPCO1\_c = o\_PPCO2\_c))), FALSE \mapsto \{o\_PPCO2\_c \mapsto Constant\_isInvolvedIn\_LogicFormulas(o\_PPCO2\_c) \cup \{2 \mapsto o\_lg\_item\_p\}\}\}(bool(PPCO2 \in dom(Concept\_corresp\_Constant)))), FALSE \mapsto (\{TRUE \mapsto \{o\_PPCO2\_c \mapsto Constant\_isInvolvedIn\_LogicFormulas(o\_PPCO2\_c) \cup \{3 \mapsto o\_lg\_item\_p\}\}, FALSE \mapsto \emptyset\}(bool(PPCO2 \in dom(Concept\_corresp\_Constant))))\}(bool(PPCO1 \in dom(Concept\_corresp\_Constant)))) \cup (\{TRUE \mapsto (\{TRUE \mapsto \{o\_ant \mapsto Constant\_isInvolvedIn\_LogicFormulas(o\_ant) \cup (\{2,3\} \times \{o\_lg\_type\})\}, FALSE \mapsto \{o\_ant \mapsto Constant\_isInvolvedIn\_LogicFormulas(o\_ant) \cup \{2 \mapsto o\_lg\_type\}, o\_im \mapsto Constant\_isInvolvedIn\_LogicFormulas(o\_im) \cup \{3 \mapsto o\_lg\_type\}\}\}(bool(o\_ant = o\_im))), FALSE \mapsto \emptyset\}(bool(ind \in dom(MapletIndividual\_antecedent\_Individual)))))$

act9: $Invariant\_involves\_Variables(o\_lg\_item\_i) := \{2 \mapsto o\_AS\}$

act10: $LogicFormula\_involves\_Sets := LogicFormula\_involves\_Sets \cup \{o\_lg\_item\_i \mapsto \emptyset\} \cup \{o\_lg\_item\_p \mapsto (\{TRUE \mapsto \emptyset, FALSE \mapsto \{2 \mapsto o\_PPCO1\_s\}\}(bool(PPCO1 \in dom(Concept\_corresp\_Constant))) \cup \{TRUE \mapsto \emptyset, FALSE \mapsto \{3 \mapsto o\_PPCO2\_s\}\}(bool(PPCO2 \in dom(Concept\_corresp\_Constant))))\} \cup (\{TRUE \mapsto \{o\_lg\_type \mapsto \emptyset\}, FALSE \mapsto \emptyset\}(bool(ind \in dom(MapletIndividual\_antecedent\_Individual))))$

act11: $LogicFormula\_definedIn\_Component := LogicFormula\_definedIn\_Component \cup (\{o\_lg\_item\_p, o\_lg\_item\_i\} \times \{o\_DM\}) \cup (\{TRUE \mapsto \{o\_lg\_type \mapsto o\_DM\}, FALSE \mapsto \emptyset\}(bool(ind \in dom(MapletIndividual\_antecedent\_Individual))))$

act12: $Constant\_typing\_Property(o\_ind) := o\_lg\_item\_p$

end

Event rule_18 ⟨convergent⟩ $\widehat{=}$

handling the transitivity of an association

any AS o_AS_c o_AS_v o_lg

where

grd0: $(Association \cap (dom(Concept\_corresp\_Constant) \cup dom(Concept\_corresp\_Variable)) \cap Association\_isTransitive^{-1}[\{TRUE\}]) \neq \emptyset$

grd1: $AS \in (Association \cap (dom(Concept\_corresp\_Constant) \cup dom(Concept\_corresp\_Variable)) \cap Association\_isTransitive^{-1}[\{TRUE\}])$

grd2: $(\{AS \mapsto isTransitive\}) \notin dom(AssociationCharacteristic\_corresp\_LogicFormula)$

grd3: $o\_AS\_c \in Constant \land (AS \in dom(Concept\_corresp\_Constant) \Rightarrow o\_AS\_c = Concept\_corresp\_Constant(AS))$

grd4: $o\_AS\_v \in Variable \land (AS \in dom(Concept\_corresp\_Variable) \Rightarrow o\_AS\_v = Concept\_corresp\_Variable(AS))$

grd5: $Concept\_definedIn\_DomainModel(AS) \in dom(DomainModel\_corresp\_Component)$

grd6: $LogicFormula\_Set \setminus LogicFormula \neq \emptyset$

grd7: $o\_lg \in LogicFormula\_Set \setminus LogicFormula$

then

act1: $Property := Property \cup (\{TRUE \mapsto \{o\_lg\}, FALSE \mapsto \emptyset\}(bool(AS \in dom(Concept\_corresp\_Constant))))$

act2: $Invariant := Invariant \cup (\{TRUE \mapsto \emptyset, FALSE \mapsto \{o\_lg\}\}(bool(AS \in dom(Concept\_corresp\_Constant))))$

act3: $LogicFormula := LogicFormula \cup \{o\_lg\}$

act4: $AssociationCharacteristic\_corresp\_LogicFormula(\{AS \mapsto isTransitive\}) := o\_lg$

act5: $Constant\_isInvolvedIn\_LogicFormulas := Constant\_isInvolvedIn\_LogicFormulas \Leftarrow (\{TRUE \mapsto \{o\_AS\_c \mapsto Constant\_isInvolvedIn\_LogicFormulas(o\_AS\_c) \cup (\{1,2,3\} \times \{o\_lg\})\}, FALSE \mapsto \emptyset\}(bool(AS \in dom(Concept\_corresp\_Constant))))$

act6: $Invariant\_involves\_Variables := Invariant\_involves\_Variables \cup (\{TRUE \mapsto \emptyset, FALSE \mapsto \{o\_lg \mapsto (\{1,2,3\} \times \{o\_AS\_v\})\}\}(bool(AS \in dom(Concept\_corresp\_Constant))))$

act7: $LogicFormula\_uses\_Operators(o\_lg) := \{1 \mapsto RelationComposition\_OP, 2 \mapsto Inclusion\_OP\}$

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

act9: $LogicFormula\_definedIn\_Component(o\_lg) := DomainModel\_corresp\_Component(Concept\_definedIn\_DomainModel(AS))$

end

END

MACHINE event_b_specs_from_ontologies_ref_2
REFINES event_b_specs_from_ontologies_ref_1
SEES EventB_Metamodel_Context,Domain_Metamodel_Context
EVENTS
Event rule_17 ⟨convergent⟩ $\widehat{=}$

variable concept initialisation

any CO o_CO o_ia inds o_inds bij_o_inds

where

grd0: $ran(Concept\_corresp\_Variable) \setminus ran(InitialisationAction\_inits\_Variable) \neq \emptyset$

grd1: $o\_CO \in ran(Concept\_corresp\_Variable) \setminus ran(InitialisationAction\_inits\_Variable)$

grd2: $CO = Concept\_corresp\_Variable^{-1}(o\_CO)$

grd3: $inds = Individual\_individualOf\_Concept^{-1}[\{CO\}] \cap Individual\_isVariable^{-1}[\{FALSE\}]$

grd4: $inds \subseteq dom(Individual\_corresp\_Constant)$

grd5: $o\_inds = Individual\_corresp\_Constant[inds]$

grd6: $InitialisationAction\_Set \setminus InitialisationAction \neq \emptyset$

grd7: $o\_ia \in InitialisationAction\_Set \setminus InitialisationAction$

grd8: $card(o\_inds) = card(inds)$

grd9: $bij\_o\_inds \in 1 \,..\, card(o\_inds) \rightarrowtail\!\!\!\!\!\twoheadrightarrow o\_inds$

then

act1: $InitialisationAction := InitialisationAction \cup \{o\_ia\}$

act2: $InitialisationAction\_uses\_Operators(o\_ia) := \{1 \mapsto BecomeEqual2SetOf\_OP\}$

act3: $InitialisationAction\_inits\_Variable(o\_ia) := o\_CO$

act4: $InitialisationAction\_involves\_Constants(o\_ia) := bij\_o\_inds$

end

END

B.4 Definition of the Adjusted Back Propagation Rules

B.4.1 Informal Definition

Table B.3 presents the revised 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 B.3 – The revised back propagation rules

		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_1	**Constant typed as subset of the correspondent of a concept**	b_CO b_PCO	$b\_CO \in$ Constant $b\_PCO \in$ AbstractSet $\cup$ Constant $b\_CO \subseteq b\_PCO$ $o\_PCO \in$ Concept	o_CO	$o\_CO \in$ Concept Concept_parent_Concept(o_CO) = o_PCO this rule does not consider constant concepts with variable parents (see rule 4_2).

4_2	**Constant typed as subset of the correspondent of a variable concept**	b_CO b_PCO o_CO	$b\_CO \in$ Constant $b\_PCO \in$ Variable $b\_CO \subseteq b\_PCO$ $\{o\_CO, o\_PCO\} \subseteq$ 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 $\cup$ Constant $b\_elt \in b\_CO$ $o\_CO \in$ Concept	o_elt	$o\_elt \in$ Individual Individual_individualOf_Concept(o_elt) = o_CO this rule does not consider constant individuals of variable concepts. Another rule similar to rule 4_2 can be defined to handle them.
6	**Variable typed as subset of the correspondent of a concept**	b_CO b_PCO	$b\_CO \in$ Variable $b\_PCO \in$ AbstractSet $\cup$ Constant $\cup$ 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$\cup$Constant$\cup$ 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 $\cup$ 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 it appears.

B.4.2 Event-B Specification

MACHINE event_b_specs_from_ontologies_ref_1
REFINES event_b_specs_from_ontologies
SEES EventB_Metamodel_Context,Domain_Metamodel_Context
EVENTS
Event rule_b_1 ⟨convergent⟩ $\widehat{=}$

Back propagating the addition of abstract sets

any b_CO o_CO
where

grd0: $AbstractSet \setminus ran(Concept\_corresp\_Set) \neq \emptyset$

grd1: $b\_CO \in AbstractSet \setminus ran(Concept\_corresp\_Set)$

grd2: $Set\_definedIn\_Component(b\_CO) \in ran(DomainModel\_corresp\_Component)$

grd3: $Concept\_Set \setminus Concept \neq \emptyset$

grd4: $o\_CO \in Concept\_Set \setminus Concept$

then

act1: $Concept := Concept \cup \{o\_CO\}$

act2: $Concept\_corresp\_Set(o\_CO) := b\_CO$

act3: $Concept\_isVariable(o\_CO) := FALSE$

act4: $Concept\_isEnumeration(o\_CO) := FALSE$

act5: $Concept\_definedIn\_DomainModel(o\_CO) := DomainModel\_corresp\_Component^{-1}(Set\_definedIn\_Component(b\_CO))$

end

Event rule_b_2 ⟨convergent⟩ $\widehat{=}$

Back propagating the addition of an enumerated set

any b_EDS o_EDS b_elements o_elements mapping_b_elements_o_elements
where

grd0: $EnumeratedSet \setminus ran(Concept\_corresp\_Set) \neq \emptyset$

grd1: $b\_EDS \in EnumeratedSet \setminus ran(Concept\_corresp\_Set)$

grd2: $Set\_definedIn\_Component(b\_EDS) \in ran(DomainModel\_corresp\_Component)$

grd3: $Concept\_Set \setminus Concept \neq \emptyset$

grd4: $o\_EDS \in Concept\_Set \setminus Concept$

grd5: $Individual\_Set \setminus Individual \neq \emptyset$

grd6: $o\_elements \subseteq Individual\_Set \setminus Individual$

grd7: $b\_elements = SetItem\_itemOf\_EnumeratedSet^{-1}[\{b\_EDS\}]$

grd8: $card(o\_elements) = card(b\_elements)$

grd9: $mapping\_b\_elements\_o\_elements \in o\_elements \rightarrowtail\!\!\!\!\!\rightarrow b\_elements$

