ModelRun, une méthode de transformations de modèles pour la vérification de propriétés de modèles de systèmes complexes par simulation

Christophe Duhil

To cite this version:

Christophe Duhil. ModelRun, une méthode de transformations de modèles pour la vérification de propriétés de modèles de systèmes complexes par simulation. Modélisation et simulation. Université de Bretagne occidentale - Brest, 2021. Français. NNT : 2021BRES0026 . tel-03616711

HAL Id: tel-03616711
https://theses.hal.science/tel-03616711v1

Submitted on 22 Mar 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

DOCTORAT BRETAGNE LOIRE / MATHSTIC

UBO Université de Bretagne Occidentale

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE BRETAGNE OCCIDENTALE

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies
de l'information et de la Communication
Spécialité : Informatique

Par

Christophe DUHIL

ModelRun, une méthode de transformations de modèles pour la vérification de propriétés de modèles de systèmes complexes par simulation.

Thèse présentée et soutenue à Brest , le 07 avril 2021
Unité de recherche : UMR 6285 Lab-STICC

Rapporteurs avant soutenance :

Julien DEANTONI	Maître de conférence, HDR, Université Côte d'Azur
Sébastien GERARD	Directeur de recherche, HDR, CEA List

Composition du Jury :

Président :	Frédéric BOULANGER	Professeur des universités, CentraleSupélec, LMF
Examinateurs :	Isabelle BORNE	Professeur des universités, Université de Bretagne Sud, IRISA
	Julien DEANTONI	Maître de conférences HDR, Polytech Nice, I3S
	Sébastien GERARD	Directeur de recherche, HDR, CEA List
	Jean-Luc VOIRIN	Directeur technique ingénierie, Thales Defense Mission Systems
Dir. de thèse :	Jean-Philippe BABAU	Professeur des universités, Université de Bretagne Occidentale, Lab-STICC

Invités :

Éric LEPICIER Responsable d'Ingénierie Système , Thales Defense Mission Systems

REMERCIEMENTS

Mes premiers remerciements vont à ma famille, **Marina** ma compagne, **Alice** et **Fleur** nos filles. Leur soutien sans faille m'a été précieux pendant ces trois années.

Je tiens à remercier particulièrement **Jean-Philippe Babau** mon directeur de thèse, pour la patience et la pédagogie dont il à fait preuve afin d'amener l'ingénieur que j'étais à acquérir méthodologie et démarche scientifique.

Ce projet n'aurait pas pu voir le jour sans **Jean-Luc Voirin** et **Éric Lépicier**. Je les remercie particulièrement pour l'accueille qu'ils m'ont fait chez Thales, leur disponibilité à mon égard et pour toutes les discutions passionnées que nous avons eu autour d'ARCADIA/Capella et de l'ingénierie système. Votre aide m'a été précieuse tant techniquement qu'humainement.

Je tiens à remercier **Julien Deantoni** et **Sébastien Gérard** pour avoir accepté de rapporter mon travail, ainsi que **Isabelle Borne** et **Frédéric Boulanger** qui complètent le jury de cette thèse.

Je remercie également toute l'équipe du département EMG *(Engeeniering Management Group)* de Thales Brest pour l'accueille au sein du département.

Mes remerciements vont aussi aux thésards du Lab-STICC avec qui j'ai eu le plaisir de travailler et d'échanger pendant ces trois ans, en particulier **Hamza**, **Emilien**, **Arwa** et **Chabba**.

Je tiens également à remercier les enseignants du département informatique, pour l'accueille et les échanges que nous avons pu avoir. Merci de la confiance dont vous avez fait preuve en me confiant des enseignements. Ce fut une expérience très enrichissante.

Enfin un grand merci à tous ceux qui m'ont aidé dans mes travaux pendant ces trois années, notamment **Juan Navas** pour l'aide sur les articles que nous avons publiés et **Emmanuel** pour son aide de relecture de ce manuscrit.

SOMMAIRE

Introduction **7**

I Conception des Systèmes Dirigée par les Modèles 10

1 Modèles et systèmes **11**

1.1 Les systèmes 11

1.1.1 Définition 11

1.1.2 Ingénierie système 12

1.2 Introduction à la notion de modèles 12

1.2.1 Définition 13

1.2.2 Réalité, concept et langage ; Le triangle sémiotique d'Ulmann . . 13

1.2.3 Principes généraux de la modélisation 14

1.3 Ingénierie dirigée par les modèles 15

1.3.1 Définition 16

1.3.2 IDM, la vision du Object Management Group (OMG) 17

1.3.3 Les transformations de modèles 19

1.4 Ingénierie système basée sur les modèles. 24

1.4.1 SysML, Arcadia 24

1.4.2 Arcadia 28

1.5 Exemple d'applications 30

1.6 Conclusion du premier chapitre 32

2 Formalisation des modèles par grammaire de graphes **33**

2.1 Grammaire de graphes 33

2.1.1 Formalisation par grammaire de graphes 33

2.1.2 Graphes et morphismes de graphes 34

2.1.3 Graphes et morphisme de graphes avec héritage, contenance et attributs. 36

2.2 Approche algébrique de la transformation des graphes. 42

2.2.1 Transformation de modèles et grammaire de graphes triples . . . 46
2.2.2 Conclusion sur les grammaires de graphes 54

3 Traçabilité 55
3.1 Traçabilité 55
3.1.1 Définition et Terminologies 55
3.1.2 Modèles de traçabilité 57
3.1.3 La traçabilité dans l'IDM. 59
3.1.4 Conclusion sur la traçabilité 63

II État de l'art 64

4 Transformation et exécution de modèles 65
4.1 Définition d'une sémantique en IDM pour la vérification de modèles . . 65
4.2 Transformation de modèles de système vers un environnement de simulation 68
4.2.1 Transformation directe du modèle vers le domaine de simulation 69
4.2.2 Extension du langage de modélisation suivie d'une transformation de modèles 70
4.2.3 Chaîne de transformations de modèles 73
4.3 Synthèse des approches 74

III Proposition et application 76

5 Méthode de Transformation de Modèle 77
5.1 Introduction a ModelRun 77
5.1.1 Principes généraux 77
5.1.2 Formalisation des étapes de transformation 78
5.1.3 Traçabilité des étapes de transformations 80
5.1.4 Exemple de cas d'application 80
5.1.5 Présentation des étapes de transformation 84
5.2 Les étapes de transformations 84
5.2.1 Sélection 84
5.2.2 Réorganisation 92

5.2.3 Alignement 102
5.2.4 Enrichissement 106
5.2.5 Adaptation 109
5.2.6 Exécution du modèle de simulation 112
5.3 Conclusion du troisième chapitre 115

6 Application 117

6.1 Application à la vérification d'un modèle de drone. 117
6.1.1 Introduction 117
6.1.2 PythaDrone 117
6.1.3 Éléments d'analyse opérationnelle 118
6.1.4 Éléments d'analyse du besoin système 120
6.1.5 Éléments d'architecture logique 120
6.1.6 Éléments d'architecture physique 122
6.1.7 Problématique de cohérence des éléments de modèles entre eux. 128
6.2 Modèle cible de simulation 129
6.2.1 Objectifs de vérification 129
6.2.2 Sémantique des machines d'états et modes 132
6.2.3 Sémantique des *data-flows* 133
6.2.4 Sémantique des scénarios 133
6.2.5 Implémentation de l'outil de simulation 135
6.3 Transformation de modèles Capella, par application de la méthode ModelRun 141
6.3.1 Étape de sélection 142
6.3.2 Étape de réorganisation 145
6.3.3 Étape d'alignement 148
6.3.4 Étapes d'enrichissement et d'adaptation 148
6.4 Résultats de simulation 152
6.4.1 Vérification de la cohérence entre le *data-flow* et les machines d'états et modes. 152
6.4.2 Vérification de propriétés dynamiques dans l'enchaînement des situations 155
6.4.3 Vérification de la faisabilité d'un scénario 157
6.4.4 Bilan des résultats de vérification 160