grd10: $ran(Individual\_corresp\_SetItem) \cap b\_elements = \emptyset$

then

act1: $Concept := Concept \cup \{o\_EDS\}$

act2: $Concept\_corresp\_Set(o\_EDS) := b\_EDS$

act3: $Concept\_isVariable(o\_EDS) := FALSE$

act4: $Concept\_isEnumeration(o\_EDS) := TRUE$

act5: $Concept\_definedIn\_DomainModel(o\_EDS) := DomainModel\_corresp\_Component^{-1}(Set\_definedIn\_Component(b\_EDS))$

act6: $Individual := Individual \cup o\_elements$

act7: $Individual\_isVariable := Individual\_isVariable \cup (o\_elements \times \{FALSE\})$

act8: $Individual\_isNamed := Individual\_isNamed \cup (o\_elements \times \{TRUE\})$

act9: $Individual\_individualOf\_Concept := Individual\_individualOf\_Concept \cup (o\_elements \times \{o\_EDS\})$

act10: $Individual\_corresp\_SetItem := Individual\_corresp\_SetItem \cup mapping\_b\_elements\_o\_elements$

act11: $Individual\_definedIn\_DomainModel := Individual\_definedIn\_DomainModel$
$\cup (o\_elements \times \{DomainModel\_corresp\_Component^{-1}(Set\_definedIn\_Component(b\_EDS))\})$

end

Event rule_b_3 ⟨convergent⟩ $\widehat{=}$

Back propagating the addition of a new element in an existing enumerated set

any EDS b_EDS o_element b_element

where

grd0: $dom(SetItem\_itemOf\_EnumeratedSet) \setminus ran(Individual\_corresp\_SetItem) \neq \emptyset$

grd1: $b\_element \in dom(SetItem\_itemOf\_EnumeratedSet) \setminus ran(Individual\_corresp\_SetItem)$

grd2: $b\_EDS = SetItem\_itemOf\_EnumeratedSet(b\_element)$

grd3: $ran(Concept\_corresp\_Set) \neq \emptyset \wedge b\_EDS \in ran(Concept\_corresp\_Set)$

grd4: $EDS = Concept\_corresp\_Set^{-1}(b\_EDS)$

grd5: $Individual\_Set \setminus Individual \neq \emptyset$

grd6: $o\_element \in Individual\_Set \setminus Individual$

grd7: $Set\_definedIn\_Component(b\_EDS) \in ran(DomainModel\_corresp\_Component)$

then

act1: $Individual := Individual \cup \{o\_element\}$

act2: $Individual\_individualOf\_Concept(o\_element) := EDS$

act3: $Individual\_corresp\_SetItem(o\_element) := b\_element$

act4: $Individual\_isVariable(o\_element) := FALSE$

act5: $Individual\_isNamed(o\_element) := TRUE$

act6: $Individual\_definedIn\_DomainModel(o\_element) := DomainModel\_corresp\_Component^{-1}(Set\_definedIn\_Component(b\_EDS))$

end

Event rule_b_4 ⟨convergent⟩ $\widehat{=}$

(rule_b_4_1 & rule_b_4_2) Back propagating the addition of a constant typed as subset of the correspondent of a concept

any b_CO o_CO b_PCO_s b_PCO_c b_PCO_v b_lg PCO olges olgis b_inds o_inds b_inds_map_o_inds

where

grd0: $dom(Constant\_typing\_Property) \setminus (ran(Concept\_corresp\_Constant) \cup ran(Individual\_corresp\_Constant) \cup ran(Association\_Type\_Constant)) \neq \emptyset$

grd1: $b\_CO \in dom(Constant\_typing\_Property) \setminus (ran(Concept\_corresp\_Constant) \cup ran(Individual\_corresp\_Constant) \cup ran(Association\_Type\_Constant))$

grd2: $b\_lg = Constant\_typing\_Property(b\_CO)$

grd3: $LogicFormula\_uses\_Operators(b\_lg) = \{1 \mapsto Inclusion\_OP\}$

grd4: $Concept \neq \emptyset \land PCO \in Concept$

grd5: $Constant \neq \emptyset \land b\_PCO\_c \in Constant$

grd6: $LogicFormula\_involves\_Sets(b\_lg) \neq \emptyset \Rightarrow (ran(Concept\_corresp\_Set) \neq \emptyset \land b\_PCO\_s \in ran(Concept\_corresp\_Set) \land (2 \mapsto b\_PCO\_s) \in LogicFormula\_involves\_Sets(b\_lg) \land PCO = Concept\_corresp\_Set^{-1}(b\_PCO\_s) \land \forall co \cdot (co \in Constant \Rightarrow (2 \mapsto b\_lg) \notin Constant\_isInvolvedIn\_LogicFormulas(co)))$

grd7: $LogicFormula\_involves\_Sets(b\_lg) = \emptyset \Rightarrow (ran(Concept\_corresp\_Constant) \neq \emptyset \land b\_PCO\_c \in ran(Concept\_corresp\_Constant) \land (2 \mapsto b\_lg) \in Constant\_isInvolvedIn\_LogicFormulas(b\_PCO\_c) \land PCO = Concept\_corresp\_Constant^{-1}(b\_PCO\_c) \land \forall co \cdot (co \in Constant \setminus \{b\_PCO\_c\} \Rightarrow (2 \mapsto b\_lg) \notin Constant\_isInvolvedIn\_LogicFormulas(co)))$

grd8: $Concept\_Set \setminus Concept \neq \emptyset \land o\_CO \in Concept\_Set \setminus Concept$

grd9: $Constant\_definedIn\_Component(b\_CO) \in ran(DomainModel\_corresp\_Component)$

grd10: $b\_inds \subseteq dom(Constant\_typing\_Property) \setminus (ran(Concept\_corresp\_Constant) \cup ran(Individual\_corresp\_Constant) \cup ran(Association\_Type\_Constant) \cup \{b\_CO\})$

grd11: $o\_inds \subseteq Individual\_Set \setminus Individual$

grd12: $b\_inds\_map\_o\_inds \in Individual\_Set \rightarrowtail\!\!\!\!\!\!\rightarrow Constant$

grd13:

$olges = \{o\_lge \cdot ($

$o\_lge \in LogicFormula \setminus (ran(AssociationCharacteristic\_corresp\_LogicFormula) \cup ran(ConcreteEnumeration\_corresp\_-$

$IndividualSetLogicalFormula) \cup ran(UnnamedMapletIndividual\_corresp\_LogicalFormula))$

$\wedge\ LogicFormula\_uses\_Operators(o\_lge) = \{1 \mapsto Equal2SetOf\_OP\}$

$\wedge\ (1 \mapsto o\_lge) \in Constant\_isInvolvedIn\_LogicFormulas(b\_CO)$

$\wedge\ b\_inds \neq \emptyset \wedge card(o\_inds) = card(b\_inds) \wedge b\_inds\_map\_o\_inds \in o\_inds \rightarrowtail\!\!\!\!\!\!\rightarrow b\_inds$

$\wedge\ b\_inds \cap (ran(Concept\_corresp\_Constant) \cup ran(Individual\_corresp\_Constant)$

$\cup\ ran(Association\_Type\_Constant)) = \emptyset\ \wedge (\exists bij\_b\_inds \cdot ($

$bij\_b\_inds \in b\_inds \rightarrowtail\!\!\!\!\!\!\rightarrow 2\,..\,(card(b\_inds) + 1)\ \wedge \forall b\_ind \cdot ($

$b\_ind \in b\_inds \Rightarrow (bij\_b\_inds(b\_ind) \mapsto o\_lge \in Constant\_isInvolvedIn\_LogicFormulas(b\_ind)))))$

$)|o\_lge\}$

grd14: $card(olges) \leq 1$

grd16: $Variable \neq \emptyset \wedge b\_PCO\_v \in Variable$

grd17: $olgis = \{o\_lgi \cdot (o\_lgi \in Invariant \wedge LogicFormula\_uses\_Operators(o\_lgi) = \{1 \mapsto Inclusion\_OP\}$

$\wedge\ (1 \mapsto o\_lgi) \in Constant\_isInvolvedIn\_LogicFormulas(b\_CO)$

$\wedge\ (2 \mapsto b\_PCO\_v) \in Invariant\_involves\_Variables(o\_lgi)) | o\_lgi\}$

grd18: $olgis \neq \emptyset \Rightarrow (ran(Concept\_corresp\_Variable) \neq \emptyset \wedge b\_PCO\_v \in ran(Concept\_corresp\_Variable)$

$\wedge\ PCO = Concept\_corresp\_Variable^{-1}(b\_PCO\_v))$

then

act1: $Concept := Concept \cup \{o\_CO\}$

act2: $Concept\_corresp\_Constant(o\_CO) := b\_CO$

act3: $Concept\_definedIn\_DomainModel(o\_CO) := DomainModel\_corresp\_Component^{-1}(Constant\_definedIn\_Component(b\_CO))$

act4: $Concept\_parentConcept\_Concept(o\_CO) := PCO$

act5: $Concept\_isVariable(o\_CO) := FALSE$

act6: $Concept\_isEnumeration(o\_CO) := bool(olges \neq \emptyset)$

act7: $ConcreteEnumeration\_corresp\_IndividualSetLogicalFormula := ConcreteEnumeration\_corresp\_IndividualSetLogicalFormula$

$\cup\ (\{TRUE \mapsto \{o\_CO\} \times olges, FALSE \mapsto \emptyset\}(bool(olges \neq \emptyset)))$

act8: $Individual := Individual \cup (\{TRUE \mapsto o\_inds, FALSE \mapsto \emptyset\}(bool(olges \neq \emptyset)))$

act9: $Individual\_individualOf\_Concept := Individual\_individualOf\_Concept$

$\cup\ (\{TRUE \mapsto o\_inds \times \{o\_CO\}, FALSE \mapsto \emptyset\}(bool(olges \neq \emptyset)))$

act10: $Individual\_isVariable := Individual\_isVariable$

$\cup\ (\{TRUE \mapsto o\_inds \times \{FALSE\}, FALSE \mapsto \emptyset\}(bool(olges \neq \emptyset)))$

act11: $Individual\_isNamed := Individual\_isNamed$

$\cup\ (\{TRUE \mapsto o\_inds \times \{TRUE\}, FALSE \mapsto \emptyset\}(bool(olges \neq \emptyset)))$

act12: $Individual\_corresp\_Constant := Individual\_corresp\_Constant \cup (\{TRUE \mapsto b\_inds\_map\_o\_inds, FALSE \mapsto \emptyset\}(bool(olges \neq \emptyset)))$

act13: $Individual\_definedIn\_DomainModel := Individual\_definedIn\_DomainModel \cup (\{TRUE \mapsto o\_inds \times \{DomainModel\_corresp\_Component^{-1}(Constant\_definedIn\_Component(b\_CO))\}, FALSE \mapsto \emptyset\}(bool(olges \neq \emptyset)))$

end

Event rule_b_5 ⟨convergent⟩ $\widehat{=}$

(rule_b_5_1 & rule_b_5_2) Back propagating the addition of a constant typed as item of the correspondent of a concept

any b_ind o_ind b_CO_s b_CO_c b_CO_v b_lg CO olgis

where

grd0: $dom(Constant\_typing\_Property) \setminus (ran(Concept\_corresp\_Constant) \cup ran(Individual\_corresp\_Constant) \cup ran(Association\_Type\_Constant)) \neq \emptyset$

grd1: $b\_ind \in dom(Constant\_typing\_Property) \setminus (ran(Concept\_corresp\_Constant) \cup ran(Individual\_corresp\_Constant) \cup ran(Association\_Type\_Constant))$

grd2: $b\_lg = Constant\_typing\_Property(b\_ind)$

grd3: $LogicFormula\_uses\_Operators(b\_lg) = \{1 \mapsto Belonging\_OP\}$

grd4: $Concept \neq \emptyset \land CO \in Concept$

grd5: $Constant \neq \emptyset \land b\_CO\_c \in Constant$

grd6: $LogicFormula\_involves\_Sets(b\_lg) \neq \emptyset \Rightarrow (ran(Concept\_corresp\_Set) \neq \emptyset \land b\_CO\_s \in ran(Concept\_corresp\_Set) \land (2 \mapsto b\_CO\_s) \in LogicFormula\_involves\_Sets(b\_lg) \land CO = Concept\_corresp\_Set^{-1}(b\_CO\_s) \land \forall co \cdot (co \in Constant \Rightarrow (2 \mapsto b\_lg) \notin Constant\_isInvolvedIn\_LogicFormulas(co)))$

grd7: $LogicFormula\_involves\_Sets(b\_lg) = \emptyset \Rightarrow (ran(Concept\_corresp\_Constant) \neq \emptyset \land b\_CO\_c \in ran(Concept\_corresp\_Constant) \land (2 \mapsto b\_lg) \in Constant\_isInvolvedIn\_LogicFormulas(b\_CO\_c) \land CO = Concept\_corresp\_Constant^{-1}(b\_CO\_c) \land \forall co \cdot (co \in Constant \setminus \{b\_CO\_c\} \Rightarrow (2 \mapsto b\_lg) \notin Constant\_isInvolvedIn\_LogicFormulas(co)))$

grd8: $Individual\_Set \setminus Individual \neq \emptyset \land o\_ind \in Individual\_Set \setminus Individual$

grd9: $Constant\_definedIn\_Component(b\_ind) \in ran(DomainModel\_corresp\_Component)$

grd10: $Variable \neq \emptyset \land b\_CO\_v \in Variable$

grd11: $olgis = \{o\_lgi \cdot (o\_lgi \in Invariant \land LogicFormula\_uses\_Operators(o\_lgi) = \{1 \mapsto Belonging\_OP\} \land (1 \mapsto o\_lgi) \in Constant\_isInvolvedIn\_LogicFormulas(b\_ind) \land (2 \mapsto b\_CO\_v) \in Invariant\_involves\_Variables(o\_lgi)) | o\_lgi\}$

grd12: $olgis \neq \emptyset \Rightarrow (ran(Concept\_corresp\_Variable) \neq \emptyset \land b\_CO\_v \in ran(Concept\_corresp\_Variable) \land CO = Concept\_corresp\_Variable^{-1}(b\_CO\_v))$

then

act1: $Individual := Individual \cup \{o\_ind\}$

act2: $Individual\_corresp\_Constant(o\_ind) := b\_ind$

act3: $Individual\_definedIn\_DomainModel(o\_ind) := DomainModel\_corresp\_Component^{-1}(Constant\_definedIn\_Component(b\_ind))$

act4: $Individual\_individualOf\_Concept(o\_ind) := CO$

act5: $Individual\_isVariable(o\_ind) := FALSE$

act6: $Individual\_isNamed(o\_ind) := TRUE$

end

Event rule_b_6 ⟨convergent⟩ $\widehat{=}$

Back propagating the addition of a variable typed as subset of the correspondent of a concept

any b_CO o_CO b_PCO_s b_PCO_c b_PCO_v b_lg PCO

where

grd0: $dom(Variable\_typing\_Invariant) \setminus (ran(Concept\_corresp\_Variable) \cup ran(Individual\_corresp\_Variable) \cup ran(Association\_Type\_Variable)) \neq \emptyset$

grd1: $b\_CO \in dom(Variable\_typing\_Invariant) \setminus (ran(Concept\_corresp\_Variable) \cup ran(Individual\_corresp\_Variable) \cup ran(Association\_Type\_Variable))$

grd2: $b\_lg = Variable\_typing\_Invariant(b\_CO)$

grd3: $LogicFormula\_uses\_Operators(b\_lg) = \{1 \mapsto Inclusion\_OP\}$

grd4: $Concept \neq \emptyset \wedge PCO \in Concept$

grd5: $Constant \neq \emptyset \wedge b\_PCO\_c \in Constant$

grd6: $LogicFormula\_involves\_Sets(b\_lg) \neq \emptyset \Rightarrow (ran(Concept\_corresp\_Set) \neq \emptyset \wedge b\_PCO\_s \in ran(Concept\_corresp\_Set) \wedge (2 \mapsto b\_PCO\_s) \in LogicFormula\_involves\_Sets(b\_lg) \wedge PCO = Concept\_corresp\_Set^{-1}(b\_PCO\_s) \wedge \forall co \cdot (co \in Constant \Rightarrow (2 \mapsto b\_lg) \notin Constant\_isInvolvedIn\_LogicFormulas(co)) \wedge \forall co2 \cdot (co2 \in Variable \Rightarrow (2 \mapsto co2) \notin Invariant\_involves\_Variables(b\_lg)))$

grd7: $LogicFormula\_involves\_Sets(b\_lg) = \emptyset \Rightarrow ((ran(Concept\_corresp\_Constant) \neq \emptyset \wedge b\_PCO\_c \in ran(Concept\_corresp\_Constant) \wedge (2 \mapsto b\_lg) \in Constant\_isInvolvedIn\_LogicFormulas(b\_PCO\_c) \wedge PCO = Concept\_corresp\_Constant^{-1}(b\_PCO\_c) \wedge \forall co1 \cdot (co1 \in Constant \setminus \{b\_PCO\_c\} \Rightarrow (2 \mapsto b\_lg) \notin Constant\_isInvolvedIn\_LogicFormulas(co1)) \wedge \forall co2 \cdot (co2 \in Variable \Rightarrow (2 \mapsto co2) \notin Invariant\_involves\_Variables(b\_lg))) \vee (ran(Concept\_corresp\_Variable) \neq \emptyset \wedge b\_PCO\_v \in ran(Concept\_corresp\_Variable) \wedge (2 \mapsto b\_PCO\_v) \in Invariant\_involves\_Variables(b\_lg) \wedge PCO = Concept\_corresp\_Variable^{-1}(b\_PCO\_v) \wedge \forall co2 \cdot (co2 \in Constant \Rightarrow (2 \mapsto b\_lg) \notin Constant\_isInvolvedIn\_LogicFormulas(co2)) \wedge \forall co3 \cdot (co3 \in Variable \setminus \{b\_PCO\_v\} \Rightarrow (2 \mapsto co3) \notin Invariant\_involves\_Variables(b\_lg))))$

grd8: $Concept\_Set \setminus Concept \neq \emptyset \wedge o\_CO \in Concept\_Set \setminus Concept$

grd9: $Variable\_definedIn\_Component(b\_CO) \in ran(DomainModel\_corresp\_Component)$

then

act1: $Concept := Concept \cup \{o\_CO\}$

act2: $Concept\_corresp\_Variable(o\_CO) := b\_CO$

act3: $Concept\_definedIn\_DomainModel(o\_CO) := DomainModel\_corresp\_Component^{-1}(Variable\_definedIn\_Component(b\_CO))$

act4: $Concept\_parentConcept\_Concept(o\_CO) := PCO$

act5: $Concept\_isVariable(o\_CO) := TRUE$

act6: $Concept\_isEnumeration(o\_CO) := FALSE$

end

Event rule_b_7 ⟨convergent⟩ $\widehat{=}$

Back propagating the addition of a variable typed as item of the correspondent of a concept

any b_ind o_ind b_CO_s b_CO_c b_CO_v b_lg CO

where

grd0: $dom(Variable\_typing\_Invariant) \setminus (ran(Concept\_corresp\_Variable) \cup ran(Individual\_corresp\_Variable) \cup ran(Association\_Type\_Variable)) \neq \emptyset$

grd1: $b\_ind \in dom(Variable\_typing\_Invariant) \setminus (ran(Concept\_corresp\_Variable) \cup ran(Individual\_corresp\_Variable) \cup ran(Association\_Type\_Variable))$

grd2: $b\_lg = Variable\_typing\_Invariant(b\_ind)$

grd3: $LogicFormula\_uses\_Operators(b\_lg) = \{1 \mapsto Belonging\_OP\}$

grd4: $Concept \neq \emptyset \wedge CO \in Concept$

grd5: $Constant \neq \emptyset \wedge b\_CO\_c \in Constant$

grd6: $LogicFormula\_involves\_Sets(b\_lg) \neq \emptyset \Rightarrow (ran(Concept\_corresp\_Set) \neq \emptyset \wedge b\_CO\_s \in ran(Concept\_corresp\_Set) \wedge (2 \mapsto b\_CO\_s) \in LogicFormula\_involves\_Sets(b\_lg) \wedge CO = Concept\_corresp\_Set^{-1}(b\_CO\_s) \wedge \forall co \cdot (co \in Constant \Rightarrow (2 \mapsto b\_lg) \notin Constant\_isInvolvedIn\_LogicFormulas(co)))$

grd7: $LogicFormula\_involves\_Sets(b\_lg) = \emptyset \Rightarrow ((ran(Concept\_corresp\_Constant) \neq \emptyset \wedge b\_CO\_c \in ran(Concept\_corresp\_Constant) \wedge (2 \mapsto b\_lg) \in Constant\_isInvolvedIn\_LogicFormulas(b\_CO\_c) \wedge CO = Concept\_corresp\_Constant^{-1}(b\_CO\_c) \wedge \forall co \cdot (co \in Constant \setminus \{b\_CO\_c\} \Rightarrow (2 \mapsto b\_lg) \notin Constant\_isInvolvedIn\_LogicFormulas(co))) \vee (ran(Concept\_corresp\_Variable) \neq \emptyset \wedge b\_CO\_v \in ran(Concept\_corresp\_Variable) \wedge (2 \mapsto b\_CO\_v) \in Invariant\_involves\_Variables(b\_lg) \wedge CO = Concept\_corresp\_Variable^{-1}(b\_CO\_v) \wedge \forall co \cdot (co \in Constant \Rightarrow (2 \mapsto b\_lg) \notin Constant\_isInvolvedIn\_LogicFormulas(co))))$

grd8: $Individual\_Set \setminus Individual \neq \emptyset \wedge o\_ind \in Individual\_Set \setminus Individual$

grd9: $Variable\_definedIn\_Component(b\_ind) \in ran(DomainModel\_corresp\_Component)$

then

act1: $Individual := Individual \cup \{o\_ind\}$

act2: $Individual\_corresp\_Variable(o\_ind) := b\_ind$

act3: $Individual\_definedIn\_DomainModel(o\_ind) := DomainModel\_corresp\_Component^{-1}(Variable\_definedIn\_Component(b\_ind))$

act4: $Individual\_individualOf\_Concept(o\_ind) := CO$

act5: $Individual\_isVariable(o\_ind) := TRUE$

act6: *Individual_isNamed*(*o_ind*) : = *TRUE*

end

END

Annexe C

Guide d'utilisation de l'outil FORMOD

FORMOD désigne l'outil open source construit à partir d'Openflexo afin de supporter la méthodologie SysML/KAOS.