Conclusion 161

Annexes 165

Bibliography 169

INTRODUCTION

Contexte

Les domaines du transport, de l'aérospatial, de l'énergie, de l'Internet des objets, pour ne citer qu'eux, utilisent de plus en plus d'applications de systèmes complexes comme des véhicules, des drones, des éoliennes ou des satellites. Ces systèmes sont complexes dans le sens où ils intègrent de nombreux composants et technologies hétérogènes. On peut trouver dans ces systèmes des composants mécaniques, électriques, électroniques ainsi que du logiciel. Leur conception requière l'expertise de nombreuses personnes dans des domaines souvent très éloignés. La maîtrise du développement, de tels systèmes est un enjeu industriel majeur, nécessitant l'utilisation de méthodes d'ingénierie système pour leur développement. Pour faciliter l'échange d'informations, l'utilisation de méthodes basées sur les modèles, permet d'abord de recueillir les exigences et les besoins de toutes les parties prenantes. Les besoins et exigences sont ensuite modélisés sous forme de diagrammes généralement exprimés dans des sémantiques hétérogènes. Les modèles sont enfin raffinés pour obtenir des modèles d'architecture, base de la réalisation concrète du système.

Problématique

Dans un contexte industriel, une erreur de modélisation peu impacter gravement la viabilité d'un projet, surtout si elle est détectée dans les dernières phases du développement. Pour limiter ces erreurs, il existe des méthodes telle que "Arcadia" [1] visant à faciliter la conception de systèmes complexes. Cependant alors que les systèmes deviennent de plus en plus conséquents et intègrent des technologies hétérogènes de plus en plus pointues, il devient difficile d'assurer la cohérence des éléments de modèles entre eux, notamment dans l'aspect comportemental du système. Le besoin émerge alors, de pouvoir simuler le comportement du système pour y détecter d'éventuelles incohérences de modélisation et ce le plus tôt possible dans les phases de

conception. Cependant bien que l'ensemble des diagrammes d'une architecture système peut décrire le fonctionnement d'un système, les modèles ne disposent pas d'une sémantique permettant leur exécution.

Contribution

Nous proposons dans ce mémoire une méthode de transformation de modèles dont l'objectif est de transformer un ensemble de diagrammes d'un modèle d'ingénierie système en un modèle exprimé dans un domaine disposant d'une sémantique exécutable. Notre méthode baptisée « ModelRun » est composée de cinq étapes (sélection, réorganisation, alignement, enrichissement, adaptation). En appliquant les cinq étapes de la chaîne de transformation, nous obtenons une sémantique opérationnelle permettant la simulation d'une sélection d'un modèle source réorganisé et adapté auquel nous avons ajouté des informations nécessaires pour la simulation. Dans notre méthode la traçabilité est un point clé pour interpréter les résultats des simulations au regard des concepts du modèle source. Ainsi nous définissons une chaîne de traçabilité entre les concepts sources et les concepts cibles au travers des cinq étapes de la chaîne de transformation.

Structure du mémoire

Ce mémoire de thèse est composé de trois parties principales ; une première partie consacrée à la « conception des systèmes dirigées par les modèles » dans laquelle à travers les articles de la littérature, nous exposons les concepts de la modélisation de l'ingénierie dirigée par les modèles, des grammaires de graphes et de la traçabilité. La seconde partie est consacrée à l'état de l'art des méthodes permettant de simuler un modèle. Dans la troisième partie « Contribution et Application », nous proposons « ModelRun » notre méthode de transformation de modèles, puis nous l'appliquons à la transformation à des fins de simulation d'un modèle de drone.

Nous détaillons ci-dessous le contenue de chaque partie :

La première partie , **« Conception des systèmes dirigée par les modèles »** est constitué de trois chapitres :

— **Chapitre 1,« Modèle et systèmes »** : Dans ce chapitre nous présentons les

notions générales liées aux systèmes et modèles. Nous présentons ensuite l'ingénierie dirigée par les modèles ainsi que l'ingénierie système basée sur les modèles.

— **Chapitre 2,« Formalisation des modèles par grammaire de graphes »** : Dans un premier temps, nous présentons la définition de graphes, en partant de la définition d'un graphe simple constitué uniquement de nœuds et d'arcs, pour aboutir à la définition de graphes attribués avec héritage et lien de contenance. Puis nous présentons une approche algébrique de transformation de graphes. Notion que nous étendons à la transformation de modèles avec des grammaires de graphes triples. Ces notions sont la base de la formalisation de notre méthode de transformation de modèles présentée en seconde partie.

— **Chapitre 3,« Traçabilité »** : Nous présentons dans ce chapitre les définitions et les terminologies utilisées dans le domaine de la traçabilité. Nous décrivons quelques modèles de traçabilité puis nous montrons comment la traçabilité est utilisée en ingénierie dirigée par les modèles.

La seconde partie **« État de l'art »** est constituée d'un chapitre :

— **Chapitre 4 « Transformation et exécution de modèles »** Dans ce chapitre nous faisons un tour d'horizon de la littérature consacrée aux méthodes permettant de simuler un modèle que ce soit en étendant son langage de modélisation ou en le transformant.

La troisième partie « Contribution et Application » est constituée de deux chapitres :

— **Chapitre 5 « Méthode de transformation de modèles »** Nous présentons et formalisons ici les cinq étapes de « ModelRun » notre méthode de transformation de modèle.

— **Chapitre 6 « Application »** Nous appliquons alors « ModelRun » à la vérification de propriétés d'un modèle de système complexe de drone.

Première partie

Conception des Systèmes Dirigée par les Modèles

CHAPITRE 1

MODÈLES ET SYSTÈMES

Dans les années 1960, deux organismes américains, la NASA et l'United States Air Force (USAF) ont entrepris une démarche de rationalisation du développement des programmes militaires et d'exploration spatiale. Cette démarche a permis l'émergence de l'ingénierie système. Suite à ces efforts est créé en 1991 l'International Council on Systems Engineering (INCOSE) [2], dont la mission est la promotion de l'ingénierie système, de son développement et de son usage auprès des ingénieurs systèmes. En 2007, l'INCOSE promeut l'utilisation des modèles dans l'ingénierie système. Dans ce chapitre consacré aux modèles et aux systèmes, nous reprenons dans un premier temps les définitions de la notion de modèle tel que proposées par la communauté scientifique dédiée aux sciences de l'information. Nous ferons ensuite le lien entre la réalité d'un objet, sa conceptualisation et le langage de modélisation permettant de le représenter. Nous verrons alors comment les modèles sont utilisés dans l'ingénierie logicielle à travers l'Ingénierie Dirigée par les Modèles (IDM) ainsi que dans l'ingénierie des systèmes complexes à travers l'Ingénierie des Systèmes Basée sur les Modèles (ISBM).

1.1 Les systèmes

Par système, on entend un ensemble d'éléments humains ou matériels en interdépendance les uns avec les autres et avec leur environnement. Les éléments matériels peuvent être composés de sous-ensembles de technologies variées : mécanique, électrique, électronique, matériels informatiques, logiciels, réseaux de communication, etc.