- Au lancement, l'outil *FORMOD* invite l'utilisateur à créer un projet (Figure C.1).
- Ensuite, l'utilisateur est invité à donner à son projet la nature *FORMOSE* afin de préciser qu'il s'agit d'un projet de spécification formelle des exigences d'un système critique (Figure C.2). En effet, l'outil permet de créer plusieurs autres catégories de projet [73].
- L'utilisateur peut ensuite instancier la méthodologie *SysML/KAOS* (Figure C.3), ce qui lui donne la possibilité de définir le modèle des buts du système conformément au langage SysML/KAOS de modélisation des buts (Figure C.4).
  Le modèle de la Figure C.4 définit deux niveaux de raffinement : le niveau racine `L0` qui introduit un but *Process* et le niveau `L1` qui décrit le raffinement de *Process* en trois sous buts (*Get*, *Compute* et *Put*).
- L'utilisateur peut par la suite instancier la méthodologie *Modèle de domaine* (Figure C.5), ce qui lui donne la possibilité de modéliser le domaine d'application du système conformément au langage SysML/KAOS de modélisation du domaine (Figure C.6). Chaque modèle de domaine est associé à un niveau de raffinement du modèle des buts fonctionnels.
  Le modèle de la Figure C.6 définit trois concepts et trois individus : (i) deux concepts `T_IN` et `T_OUT` reliés par une association `FB`, et (ii) deux individus variables `in` (individu de `T_IN`) et `out` (individu de `T_OUT`) reliés par un maplet `ma` (individu de `FB`).

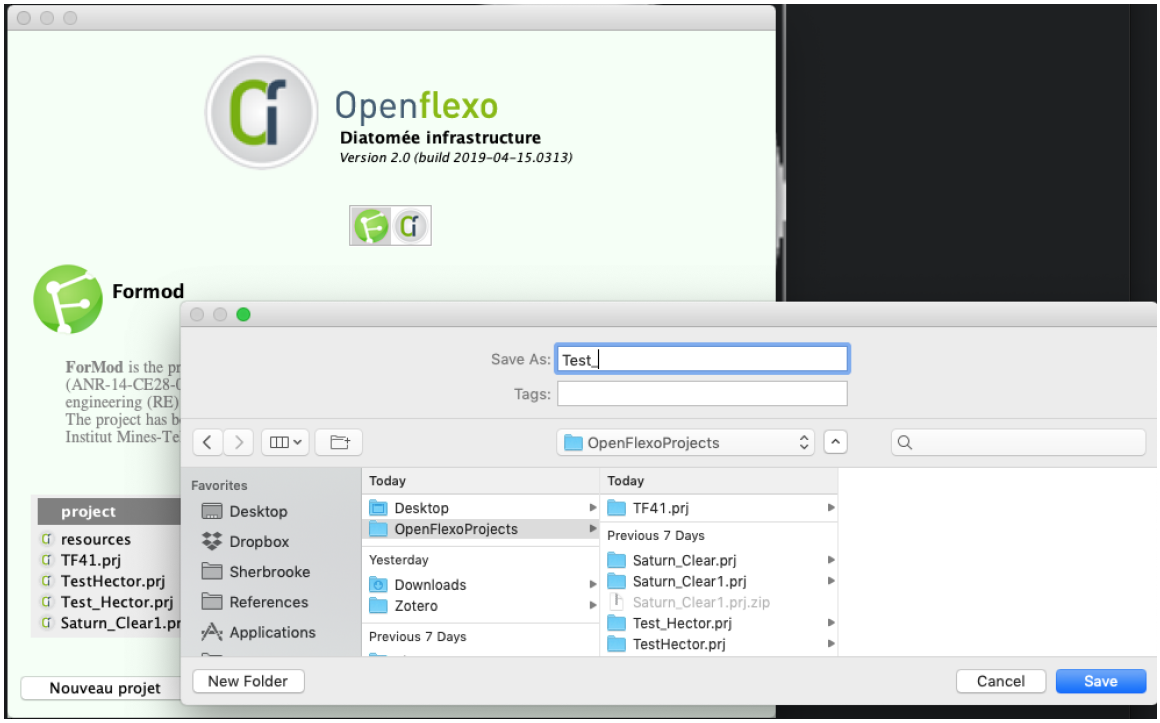


figure C.1 – Utilisation de FORMOD : Étape 1

- L'utilisateur peut finalement instancier la méthodologie *B System* (Figure C.7), ce qui lui donne l'occasion de référencer le projet *Atelier B* qui sera fédéré avec les modèles SysML/KAOS définis, conformément aux règles décrites à l'annexe B (Figure C.8).
  Le comportement *Update the formal model* (mettre à jour le modèle formel) permet de propager l'ajout et la suppression d'éléments au sein des modèles de buts et de domaine tandis que le comportement *Back propagate structural part updates* (propager les modifications de la partie structurelle) permet de propager l'ajout d'éléments au sein de la partie structurelle de la spécification *B System*.