1.1.1 Définition

L'INCOSE donne la définition générale suivante d'un système [3] :

> « A **system** is an arrangement of parts or elements that together exhibit behaviour or meaning that the individual constituents do not. »

Pour l'INCOSE les propriétés d'un système résultent :

- des éléments du système et de leurs propriétés individuelles,
- des relations et interactions entre les éléments du système,
- des relations entre le système et son environnement.

1.1.2 Ingénierie système

L'ingénierie système propose une approche scientifique interdisciplinaire dans le but de formaliser et d'appréhender la conception et la validation de systèmes complexes. L'ingénierie système a pour objectif de maîtriser et de contrôler la conception de systèmes dont la complexité et l'hétérogénéité ne permet pas une conception simple. L'INCOSE propose la définition suivante de l'ingénierie système [3] :

> « **Systems Engineering** is a transdisciplinary and integrative approach to enable the successful realization, use, and retirement of engineered systems, using systems principles and concepts, and scientific, technological, and management methods. »

Ainsi pour l'INCOSE, l'ingénierie système prend en compte à la fois les besoins commerciaux et techniques des clients dans le but de leur fournir une solution de qualité qui répond aux besoins des utilisateurs et des autres parties prenantes.

1.2 Introduction à la notion de modèles

Les modèles jouent un rôle central dans la représentation et la conception de systèmes. De la maquette, des plans sur papier, jusqu'aux équations formelles, les modèles permettent de raisonner et de communiquer sur des phénomènes ou des systèmes que l'on ne sait pas théoriser ou bien trop complexes pour être appréhendés directement par l'esprit humain [4] . Les modèles peuvent refléter des phénomènes ou des systèmes existants : ils sont alors **descriptifs**. Ce peut être par exemple la modélisation d'un phénomène physique ou naturel. Cela peut aussi être la modélisation d'un domaine de connaissance. On parle alors d'ontologie. Les modèles peuvent aussi être

utilisés dans l'élaboration d'un système qui n'a pas encore d'existence réelle : le modèle est alors **prescriptif**. Dans ce cas, le modèle est généralement l'expression d'un ensemble d'exigences décrivant ce que sera et ce que devra réaliser le futur système.

1.2.1 Définition

Franck Varenne reprend dans [4] la définition proposée pour la première fois par le modélisateur et informaticien Marvin Minsky en 1965 :

> « Pour un observateur B, un objet A* est un modèle d'un objet A dans la mesure où B peut utiliser A* pour répondre à des questions qui l'intéresse au sujet de A. »

Dans la communauté scientifique dédiée aux sciences de l'information, de nombreux auteurs ont proposés une définition de ce qu'est un modèle. Parmi toutes ces définitions nous pouvons retenir les définitions complémentaires proposées par Brian Henderson-Sellers et Jean Bézivin. Ainsi pour Brian Henderson-Sellers [5] :

> « A model is an abstraction of reality according to a certain conceptualisation. »

Pour Bézivin et Gerbé [6] :

> « A model is a simplification of a system built with an intended goal in mind. The model should be able to answer questions in place of the actual system. »

Un ensemble d'auteurs (Stachowiak, Ludewig, Khune, Gonzalez, Henderson-Sellers) font la suggestion que les modèles doivent suivre trois critères [5] énoncés ainsi par [7] :

Mapping feature : A model is based on an original.

Reduction feature : A model only reflects a relevant selection of the original's properties.

Pragmatic feature : A model needs to be usable in place of the original with respect to some purpose.

1.2.2 Réalité, concept et langage ; Le triangle sémiotique d'Ulmann

Giancarlo Guizzardi établit dans [8] le lien entre la réalité d'un objet, la conceptualisation de cet objet et le besoin d'utiliser ou d'établir un langage capable de représenter

l'objet conceptualisé. Pour ce faire, l'auteur utilise le triangle d'Ulmann [9] (triangle lui même dérivé des travaux des linguistes Ogden et Richards [10] et de Ferdinand de Saussure [11]) (figure 1.1)

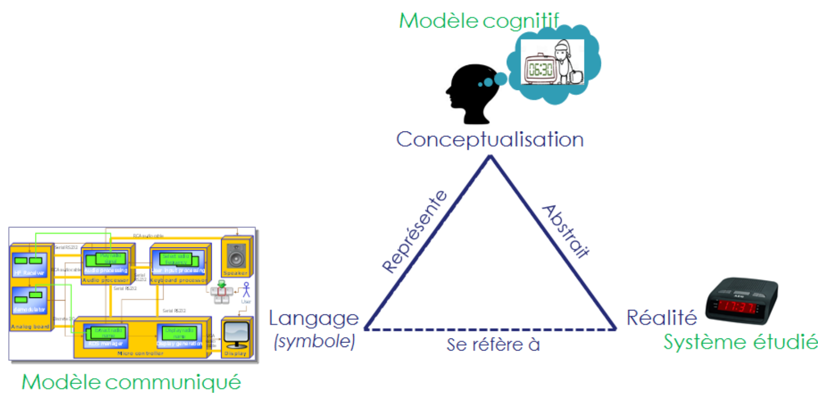


FIGURE 1.1 – Triangle sémiotique d'Ulmann

L'abstraction et la conceptualisation d'une partie de la réalité d'un système sont produites par la conscience de l'utilisateur (relation « *Abstrait* » du triangle figure 1.1). Pour être analysés, documentés et communiqués, les concepts doivent être représentés sous la forme d'artefacts concrets, ce qui implique la définition ou l'utilisation d'un langage de modélisation spécifique au domaine conceptualisé. (relation « *Représente* » du triangle figure 1.1). Ce langage peut être grammatical ou bien composé de symboles (langage diagrammatique). La relation *« se réfère à »* entre le langage de modélisation et la réalité est alors le fruit d'une médiation induite par une conceptualisation de cette réalité. Franck Varenne fait le constat suivant dans [12]

> « un modèle n'est pas indépendant d'un utilisateur, d'un observateur. Le modèle d'un objet n'est modèle de cet objet que pour un observateur particulier, avec son point de vue et ses problématiques. »

1.2.3 Principes généraux de la modélisation

La définition d'un langage de modélisation spécifique pour un domaine est un élément central de l'ingénierie dirigée par les modèles (IDM). L'élaboration d'un (ou de plusieurs) méta-modèle permet de définir les concepts d'un domaine d'intérêt spécifique ainsi que les relations qui unissent les concepts entre eux. C'est la définition d'une syntaxe abstraite [13] . Le méta-modèle définit le langage de modélisation spécifique au domaine [14]. La figure 1.2 montre un exemple simple de méta-modèle.