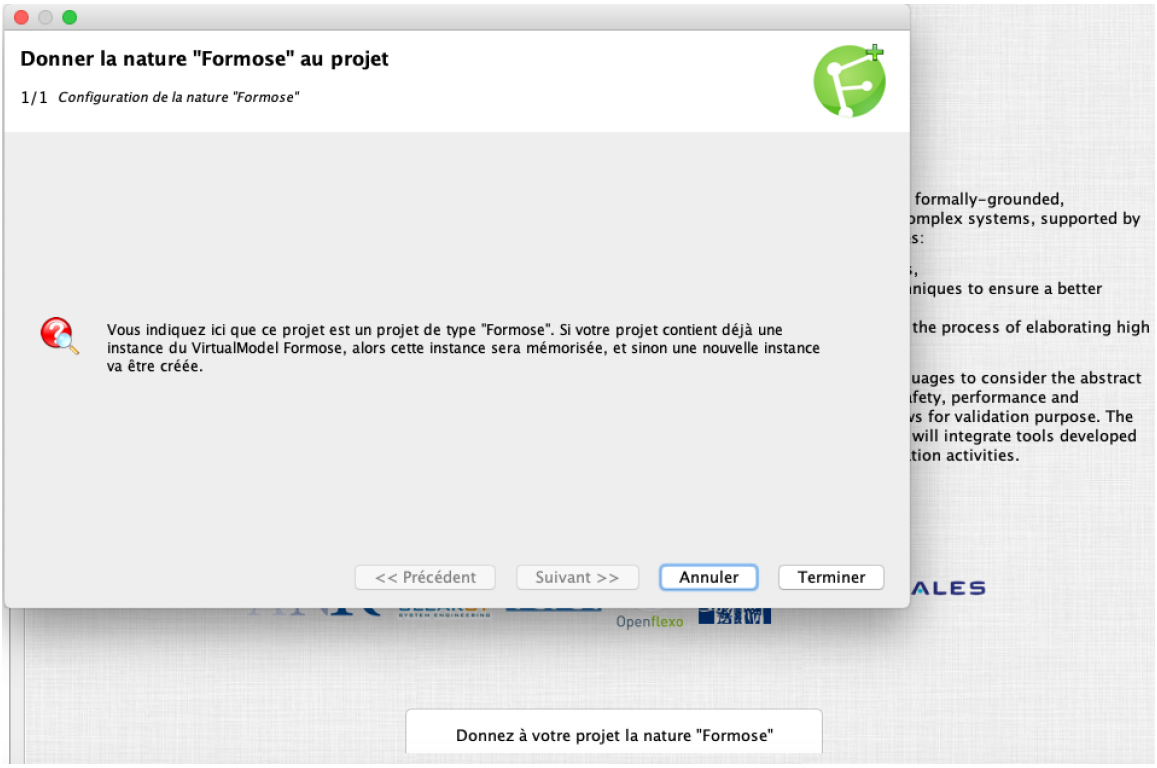


figure C.2 – Utilisation de FORMOD : Étape 2

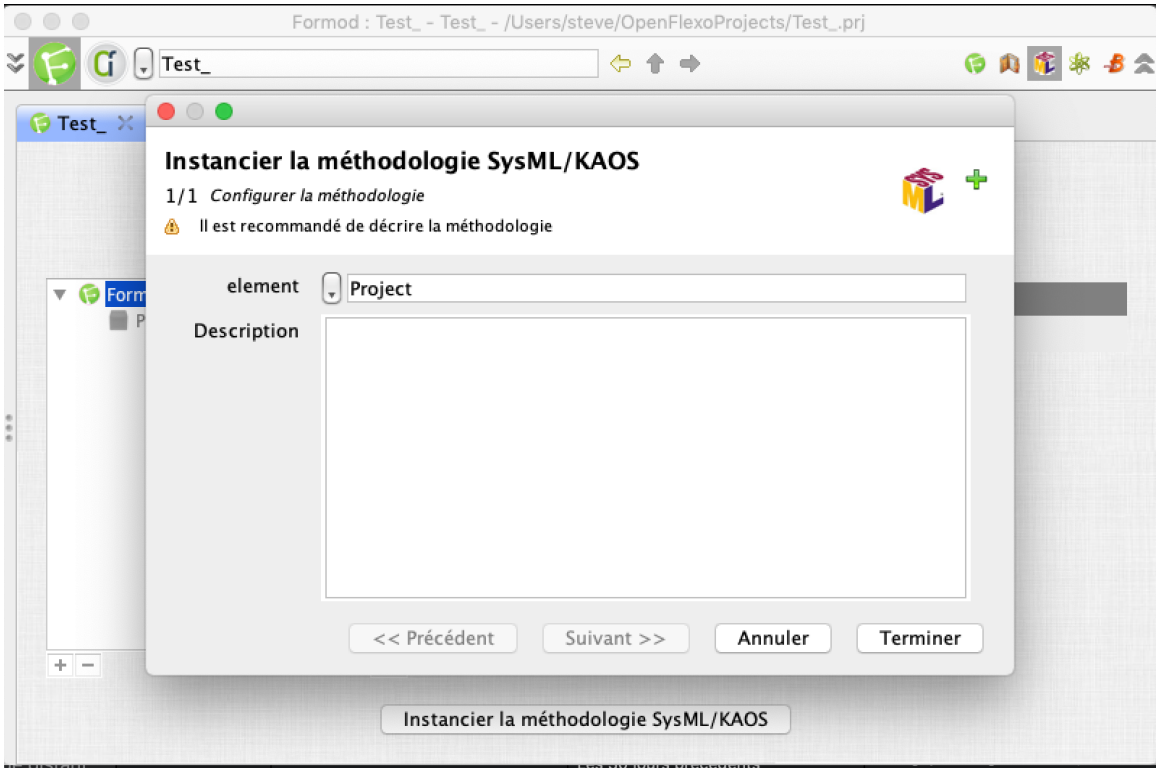


figure C.3 – Utilisation de FORMOD : Étape 3

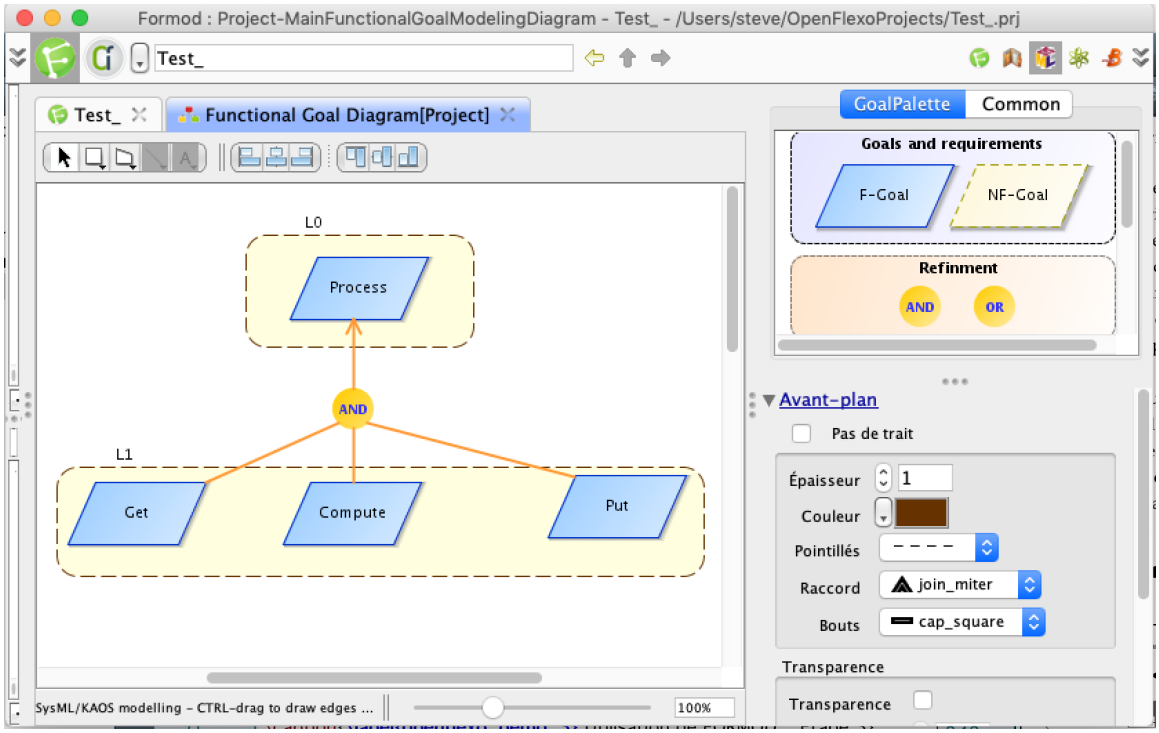


figure C.4 – Utilisation de FORMOD : Étape 4

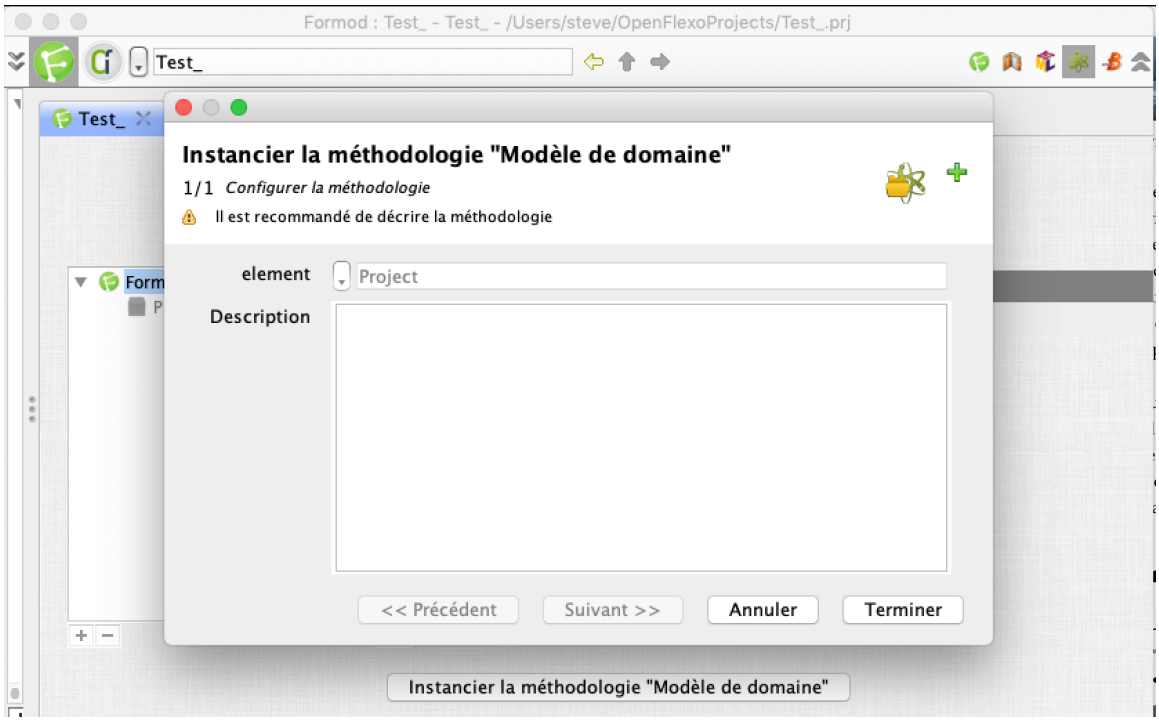


figure C.5 – Utilisation de FORMOD : Étape 5

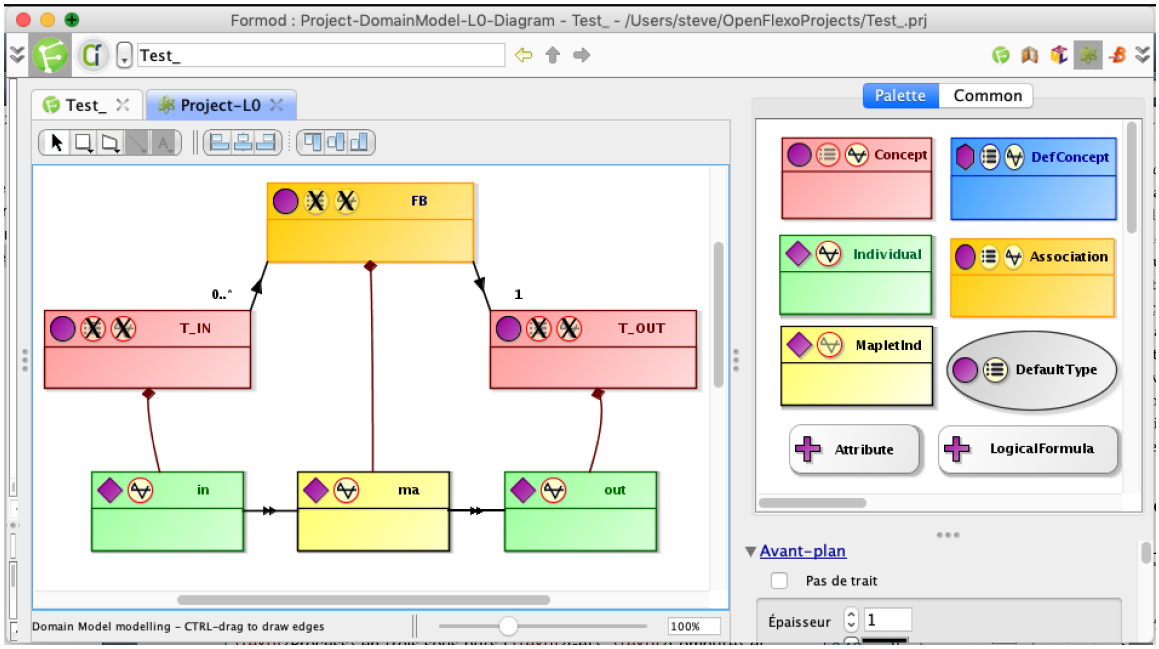


figure C.6 – Utilisation de FORMOD : Modèle de domaine associé au niveau de raffinement `L0`

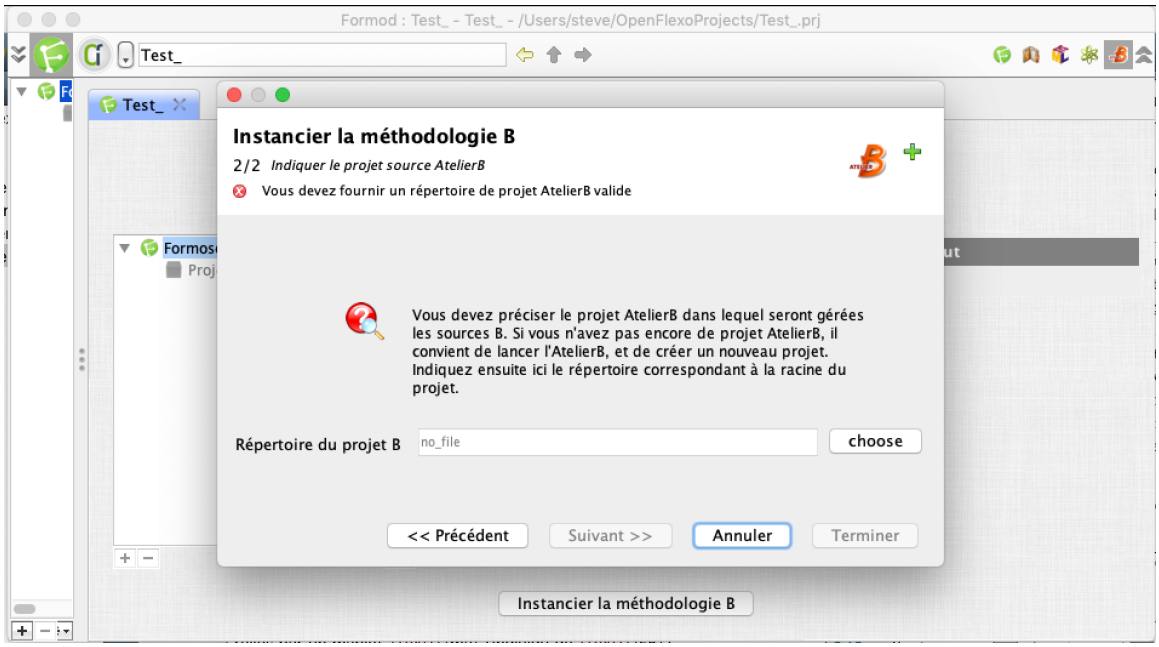


figure C.7 – Utilisation de FORMOD : Étape 7

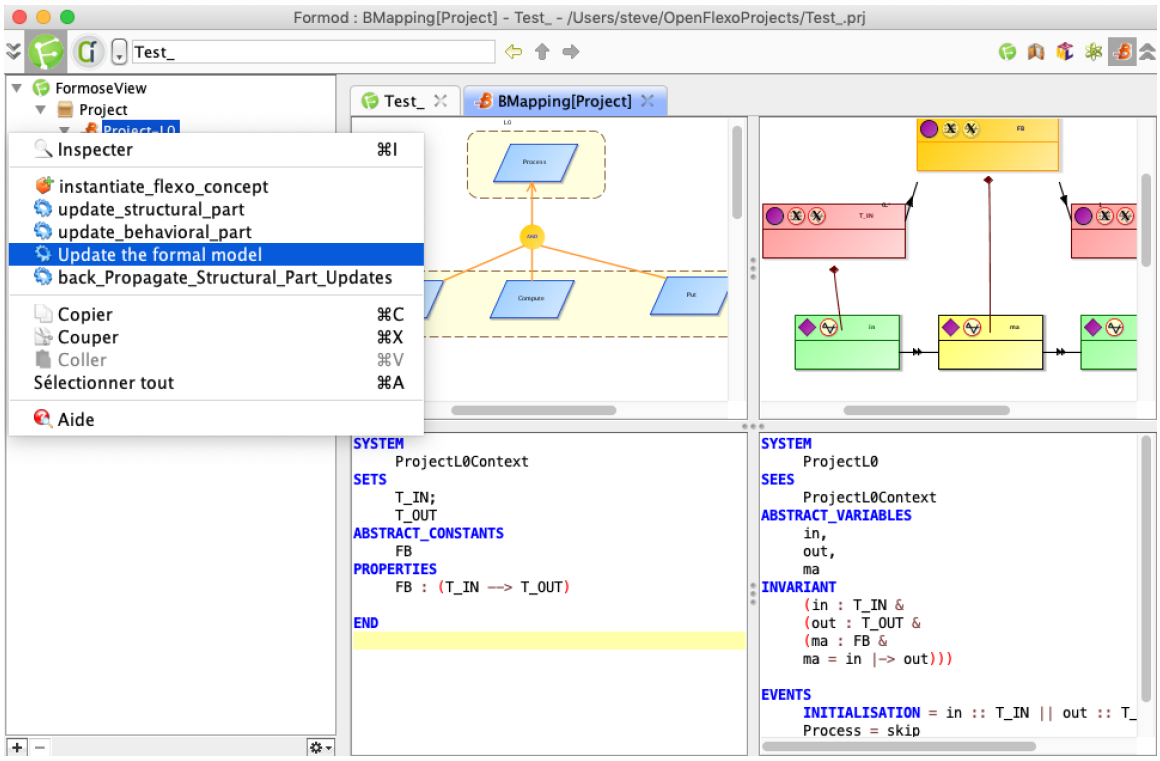


figure C.8 – Utilisation de FORMOD : Vue fédérée centrée sur le niveau de raffinement `L0`