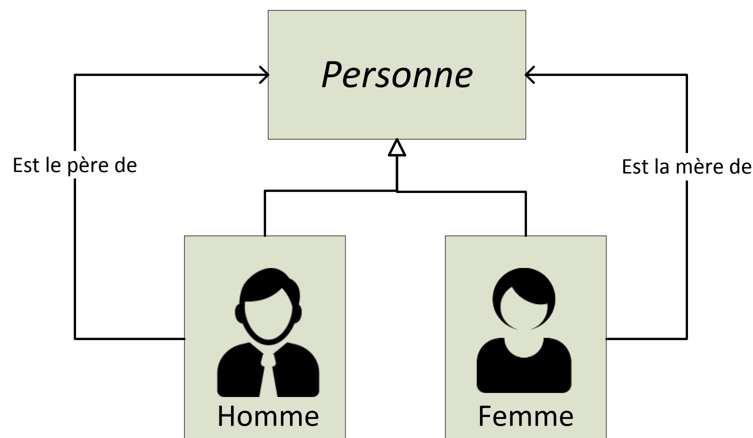


FIGURE 1.2 – Exemple d'un méta-modèle simple

Le méta-modèle est constitué de trois concepts *Personne*, *Homme* et *Femme*. Le concept *Personne* est un concept abstrait, il généralise les concepts *Homme* et *Femme*. Les concepts sont reliés par deux relations : "est le père de" et "est la mère de", dont la cardinalité est 0 à ∞. Ainsi, un *Homme* peut être (ou pas) le père d'une ou plusieurs *Personne*, (*Homme* ou *Femme*) ; une *Femme* peut être la mère (ou pas) d'une ou plusieurs *Personne* (*Homme* ou *Femme*). Ce méta-modèle définit une syntaxe abstraite, permettant de modéliser un arbre généalogique.

La réalisation d'un modèle concret conforme au méta-modèle constitue une **instance** de ce dernier [15]. Le diagramme de la figure 1.3 montre une instance du méta-modèle d'arbre généalogique présenté en figure 1.2 .

1.3 Ingénierie dirigée par les modèles

Poussés par la complexification croissante des applications logicielles, les concepteurs de solutions logicielles ont intégré les modèles dans toutes les phases du cycle de vie des systèmes, depuis les phases d'expression des besoins jusqu'aux phases d'exploitation et de maintenance. Entre autre apport de l'utilisation des modèles comme élément central de l'ingénierie, les modèles permettent de gérer la complexité de l'ensemble d'un système en augmentant le niveau d'abstraction de sa représentation. Ils permettent aussi de communiquer entre toutes les parties prenantes d'un projet.

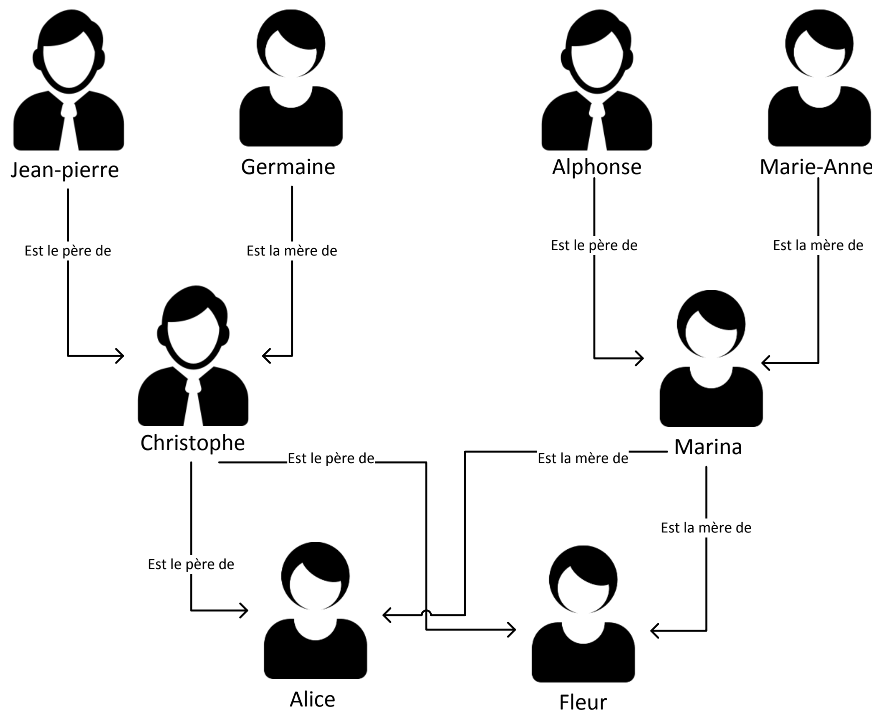


FIGURE 1.3 – Instance de modèle généalogique

1.3.1 Définition

Pour Jean Bézivin le principe de base de l'Ingénierie Dirigée par les Modèles (IDM) est que « tout est modèle » [16]. Robert France et Bernhard Rumpe proposent cependant une définition plus précise de l'IDM [17] :

> « The term Model-Driven Engineering (MDE) is typically used to describe software development approaches in which abstract models of software systems are created and systematically transformed to concrete implementations. »

Ainsi pour Robert France et Bernhard Rumpe le MDE est une approche dans laquelle, un système logiciel est développé sous la forme de modèles abstraits puis systématiquement transformés en implémentations concrètes.

1.3.2 IDM, la vision du Object Management Group (OMG)

Objectif du MDA

En Novembre 2000, l'OMG propose sa vision de l'IDM à travers le MDA^{TM} (Model Driven Architecture). L'objectif du MDA est d'unifier chaque étape du développement d'une application. Depuis son origine par la définition de modèles indépendants de toute plateforme cible (Plateforme Independant Model PIM), jusqu'à la définition d'un ou plusieurs modèles pour des plateformes spécifiques (Platform Specific Model, PSM), afin de générer le code d'une application (figure 1.4). Pour ce faire, le MDA propose un ensemble de normes, telles que :

- Le MOF^{TM}(Meta Object Facility), un langage abstrait permettant de définir et manipuler les méta-modèles,
- Le langage de modélisation UML^{TM} (Unified Modeling Langage), un langage de modélisation généraliste,
- Le CWM^{TM}(Common Warehouse Meta-model), un langage d'échange de méta-données à travers un entrepôt de données, un système décisionnel, un système d'ingénierie des connaissances,
- Le format d'échange de méta-données XMI^{TM} (XML Metadata Interchange), un format d'échange de méta-données basé sur XML, permettant notamment la sérialisation des modèles.

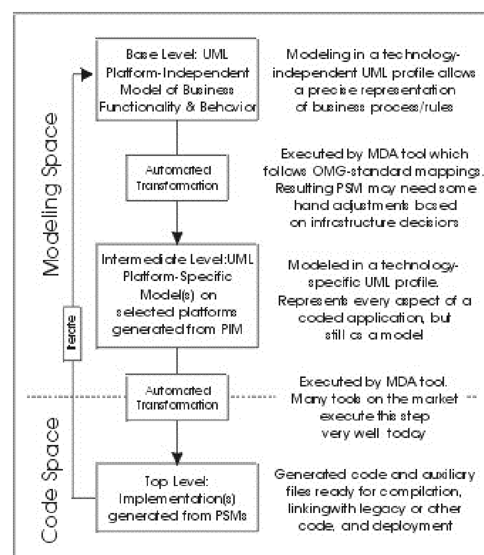


FIGURE 1.4 – Diagramme MDA

Meta-Modélisation

L'OMG a défini une infrastructure de méta-modélisation hiérarchisée en 4 niveaux d'abstraction (figure 1.5). Chaque niveau (sauf le niveau M3) est une instance du niveau supérieur ainsi :

- Le niveau M0 correspond au monde réel. Il contient les données réelles des utilisateurs de l'application.
- Le niveau M1 contient le modèle des concepts de l'application. Ce peut être par exemple un modèle UML.
- Le niveau M2 modélise les informations du niveau M1. Comme c'est un modèle de modèle il est souvent appelé méta-modèle.Il définit le langage de modélisation du niveau M1. On trouve à ce niveau par exemple la définition des langages UML ou ECore.
- Le niveau M3 est le méta-méta-modèle il définit le langage de spécification des méta-modèles M2. l'OMG a défini le Meta Object Facility (MOF) pour la spécification des langages de modélisation.