Bibliographie

[1] « Road Transportation System : Description of Domain Models », 2018.
https ://github.com/stuenofotso/SysML_KAOS_Domain_Model_Parser/tree/master/-Bonaventure_project/Deliverables/Domain

[2] « Road Transportation System : Description of the Functional Goal Model », 2018.
https ://github.com/stuenofotso/SysML_KAOS_Domain_Model_Parser/tree/master/-Bonaventure_project/Deliverables/Functional

[3] « Road Transportation System : Description of the Non-Functional Goal Model », 2018.
https ://github.com/stuenofotso/SysML_KAOS_Domain_Model_Parser/tree/master/-Bonaventure_project/Deliverables/Non

[4] « Specification of the road transportation system - Rodin Project », 2018.
https ://github.com/stuenofotso/SysML_KAOS_Domain_Model_Parser/tree/master/-Bonaventure_project/Deliverables/ProjetBonaventure-Rodin_Project.

[5] « Specification of the road transportation system - Transformed Rodin Project », 2018.
https ://github.com/stuenofotso/SysML_KAOS_Domain_Model_Parser/tree/master/-Bonaventure_project/Deliverables/ProjetBonaventureAnimation.

[6] « SysML/KAOS Requirements Modeling of a Road Transportation System », 2018.
https ://github.com/stuenofotso/SysML_KAOS_Domain_Model_Parser/-tree/master/Bonaventure_project.

[7] Jean-Raymond Abrial.
The B-book - assigning programs to meanings.
Cambridge University Press, 2005.

[8] Jean-Raymond Abrial.
Modeling in Event-B - System and Software Engineering.
Cambridge University Press, 2010.

[9] Jean-Raymond Abrial.
« The ABZ-2018 Case Study with Event-B ».
Dans Butler et al. [36], pages 322–337.

[10] Jean-Raymond Abrial et Jean-Raymond Abrial.
The B-book : assigning programs to meanings.
Cambridge University Press, 2005.

[11] Jean-Raymond Abrial et Stefan Hallerstede.
« Refinement, decomposition, and instantiation of discrete models : Application to Event-B ».
Fundamenta Informaticae, 77(1-2):1–28, 2007.

[12] Eman Alkhammash, Michael J. Butler, Asieh Salehi Fathabadi et Corina Cîrstea.
« Building traceable Event-B models from requirements ».
Sci. Comput. Program., 111:318–338, 2015.

[13] Alkhammash, Eman H..
« Derivation of Event-B Models from OWL Ontologies ».
MATEC Web Conf., 76:04008, 2016.

[14] Rajeev Alur, Thomas A Henzinger et Orna Kupferman.
« Alternating-time temporal logic ».
Journal of the ACM (JACM), 49(5):672–713, 2002.

[15] Yamine Aït Ameur, Mickaël Baron, Ladjel Bellatreche, Stéphane Jean et Eric Sardet.
« Ontologies in engineering : the OntoDB/OntoQL platform ».
Soft Comput., 21(2):369–389, 2017.

[16] ANR-06-SETIN-017.
« TACOS ANR Project », 2017.

[17] ANR-14-CE28-0009.
« Formose ANR Project », 2017.

[18] Paolo Arcaini, Pavel Jezek et Jan Kofron.
« Modelling the Hybrid ERTMS/ETCS Level 3 Case Study in Spin ».
Dans Butler et al. [36], pages 277–291.

[19] Chetan Arora, Mehrdad Sabetzadeh et Lionel Briand.
« An Empirical Study on the Potential Usefulness of Domain Models for Completeness Checking of Requirements ».
Empirical Software Engineering.

[20] Robert Arp, Barry Smith et Andrew D Spear.
Building ontologies with Basic Formal Ontology.
Mit Press, 2015.

[21] B Atelier.
« the industrial tool to efficiently deploy the B Method ».
URL : http ://www. atelierb. eu/index-en. php (access date 22.03. 2015), 2008.

[22] Jean-Claude Bauer.
« Specification for a software program for a boiler water content monitor and control system ».
Institute of Risk Research, University of Waterloo, 1993.

[23] Patrick Behm, Paul Benoit, Alain Faivre et Jean-Marc Meynadier.
« Météor : A Successful Application of B in a Large Project ».
Dans Jeannette M. Wing, Jim Woodcock et Jim Davies, éditeurs, *FM'99 - Formal Methods, World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, September 20-24, 1999, Proceedings, Volume I*, volume 1708 de *Lecture Notes in Computer Science*, pages 369–387. Springer, 1999.

[24] Dines Bjørner et Asger Eir.
« Compositionality : Ontology and Mereology of Domains ».
volume 5930 de *Essays in Honor of Willem-Paul de Roever, LNCS*, pages 22–59. Springer, 2010.

[25] Dines Bjrner.
« Domain Analysis and Description Principles, Techniques, and Modelling Languages ».
ACM Trans. Softw. Eng. Methodol., 28(2):8 :1–8 :67, 2019.

[26] Dines Bjørner.
« Domain Analysis and Description Principles, Techniques, and Modelling Languages ».
ACM Trans. Softw. Eng. Methodol. 28, 2, 67 pages, mars 2019.

[27] Jean-Paul Bodeveix, Mamoun Filali, Mohamed Tahar Bhiri et Badr Siala.
« An Event-B framework for the validation of Event-B refinement plugins ».
CoRR, abs/1701.00960, 2017.

[28] Tommaso Bolognesi et Ed Brinksma.
« Introduction to the ISO specification language LOTOS ».
Computer Networks and ISDN systems, 14(1):25–59, 1987.

[29] Frédéric Boniol et Virginie Wiels.
« The Landing Gear System Case Study ».
ABZ, pages 1–18. Springer, 2014.

[30] Manfred Broy.
« Domain Modeling and Domain Engineering : Key Tasks in Requirements Engineering ».
Dans *Perspectives on the Future of Software Engineering : Essays in Honor of Dieter Rombach*, pages 15–30.
Springer Berlin Heidelberg, 2013.

[31] Manfred Broy.
« Domain Modeling and Domain Engineering : Key Tasks in Requirements Engineering ».
Dans Jürgen Münch et Klaus Schmid, éditeurs, *Perspectives on the Future of Software Engineering : Essays in Honor of Dieter Rombach*, pages 15–30. Springer Berlin Heidelberg, 2013.

[32] Julien Brunel, David Chemouil, Alcino Cunha et Nuno Macedo.
« The Electrum analyzer : model checking relational first-order temporal specifications ».
Dans *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, pages 884–887. ACM, 2018.

[33] Michael Butler.
« Synchronisation-based decomposition for event-B ».
RODIN Deliverable D19 Intermediate report on methodology, pages 47–57, 2006.

[34] Michael J. Butler.
« An Approach to the Design of Distributed Systems with B AMN ».
Dans Jonathan P. Bowen, Michael G. Hinchey et David Till, éditeurs, *ZUM '97 : The Z Formal Specification Notation, 10th International Conference of Z Users, Reading, UK, April 3-4, 1997, Proceedings*, volume 1212 de *Lecture Notes in Computer Science*, pages 223–241. Springer, 1997.

[35] Michael J. Butler, Cliff B. Jones, Alexander Romanovsky et Elena Troubitsyna, éditeurs.
« Rigorous Development of Complex Fault-Tolerant Systems [FP6 IST-511599 RODIN project] », volume 4157 de *Lecture Notes in Computer Science*. Springer, 2006.

[36] Michael J. Butler, Alexander Raschke, Thai Son Hoang et Klaus Reichl, éditeurs.
« Abstract State Machines, Alloy, B, TLA, VDM, and Z - 6th International Conference, ABZ 2018, Southampton, UK, June 5-8, 2018, Proceedings », volume 10817 de *Lecture Notes in Computer Science*. Springer, 2018.

[37] Paulo JF Carreira et Miguel EF Costa.
« Automatically verifying an object oriented specification of the steam-boiler system ».
Dans *Proceedings of the 5th International ERCIM Workshop on Formal Methods for Industrial Critical Systems (FMICS'2000)*, pages 345–360, 2000.

[38] Lawrence Chung, Brian A Nixon, Eric Yu et John Mylopoulos.
Non-functional requirements in software engineering.
volume 5. Springer Science & Business Media, 2012.

[39] Arne F Claassen, Ruby D Lathon, Daniel M Rochowiak et Leslie D Interrante.
« Active Rescheduling for Goal Maintenance in Dynamic Manufacturing-Systems ».
Dans *Proceedings of the AAAI Spring Symposium*, 1994.

[40] Tom Clancy.
« The Standish Group CHAOS Report ».
Project Smart, pages 8–9, 2014.

[41] ClearSy.
« Atelier B : B System », 2014.
http ://clearsy.com/.

[42] ClearSy.
« ClearSy Systems Engineering », 2019.
http ://clearsy.com/.

[43] Andrew W. Crapo, Abha Moitra, Craig McMillan et Daniel Russell.
« Requirements Capture and Analysis in ASSERT(TM) ».
Dans *25th IEEE International Requirements Engineering Conference, RE 2017, Lisbon, Portugal, September 4-8, 2017*, pages 283–291. IEEE Computer Society, 2017.

[44] Alcino Cunha et Nuno Macedo.
« Validating the Hybrid ERTMS/ETCS Level 3 Concept with Electrum ».
Dans Butler et al. [36], pages 307–321.

[45] Wayne J Davis.
« Evaluating Performance of Distributed Intelligent Control Systems ».
NIST SPECIAL PUBLICATION SP, pages 225–232, 2001.

[46] Ricardo de Almeida Falbo, Giancarlo Guizzardi et Katia Cristina Duarte.
« An ontological approach to domain engineering ».
Dans *Proceedings of the 14th international conference on Software engineering and knowledge engineering, SEKE 2002, Ischia, Italy, July 15-19, 2002*, pages 351–358. ACM, 2002.