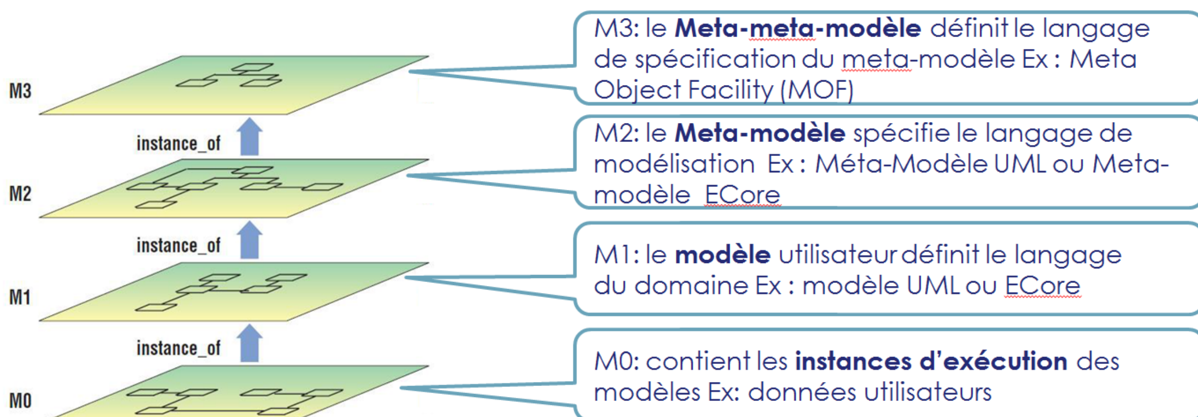


FIGURE 1.5 – Infrastructure de modélisation suivant l'OMG

Il est à noter que la seconde version du standard MOF spécifie que le nombre de niveaux de modélisation est flexible sur la base d'au moins deux niveaux.

Limite de la vision de l'OMG

Pour un certain nombres de chercheurs dont Colin Atkinson et Thomas Kühne [18], le fait que les relations entre concepts de différents niveaux soient uniquement des relations d'instances posent un certain nombre de problèmes. Ainsi, La vision de l'OMG

privilégie une dimension linguistique dans la modélisation d'un domaine au détriment de sa dimension ontologique. Ces auteurs identifient les limites suivantes à la vision de l'OMG :

- **La dualité classe-objet** : dans la vision de l'OMG, un artefact est soit une classe soit un objet, les deux concepts étant liés par une relation d'instantiation [19][20].
- **La réplication des concepts** : l'impossibilité du mécanisme d'instanciation de transmettre une information relative aux attributs et aux relations à travers plus d'un niveau d'instanciation, impose de dupliquer les informations sur des niveaux multiples [21].
- « **Deep classification** » : l'impossibilité d'instancier un artefact sur plusieurs niveaux [19][20].

Malgré ces limitations l'approche proposée par l'OMG, est massivement adoptée dans l'industrie. Nous suivons cette approche dans notre thèse.

1.3.3 Les transformations de modèles

Une démarche de l'ingénierie dirigée par les modèles implique généralement de représenter un système sous une multitude de modèles, avec différents points de vue, à différents niveaux d'abstraction. Cela est nécessaire pour pouvoir représenter un système complexe à travers plusieurs modèles représentant chacun un aspect d'une solution. Shane sendall et Wojtek Kozacznski notent dans [22], que travailler avec plusieurs modèles interdépendants nécessite un effort important pour garantir leur cohérence globale. Outre la synchronisation des modèles, il est possible de réduire considérablement cet effort, grâce à l'automatisation. Un processus automatisé prend un ou plusieurs modèles sources en entrée et produit un ou plusieurs modèles cibles en sortie, tout en suivant un ensemble de règles de transformation. Ces règles formalisent les liens entre les modèles d'entrées et les modèles de sortie. Ce processus est appelé transformation de modèles.

Kleppe et al. proposent dans [23] *(repris par [24])*, une définition de la transformation de modèles ainsi que la définition d'une règle de transformation :

> « A **transformation** is the automatic generation of a target model from a source model, according to a transformation definition. »
>
> « A **transformation definition** is a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language. »

> « A **transformation rule** is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language. »

Ces définitions peuvent être illustrées par le diagramme issu de [25] figure 1.6

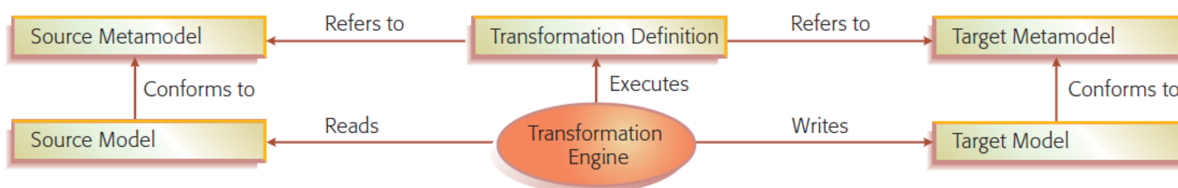


FIGURE 1.6 – Concept de base de la transformation de modèles issue de [25]

K. Czarnecki et S. Helsencette présentent dans cette figure un aperçu des principaux concepts impliqués dans une transformation du modèle. La transformation implique un modèle d'entrée (source) *"Source Model"* et un modèle de sortie (cible) *"Target Model"*. Les deux modèles sont conformes à leurs méta-modèles respectifs. Les régles de transformation sont définies par rapport aux méta-modèles *"Transformation Definition"*. La définition est exécutée sur des modèles concrets par un moteur de transformation *"Transformation Engine"*. Ce moteur de transformation exécute les règles de transformations pour écrire le modèle cible en fonction du modèle source.

Classification des transformations de modèles

Plusieurs auteurs ont proposé une classification des transformations de modèles. [24] [25][26] [27]. Dans cette thèse, nous proposons de nous appuyer sur la taxonomie des transformations de modèles proposée par Tom Mens et Pieter Van Gorp dans [24], ainsi que sur la classification des transformations de modèles en fonction des intentions de la transformation et de leurs propriétés, proposée par Levi Lùcio et al. [27].

Dans [24], Tom Mens et Pieter Van Gorp proposent un système de classement des transformations de modèles suivant deux dimensions orthogonales (cf figure 1.7) :

- Endogène versus exogène,
- Horizontal versus vertical.

Une transformation *endogène* est une transformation pour laquelle les langages des modèles source et cible sont identiques. Par opposition, une transformation *exo-*

gène, transforme un modèle source en un modèle cible exprimé dans un langage différent.

En fonction du niveau d'abstraction dans lesquels les modèles sont définis, les auteurs font la distinction entre transformation *horizontale* et *verticale*. Une transformation *horizontale* est une transformation pour laquelle les modèles cible et source sont exprimés à un même niveau d'abstraction. Une transformation verticale est une transformation impliquant des modèles source et cible exprimés à différents niveaux d'abstraction.

	horizontal	vertical
endogenous	*Refactoring*	*Formal refinement*
exogenous	*Language migration*	*Code generation*

FIGURE 1.7 – Dimensions orthogonales des transformations de modèles avec exemples, issue de [24].

Si nous considérons une opération de *refactoring* comme une transformation dans laquelle les concepts d'un modèle source sont réarrangés pour améliorer la qualité de certaines caractéristiques d'une application, alors cette transformation est une transformation *endogène* : les langages des modèles sources et cibles restent les mêmes. La transformation est *horizontale*, puisqu'elle intervient à un même niveau d'abstraction. A l'opposé, la génération de code est considérée par la taxonomie de Tom Mens et Pieter Van Gorp comme une transformation *exogène* (les langages des modèles source et cible sont différents) et *verticale* (les modèles source et cible sont exprimés à différents niveaux d'abstraction).

Levi Lùcio et al. proposent dans "Model transformation intents and their properties" [27] une classification des transformations de modèles en fonction de leurs intentions et de leurs propriétés. Ils définissent ainsi un catalogue composé de neuf catégories et de vingt-trois sous-catégories de transformations de modèles (figure 1.8)

Refinement : le raffinement regroupe les transformations de modèles qui produisent un modèle plus précis en réduisant les ambiguïtés sémantiques. Un raffinement produit des spécifications de plus bas niveau à partir de spécifications de haut

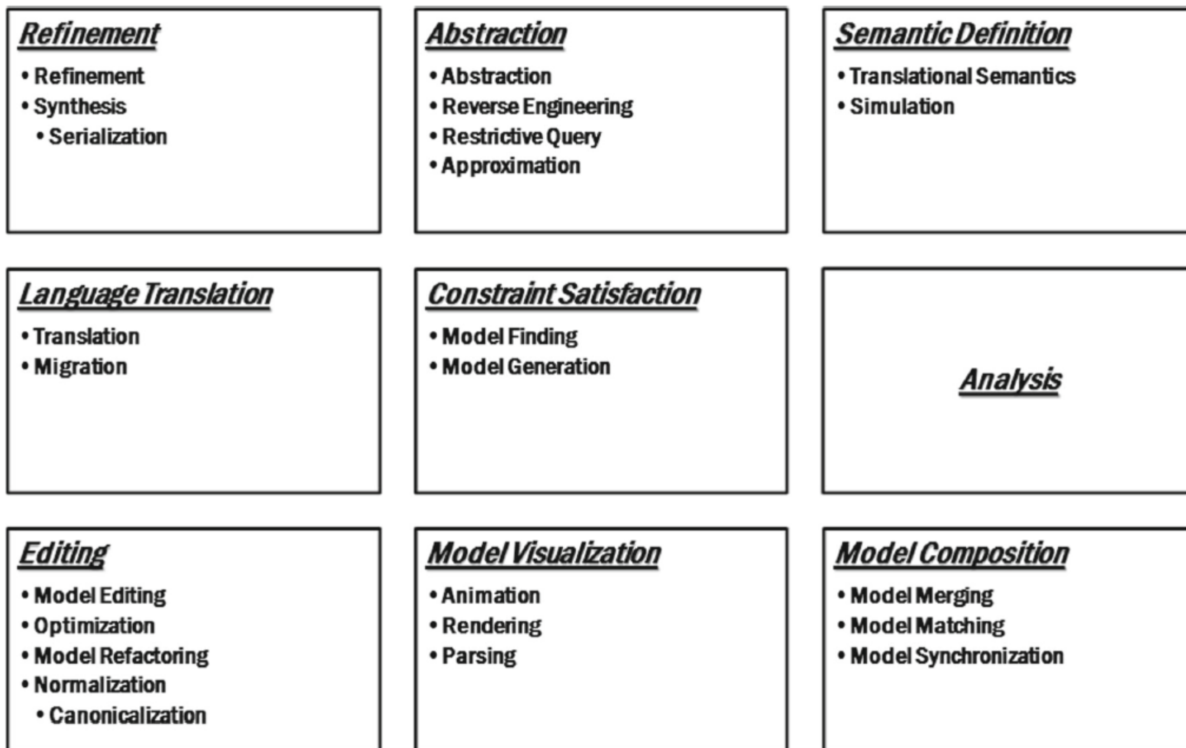


FIGURE 1.8 – Classification des intentions de transformations de modèles proposée par [27]

niveau. Le raffinement ajoute des informations au modèle raffiné. Le modèle source est alors un sous-type du modèle raffiné au sens de [28] et de [29]. Ainsi si un modèle m_1 raffine un modèle m_2, alors il contient à minima toutes les informations contenues dans m_2. Pour les auteurs, la synthèse est un cas spécifique de raffinement dans lequel un modèle source est transformé en un code exécutable.

Abstraction : l'abstraction est la transformation inverse du raffinement. Ainsi le modèle abstrait est un sous-type du modèle source au sens de [28] et de [29]. Les applications de l'opération d'abstraction peuvent être trouvées dans le *"Reverse Engineering"* lorsqu'il est utilisé comme l'opération inverse de la synthèse. Le résultat d'une requête sur un modèle complexe permet d'extraire une partie des informations de ce modèle sous la forme d'un sous-modèle.

Semantic definition : la définition sémantique permet de préciser la sémantique d'un langage de modélisation, par exemple pour des besoins de simulation.

Langage translation : la catégorie des translations de langages regroupe les transformations qui définissent une translation entre deux langages de modélisation. Ainsi la translation permet de définir les concepts définis dans un langage source en un autre langage d'un domaine cible en traduisant la sémantique du domaine source, dans les termes du domaine cible. Par exemple lorsqu'on transforme un modèle de classes en un modèle de base de données relationnelles. La migration d'un modèle exprimé dans une première version d'un langage vers une seconde version de ce langage est un autre exemple de translation de langage.

Constraint satisfaction : l'ensemble des transformations de satisfaction de contraintes regroupe les transformations pour lesquelles le modèle cible satisfait à un ensemble de contraintes. C'est le cas de la génération automatique de modèles. Une génération automatique de modèles permet de créer un ensemble d'instances conforme au langage d'un méta-modèle cible.

Analysis : les auteurs considèrent dans les intentions de cette catégorie toutes les transformations de modèle qui ont pour finalité une analyse du modèle source tel que par exemple la détection de "dead lock", via la transformation de modèles vers des modèles utilisables par des techniques de "model checking".

Editing : dans cette catégorie sont regroupées toutes les transformations endogènes de manipulation de modèles, tel que l'ajout ou la suppression d'éléments du modèle. Entrent dans cette catégorie des opérations plus complexes d'optimisation, de refactoring, ou de normalisation de modèles.

Model visualization : l'intention des transformations de modèle à des fins de visualisation est de transformer un modèle exprimé dans une syntaxe abstraite vers un modèle possédant une syntaxe permettant une représentation visuelle des concepts du modèle source. L'animation est un exemple permettant la visualisation de simulation de modèles.

Model composition : cette dernière catégorie regroupe les intentions de transformation de modèles liées aux relations de composition entre deux ou plusieurs modèles. La fusion de modèles, la recherche de correspondances entre modèles et la synchronisation de modèles font partie de cette catégorie.

La problématique exposée dans la thèse, Implique des transformations de modèles. Nous devons transformer un ensemble de modèles d'ingénierie système, vers des mo-

dèles exprimés dans des langages dotés d'une sémantique exécutable. L'intention générale de la chaine de transformation de modèles que nous exposons dans la seconde partie de ce manuscrit répond à l'intentions générale de*« analysis »* tel que proposé par Levi Lùcio et al. puisque le but est de détecter de potentiels incohérences entre les modèles d'ingénierie sources. Cependant à l'intérieur de la chaîne de transformation les transformations unitaires pourront répondre à des intentions telles que le *« langage translation »* permettant de définir les concepts de nos modèles sources en un langage conforme au méta-modèle cible. Il faudra raffiner le modèle de départ pour lever les ambiguïtés sémantiques : intention de *« refinement »*. Nous aurons besoin de préciser la sémantique de certains modèles pour les rendre exécutables, intention de *« semantic definition »*.