[47] Ville de Montréal.
« Annex 1 : Schematic view of the intervention area », 2014.
https ://github.com/stuenofotso/SysML_KAOS_Domain_Model_Parser/blob/master/-Bonaventure_project/Reference

[48] Ville de Montréal.
« Final draft of the project to connect Duke and Nazareth Streets to the Ville-Marie highway », 2014.
https ://github.com/stuenofotso/SysML_KAOS_Domain_Model_Parser/blob/master/-Bonaventure_project/Reference

[49] Babak Dehbonei et Fernando Mejia.
« Formal Methods in the Railways Signalling Industry ».
Dans Maurice Naftalin, B. Tim Denvir et Miquel Bertran, éditeurs, *FME '94 : Industrial Benefit of Formal Methods, Second International Symposium of Formal Methods Europe, Barcelona, Spain, October 24-18, 1994, Proceedings*, volume 873 de *Lecture Notes in Computer Science*, pages 26–34. Springer, 1994.

[50] Diego Dermeval, Jéssyka Vilela, Ig Ibert Bittencourt, Jaelson Castro, Seiji Isotani, Patrick Henrique da S. Brito et Alan Silva.
« Applications of ontologies in requirements engineering : a systematic review of the literature ».
Requir. Eng., 21(4):405–437, 2016.

[51] Vladan Devedzic.
« Knowledge modeling - State of the art ».
Integrated Computer-Aided Engineering, 8(3):257–281, 2001.

[52] Dana Dghaym, Michael Poppleton et Colin F. Snook.
« Diagram-Led Formal Modelling Using iUML-B for Hybrid ERTMS Level 3 ».
Dans Butler et al. [36], pages 338–352.

[53] Jin Song Dong, Jing Sun et Hai H. Wang.
« Z Approach to Semantic Web ».
Dans *Formal Methods and Software Engineering - ICFEM, LNCS*, volume 2495, pages 156–167. Springer, 2002.

[54] EEIG ERTMS Users Group.
« Hybrid ERTMS/ETCS Level 3 : Principles ».
Ref. 16E042 Version 1C, juillet 2018.

[55] Sean Falconer.
« Protégé - OntoGraph », 2010.
http ://protegewiki.stanford.edu/wiki/OntoGraf.

[56] Steve Jeffrey Tueno Fotso.
« SysML/KAOS Domain Model Parser », 2017.
https ://github.com/stuenofotso/SysML_KAOS_Domain_Model_Parser.

[57] Steve Jeffrey Tueno Fotso, Marc Frappier, Regine Laleau et Amel Mammar. « Back Propagating B System Updates on SysML/KAOS Domain Models ». Dans *23rd International Conference on Engineering of Complex Computer Systems, ICECCS 2018, Melbourne, Australia, December 12-14, 2018*, pages 160–169. IEEE Computer Society, 2018.

[58] Steve Jeffrey Tueno Fotso, Marc Frappier, Régine Laleau et Amel Mammar. « Modeling the Hybrid ERTMS/ETCS Level 3 Standard Using a Formal Requirements Engineering Approach ». Dans Butler et al. [36], pages 262–276.

[59] Steve Jeffrey Tueno Fotso, Marc Frappier, Régine Laleau, Amel Mammar et Michael Leuschel. « Formalisation of SysML/KAOS Goal Assignments with B System Component Decompositions ». Dans Carlo A. Furia et Kirsten Winter, éditeurs, *Integrated Formal Methods - 14th International Conference, IFM 2018, Maynooth, Ireland, September 5-7, 2018, Proceedings*, volume 11023 de *Lecture Notes in Computer Science*, pages 377–397. Springer, 2018.

[60] Steve Jeffrey Tueno Fotso, Regine Laleau, Hector Ruiz Barradas, Amel Mammar et Marc Frappier. « A Formal Requirements Modeling Approach : Application to Rail Communication ». Dans *Proceedings of the 14th International Conference on Software Technologies, ICSOFT 2019, Prague, Czech Republic, July 26-28, 2019*. SciTePress, 2019.

[61] Steve Jeffrey Tueno Fotso, Regine Laleau, Amel Mammar et Marc Frappier. « Towards Using Ontologies for Domain Modeling within the SysML/KAOS Approach ». Dans *IEEE 25th International Requirements Engineering Conference Workshops, RE 2017 Workshops, Lisbon, Portugal, September 4-8, 2017*, pages 1–5. IEEE Computer Society, 2017.

[62] Steve Jeffrey Tueno Fotso, Regine Laleau, Amel Mammar et Marc Frappier. « Modélisation du Domaine au Sein d'une Méthode Formelle d'Ingénierie des Exigences ». Dans *AFADL 2019 : 18èmes journées des Approches Formelles dans l'Assistance au Développement de Logiciels*. AFADL, 2019.

[63] Steve Jeffrey Tueno Fotso, Régine Laleau, Amel Mammar et Marc Frappier. « Formal Representation of SysML/KAOS Domain Model (Complete Version) ». *ArXiv e-prints, cs.SE, 1712.07406*, décembre 2017.

[64] Steve Jeffrey Tueno Fotso, Régine Laleau, Amel Mammar et Marc Frappier.
« The SysML/KAOS Domain Modeling Approach ».
ArXiv e-prints, cs.SE, 1710.00903, septembre 2017.

[65] Steve Jeffrey Tueno Fotso, Amel Mammar, Régine Laleau et Marc Frappier.
« Event-B Expression and Verification of Translation Rules Between SysML/-KAOS Domain Models and B System Specifications ».
Dans Butler et al. [36], pages 55–70.

[66] Marc Frappier, Frédéric Gervais, Régine Laleau, Benoît Fraikin et Richard St.-Denis.
« Extending statecharts with process algebra operators ».
ISSE, 4(3):285–292, 2008.

[67] Etienne M Gagnon et Laurie J Hendren.
SableCC, an object-oriented compiler framework.
IEEE, 1998.

[68] Christophe Gnaho, Régine Laleau, Farida Semmak et Jean-Michel Bruel.
« bCMS requirements modelling using SysML/KAOS ».

[69] Christophe Gnaho et Farida Semmak.
« Une extension SysML pour l'ingénierie des exigences non fonctionnelles orientée but ».
Ingénierie des Systèmes d'Information, 16(1):9–32, 2011.

[70] Christophe Gnaho, Farida Semmak et Régine Laleau.
« Modeling the Impact of Non-functional Requirements on Functional Requirements ».
Dans Jeffrey Parsons et Dickson K. W. Chiu, éditeurs, *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 de *Lecture Notes in Computer Science*, pages 59–67. Springer, 2013.

[71] Christophe Gnaho, Farida Semmak et Regine Laleau.
« An overview of a SysML extension for goal-oriented NFR modelling ».
Dans Roel Wieringa, Selmin Nurcan, Colette Rolland et Jean-Louis Cavarero, éditeurs, *RCIS 2013, Paris, France, May 29-31, 2013*, pages 1–2. IEEE, 2013.

[72] Fahad Rafique Golra, Antoine Beugnard, Fabien Dagnat, Sylvain Guérin et Christophe Guychard.
« Addressing modularity for heterogeneous multi-model systems using model federation ».
Dans Lidia Fuentes, Don S. Batory et Krzysztof Czarnecki, éditeurs, *Companion Proceedings of the 15th International Conference on Modularity, Málaga, Spain, March 14 - 18, 2016*, pages 206–211. ACM, 2016.

[73] Fahad Rafique Golra, Fabien Dagnat, Jeanine Souquières, Imen Sayar et Sylvain Guérin.
« Bridging the Gap Between Informal Requirements and Formal Specifications Using Model Federation ».
Dans Einar Broch Johnsen et Ina Schaefer, éditeurs, *Software Engineering and Formal Methods - 16th International Conference, SEFM 2018, Held as Part of STAF 2018, Toulouse, France, June 27-29, 2018, Proceedings*, volume 10886 de *Lecture Notes in Computer Science*, pages 54–69. Springer, 2018.

[74] Valentin Goranko.
« Coalition games and alternating temporal logics ».
Dans *Proceedings of the 8th conference on Theoretical aspects of rationality and knowledge*, pages 259–272. Morgan Kaufmann Publishers Inc., 2001.

[75] Thomas R Gruber.
« A translation approach to portable ontology specifications ».
Knowledge acquisition, 5(2):199–220, 1993.

[76] Kahina Hacid et Yamine Aït Ameur.
« Strengthening MDE and Formal Design Models by References to Domain Ontologies. A Model Annotation Based Approach ».
Dans Tiziana Margaria et Bernhard Steffen, éditeurs, *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 de *Lecture Notes in Computer Science*, pages 340–357, 2016.

[77] Dominik Hansen, Michael Leuschel, David Schneider, Sebastian Krings, Philipp Körner, Thomas Naulin, Nader Nayeri et Frank Skowron.
« Using a Formal B Model at Runtime in a Demonstration of the ETCS Hybrid Level 3 Concept with Real Trains ».
Dans Butler et al. [36], pages 292–306.

[78] Matthew Hause et others.
« The SysML modelling language ».
Dans *Fifteenth European Systems Engineering Conference*, volume 9. Citeseer, 2006.

[79] Thai Son Hoang, Michael J. Butler et Klaus Reichl.
« The Hybrid ERTMS/ETCS Level 3 Case Study ».
Dans Butler et al. [36], pages 251–261.

[80] Gerard J. Holzmann.
The SPIN Model Checker - primer and reference manual.
Addison-Wesley, 2004.

[81] Alexei Iliasov, Elena Troubitsyna, Linas Laibinis, Alexander B. Romanovsky, Kimmo Varpaaniemi, Dubravka Ilic et Timo Latvala.
« Supporting Reuse in Event B Development : Modularisation Approach ».
Dans Marc Frappier, Uwe Glässer, Sarfraz Khurshid, Régine Laleau et Steve Reeves, éditeurs, *Abstract State Machines, Alloy, B and Z, Second International Conference, ABZ 2010, Orford, QC, Canada, February 22-25, 2010. Proceedings*, volume 5977 de *Lecture Notes in Computer Science*, pages 174–188. Springer, 2010.

[82] Télécommunications Grimard Inc..
« Functional Description of the Automatic Incident Detector (AID) System », 2018.
https ://github.com/stuenofotso/SysML_KAOS_Domain_Model_Parser/blob/master/-Bonaventure_project/Reference

[83] Daniel Jackson.
Software Abstractions - Logic, Language, and Analysis.
MIT Press, 2006.

[84] Daniel Jackson.
Software Abstractions : logic, language, and analysis.
MIT press, 2012.

[85] Michael Jackson et Pamela Zave.
« Domain descriptions ».
Dans *Proceedings of IEEE International Symposium on Requirements Engineering, RE 1993, San Diego, California, USA, January 4-6, 1993*, pages 56–64. IEEE Computer Society, 1993.

[86] Michael A. Jackson.
Software requirements and specifications - a lexicon of practice, principles and prejudices.
Addison-Wesley, 1995.

[87] Jetbrains.
« Jetbrains MPS », 2017.
https ://www.jetbrains.com/mps/.

[88] Michael Kifer et Georg Lausen.
« F-Logic : A Higher-Order language for Reasoning about Objects, Inheritance, and Scheme ».
Dans *Proceedings of the 1989 ACM SIGMOD*, pages 134–146. ACM Press, 1989.

[89] Motohiro Kitamura, Ryo Hasegawa, Haruhiko Kaiya et Motoshi Saeki.
« An Integrated Tool for Supporting Ontology Driven Requirements Elicitation ».
Dans Joaquim Filipe, Boris Shishkov et Markus Helfert, éditeurs, *ICSOFT 2007, Volume SE*, pages 73–80. INSTICC Press, 2007.

[90] Lukas Ladenberger, Jens Bendisposto et Michael Leuschel.
« Visualising Event-B models with B-Motion Studio ».
Dans *International Workshop on Formal Methods for Industrial Critical Systems*, pages 202–204. Springer, 2009.