1.4 Ingénierie système basée sur les modèles.

L'IDM est un concept généralement utilisé en génie logiciel. Pour les systèmes complexes intégrant une multitude de technologies, L'INCOSE définit une méthodologie d'ingénierie système basée sur les modèles (en anglais *Model Based System Engineering, MBSE*). L'INCOSE définissait en 2017 sa vision du MBSE pour 2020 [30] ainsi :

> « Model-based systems engineering (MBSE) is the formalized application of modeling to support system requirements, design, analysis, verification and validation activities beginning in the conceptual design phase and continuing throughout development and later life cycle phases. »

1.4.1 SysML, Arcadia

SysML

SysML[31] est un langage de modélisation graphique développé par l'OMG, INCOSE et AP-233 [32]. SysML fournit aux ingénieurs système un langage de modélisation leur permettant de :

- spécifier les systèmes,
- analyser la structure et le fonctionnement des systèmes,
- décrire les systèmes et concevoir des systèmes composés de sous systèmes,
- vérifier et valider la faisabilité d'un système avant sa réalisation.

A cette fin, SysML permet de représenter les composants physiques de toutes technologies, des systèmes logiciels, des énergies, des personnes ainsi que des procédures et flux divers.

SysML a été implémenté sous la forme d'un profil UML 2.0. Il est le fruit de la généralisation des concepts utilisés en UML enrichis de quelques notions. (diagramme figure 1.9).

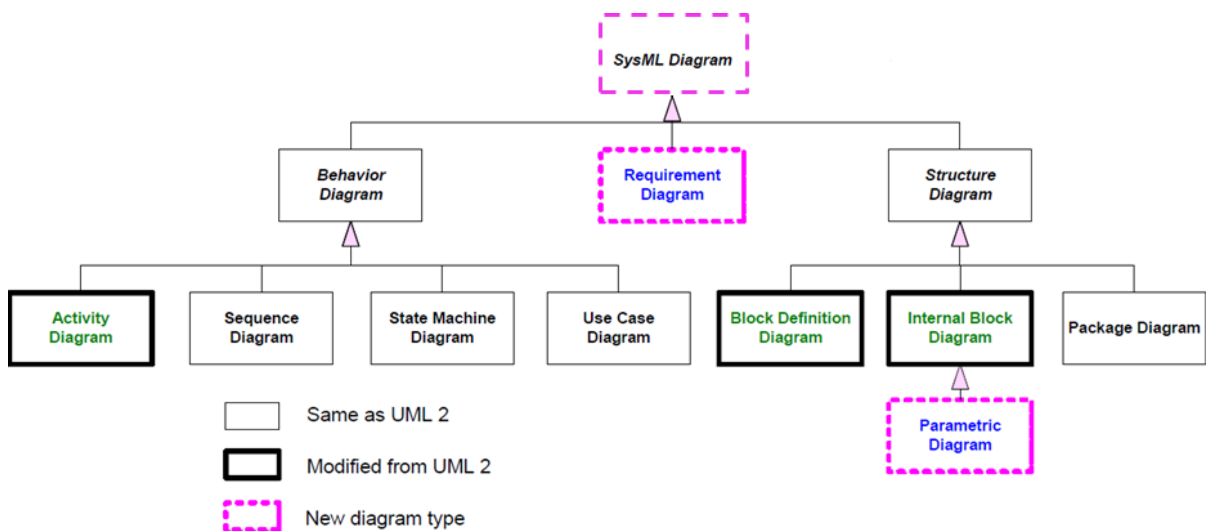


FIGURE 1.9 – Taxonomie des diagrames OMG SysML issue de [33].

Le langage SysML est structuré autour de 9 diagrammes permettant de spécifier : les exigences, le comportement du système et sa structure [34] (cf figure 1.10).

Les exigences sont modélisées dans le « **Requirements Diagram** ». Elle permettent de collecter et d'organiser toutes les exigences textuelles du système. La figure 1.11 présente un exemple de « **Requirements Diagram** ».

Les représentations de la structure du système Un concept important de SysML est le concept de «**bloc**» qui étend le concept de classe d' UML. Les blocs peuvent représenter n'importe quel niveau de la hiérarchie du système, y compris le niveau supérieur système, un sous-système ou un composant logique ou physique d'un système ou d'un environnement. Les blocks et leurs relations sont décrits dans deux diagrammes : le « **Block DefinitionDiagram** » et (BDD) et le « **Internal Block Diagram** »

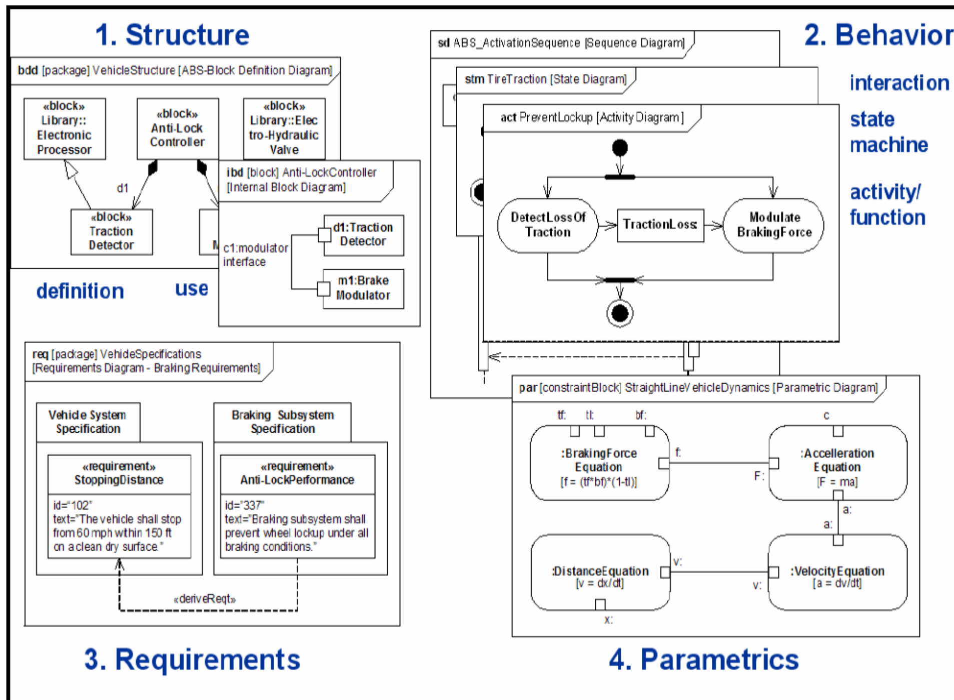


FIGURE 1.10 – Structuration des diagrammes de l'OMG SysML issue de [34].