[91] Régine Laleau et Amel Mammar.
« An Overview of a Method and Its Support Tool for Generating B Specifications from UML Notations ».
Dans *The Fifteenth IEEE International Conference on Automated Software Engineering, ASE 2000, Grenoble, France, September 11-15, 2000*, pages 269–272. IEEE Computer Society, 2000.

[92] Régine Laleau, Farida Semmak, Abderrahman Matoussi, Dorian Petit, Ahmed Hammad et Bruno Tatibouet.
« A first attempt to combine SysML requirements diagrams and B ».
Innovations in Systems and Software Engineering, 6(1-2):47–54, 2010.

[93] Thierry Lecomte, David Déharbe, Étienne Prun et Erwan Mottin.
« Applying a Formal Method in Industry : A 25-Year Trajectory ».
Dans Simone André da Costa Cavalheiro et José Luiz Fiadeiro, éditeurs, *Formal Methods : Foundations and Applications - 20th Brazilian Symposium, SBMF 2017, Recife, Brazil, November 29 - December 1, 2017, Proceedings*, volume 10623 de *Lecture Notes in Computer Science*, pages 70–87. Springer, 2017.

[94] Dai Gil Lee et Nam Pyo Suh.
« Axiomatic design and fabrication of composite structures-applications in robots, machine tools, and automobiles ».
Oxford University Press, page 732, 2005.

[95] Michael Leuschel et Michael J. Butler.
« ProB : A Model Checker for B ».
Dans Keijiro Araki, Stefania Gnesi et Dino Mandrioli, éditeurs, *FME 2003 : Formal Methods, International Symposium of Formal Methods Europe, Pisa, Italy, September 8-14, 2003, Proceedings*, volume 2805 de *Lecture Notes in Computer Science*, pages 855–874. Springer, 2003.

[96] Amel Mammar, Marc Frappier, Steve Jeffrey Tueno Fotso et Régine Laleau.
« An Event-B Model of the Hybrid ERTMS/ETCS Level 3 Standard ».
Dans Butler et al. [36], pages 353–366.

[97] Amel Mammar, Marc Frappier, Steve Jeffrey Tueno Fotso et Régine Laleau.
« An Event-B Model of the Hybrid ERTMS/ETCS Level 3 Standard ».
Dans Butler et al. [36], pages 353–366.

[98] Amel Mammar et Régine Laleau.
« On the Use of Domain and System Knowledge Modeling in Goal-Based Event-B Specifications ».
Dans *ISoLA 2016, LNCS*, pages 325–339. Springer, 2016.

[99] Atif Mashkoor et Jean-Pierre Jacquot.
« Utilizing Event-B for domain engineering : a critical analysis ».
Requir. Eng., 16(3):191–207, 2011.

[100] Atif Mashkoor et Jean-Pierre Jacquot.
« Validation of formal specifications through transformation and animation ».
Requir. Eng., 22(4):433–451, 2017.

[101] Abderrahman Matoussi.
« *Building abstract formal Specifications driven by goals* ».
Thèse de doctorat, University of Paris-Est, France, 2011.

[102] Abderrahman Matoussi, Frédéric Gervais et Régine Laleau.
« A Goal-Based Approach to Guide the Design of an Abstract Event-B Specification ».
Dans *ICECCS 2011*, pages 139–148. IEEE Computer Society, 2011.

[103] William E. McUmber et Betty H. C. Cheng.
« A General Framework for Formalizing UML with Formal Languages ».
Dans *ICSE 2001*, pages 433–442. IEEE Computer Society, 2001.

[104] Patrice Micouin, Louis Fabre, Roland Becquet, Thomas Razafimahefa et François Guérin.
« Property Model Methodology : A Landing Gear Operational Use Case ».
2018.

[105] Jérémy Milhau, Marc Frappier, Frédéric Gervais et Régine Laleau.
« Systematic Translation Rules from ASTD to Event-B ».
Dans *IFM*, volume 6396 de *Lecture Notes in Computer Science*, pages 245–259. Springer, 2010.

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

[107] Furness Nicola, van Houten Henri, Arenas Laura et Bartholomeus Maarten.
« ERTMS Level 3 : the Game-Changer ».
IRSE News View, page 232, avril 2017.

[108] Bashar Nuseibeh.
« Weaving together requirements and architectures ».
Computer, 34(3):115–119, 2001.

[109] Openflexo.
« Openflexo Project », 2015.
http ://www.openflexo.org.

[110] Openflexo.
« Openflexo SysML/KAOS Tool », 2019.
https ://downloads.openflexo.org/Formose/.

[111] David Lorge Parnas et Jan Madey.
« Functional documents for computer systems ».
Science of Computer programming, 25(1):41–61, 1995.

[112] Guy Pierra.
« The PLIB ontology-based approach to data integration ».
Dans *IFIP 18th World Computer Congress*, volume 156 de *IFIP*, pages 13–18.
Kluwer/Springer, 2004.

[113] Guy Pierra.
« Context Representation in Domain Ontologies and Its Use for Semantic Integration of Data ».
J. Data Semantics, 10:174–211, 2008.

[114] National Post.
« Here's the terrifying reason Boeing's 737 MAX 8 is grounded across the globe », 2019.
https ://nationalpost.com/news/heres-the-terrifying-reason-the-737-max-8-is-grounded.

[115] Deploy Project.
« Rodin Atelier B Provers Plug-in », 2017.
https ://www3.hhu.de/stups/handbook/rodin/current/html/atelier_b_provers.html.

[116] A Roques.
« Plantuml : Open-source tool that uses simple textual descriptions to draw uml diagrams », 2015.

[117] Sepanta Sekhavat et Jorge Hermosillo Valadez.
« The Cycab robot : a differentially flat system ».
Dans *IROS 2000*, pages 312–317. IEEE, 2000.

[118] Kunal Sengupta et Pascal Hitzler.
« Web Ontology Language (OWL) ».
Dans *Encyclopedia of Social Network Analysis and Mining*, pages 2374–2378. 2014.

[119] Amílcar Sernadas, Cristina Sernadas, Paula Gouveia, Pedro Resende et João Gouveia.
« OBLOG| Object-Oriented Logic : an informal introduction ».
Techn. Report, INESC, Lisbon, 1991.

[120] Masayuki Shibaoka, Haruhiko Kaiya et Motoshi Saeki.
« GOORE : Goal-Oriented and Ontology Driven Requirements Elicitation Method ».
Dans *ER 2007 Workshops*, volume 4802 de *Lecture Notes in Computer Science*, pages 225–234. Springer, 2007.

[121] Renato Silva.
« Towards the composition of specifications in Event-B ».
Electronic Notes in Theoretical Computer Science, 280:81–93, 2011.

[122] Renato Silva et Michael J. Butler.
« Shared Event Composition/Decomposition in Event-B ».
Dans Bernhard K. Aichernig, Frank S. de Boer et Marcello M. Bonsangue, éditeurs, *Formal Methods for Components and Objects - 9th International Symposium, FMCO 2010, Graz, Austria, November 29 - December 1, 2010. Revised Papers*, volume 6957 de *Lecture Notes in Computer Science*, pages 122–141. Springer, 2010.

[123] Renato Silva, Carine Pascal, Thai Son Hoang et Michael J. Butler.
« Decomposition tool for event-B ».
Softw., Pract. Exper., 41(2):199–208, 2011.

[124] Colin Snook et Michael Butler.
« UML-B : Formal Modeling and Design Aided by UML ».
ACM Trans. Softw. Eng. Methodol., 15(1):92–122, janvier 2006.

[125] Colin F. Snook et Michael J. Butler.
« UML-B : Formal modeling and design aided by UML ».
ACM Trans. Softw. Eng. Methodol., 15(1):92–122, 2006.

[126] GJ Suski, JM Duffy, DG Gritton, FW Holloway, JE Krammen, RG Ozarski, JR Severyn et PJ Van Arsdall.
« The Nova Control System–Goals, Architecture, and System Design ».
Dans *Distributed Computer Control Systems 1982*, pages 45–56. Elsevier, 1983.

[127] SYSTEREL.
« Rodin SMT Solvers Plug-in », 2017.
http ://wiki.event-b.org/index.php/SMT_Solvers_Plug-in.

[128] Joseph KH Tan.
Health management information systems : Methods and practical applications.
Jones & Bartlett Learning, 2001.

[129] Railway Technology.
« SATURN : SIL2-Certified Fail-safe Remote I/O System Architecture for Trains », 2015.
https ://www.railway-technology.com/contractors/computer/leroy-automation/pressreleases/presssaturn-certified-fail-safe/.

[130] A Terry Bahill et Steven J Henderson.
« Requirements development, verification, and validation exhibited in famous failures ».
Systems engineering, 8(1):1–14, 2005.

[131] Steve Tueno, Marc Frappier, Régine Laleau et Amel Mammar.
« Event-B Sources for the use of the SysML/KAOS Method on the Steam Boiler Controller Specification Case Study », 2018.
https ://github.com/stuenofotso/SysML_KAOS_Domain_Model_Parser/tree/master/-SteamBoilerController.

[132] Steve Tueno, Marc Frappier, Régine Laleau, Amel Mammar et Hector Ruiz Barradas.
« The Generic SysML/KAOS Domain Metamodel ».
ArXiv e-prints, cs.SE, 1811.04732, novembre 2018.

[133] Steve Tueno, Régine Laleau, Amel Mammar et Marc Frappier.
« The SysML/KAOS Domain Modeling Language (Tool and Case Studies) », 2017.
https ://github.com/stuenofotso/SysML_KAOS_Domain_Model_Parser/tree/master.

[134] Steve Tueno, Régine Laleau, Amel Mammar et Marc Frappier.
« SysML/KAOS Approach on the Hybrid ERTMS/ETCS Level 3 case study », 2018.
https ://github.com/stuenofotso/SysML_KAOS_Domain_Model_Parser/tree/master/-ABZ18_ERTMS.

[135] Steve Jeffrey Tueno Fotso, Marc Frappier, Régine Laleau et Amel Mammar.
« Modeling the hybrid ERTMS/ETCS level 3 standard using a formal requirements engineering approach ».
International Journal on Software Tools for Technology Transfer, Oct 2019.

[136] IMCS UL.
« OWLGrEd Home », 2017.
http ://owlgred.lumii.lv/.

[137] Wiebe Van der Hoek et Michael Wooldridge.
« Multi-agent systems ».
Foundations of Artificial Intelligence, 3:887–928, 2008.

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

[139] Axel Van Lamsweerde.
Requirements engineering : From system goals to UML models to software.
volume 10. Chichester, UK : John Wiley & Sons, 2009.

[140] Axel van Lamsweerde, Robert Darimont et Philippe Massonet.
« Goal-directed elaboration of requirements for a meeting scheduler : problems and lessons learnt ».
Dans *Second IEEE International Symposium on Requirements Engineering, March 27 - 29, 1995, York, England, UK*, pages 194–203. IEEE Computer Society, 1995.

[141] Hai H. Wang, Danica Damljanovic et Jing Sun.
« Enhanced Semantic Access to Formal Software Models ».
Dans *Formal Methods and Software Engineering - ICFEM, LNCS*, volume 6447, pages 237–252. Springer, 2010.

[142] Eric S. K. Yu.
« Towards Modeling and Reasoning Support for Early-Phase Requirements Engineering ».
Dans *RE*, pages 226–235. IEEE Computer Society, 1997.

[143] LI Zong-yong, WANG Zhi-xu, ZHANG Ai-hui et Xu Yong.
« The Domain Ontology and Domain Rules Based Requirements Model Checking ».
International Journal of Software Engineering and Its Applications, 1(1):89–100, 2007.