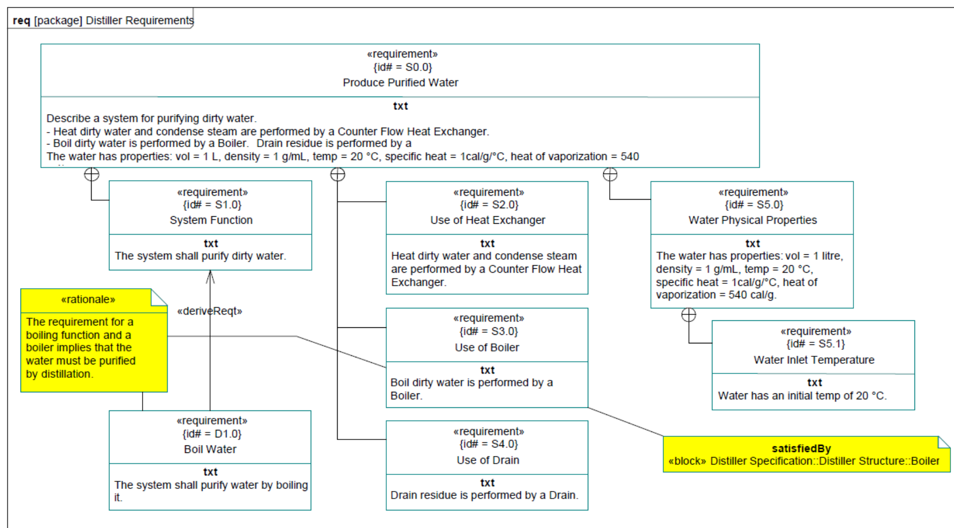


FIGURE 1.11 – Requirements Diagram issue de [34].

(IDB). La définition de la structure du système est complétée par deux autres diagrammes le « **Package Diagram** » et le « **Parametric Diagram** ».

- **Block Definition Diagram** : les diagrammes de définition de Blocs en SysML

sont semblables au diagramme de classe en UML. Ils donnent une représentation statique des entités du système, de leurs propriétés, de leurs opérations et de leurs relations. La figure 1.12 donne un exemple de BDD.

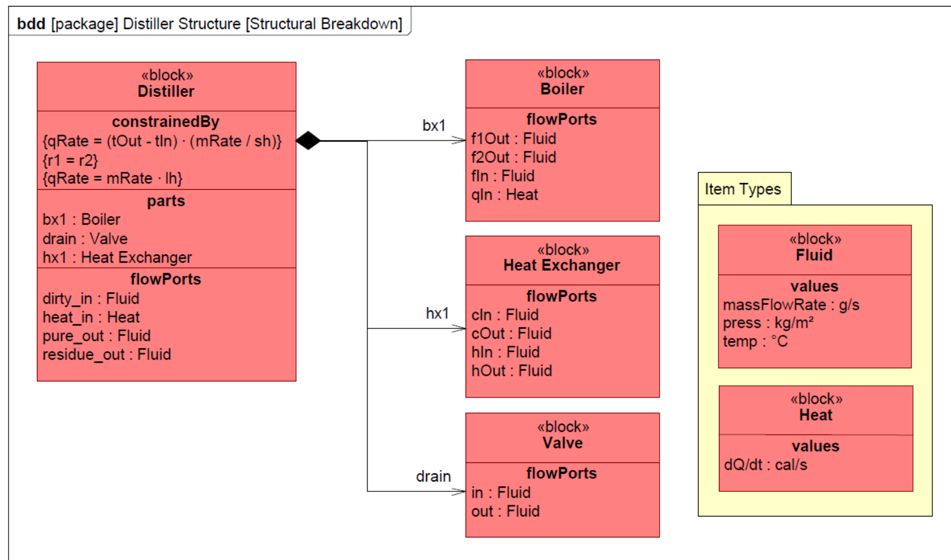


FIGURE 1.12 – Block Definition Diagram issu de [34].

- **Internal Block Diagram** : ils donnent une représentation de la structure interne des Blocks.la figure 1.13 montre un exemple de IBD.

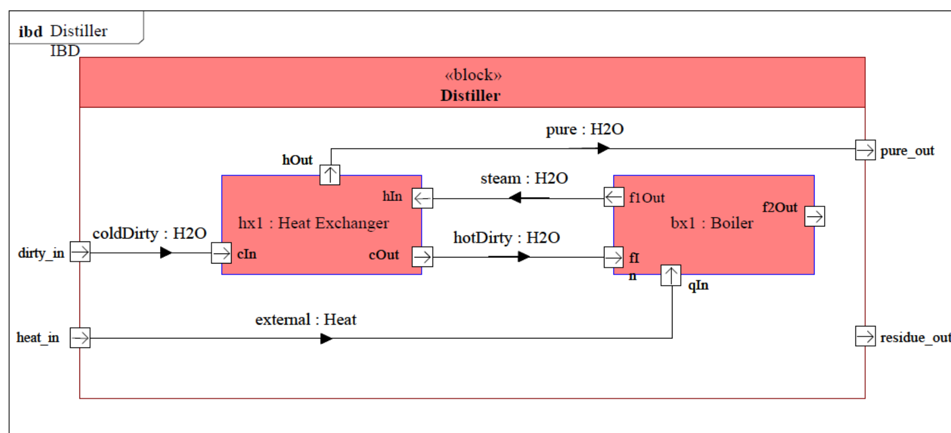


FIGURE 1.13 – Internal Block Diagram issu de [34].

- **Package Diagram** : ils montrent l'organisation générale des différentes vues du système.

- **Parametric Diagram** : Les diagrammes paramétriques sont utilisés pour décrire les contraintes sur les propriétés du système. Ces contraintes sont décrites sous forme d'équation mathématique.

Les représentations comportementales du Système Le comportement du système est modélisé en SysML à l'aide de quatre diagrammes : les « **Use Case Diagram** », les « **Sequence Diagram** », les « **Activity Diagram** », ainsi que les « **State Machine Diagram** ».

- **Use Case Diagram** : ils modélisent les fonctionnalités que le système doit fournir.
- **Sequence Diagram** : les diagrammes de séquence modélisent la chronologie des interactions entre les éléments du système et/ou entre le système et l'extérieur.
- **Activity Diagram** : les diagrammes d'activité modélisent les flux d'informations et les flux d'activité du système.
- **State Machine Diagram** : ils représentent le comportement du système modélisé à travers un ensemble d'états et de transitions.

Le langage SysML est implémenté par plusieurs outils d'ingénierie système, [35], parmi lesquels Magic Draw [36] , Modelio SysML Architect [37], Rhapsody [38] , Papyrus [39].

Bien que très largement adopté en ingénierie système, SysML n'est pas le seul langage de modélisation par exemple Arcadia *(ARChitecture Analysis and Design Integrated Approach)* [1] est une méthode et un langage d'ingénierie système basée sur les modèles.

1.4.2 Arcadia

Arcadia a été développée par Thales dans la volonté d'instaurer une ingénierie système basée sur les modèles pour toutes ses activités d'ingénierie. [40] ARCADIA, permet de construire l'architecture de systèmes complexes. Elle favorise le travail collaboratif de toutes les parties prenantes d'un projet de conception d'un système. Arcadia est structurée autour de cinq perspectives, partant de l'analyse du besoin pour aller vers une définition architecturale de la solution (cf figure 1.14). Les deux premières perspectives, l'analyse opérationnelle et l'analyse du besoin système, participent à l'analyse et la compréhension du besoin. Les trois autres perspectives, l'architecture

logique, l'architecture physique et la stratégie de construction du produit, participent à la définition architecturale de la solution.

L'analyse Opérationnelle

« Ce que les utilisateurs doivent accomplir. »

Cette perspective analyse la problématique du point de vue des utilisateurs opérationnels. Elle participe à identifier *les acteurs* devant interagir avec le système, leurs *buts*, leurs *activités*, leurs *contraintes* ainsi que les conditions d'interactions entre eux.

L'analyse du besoin système

« Ce que le système doit réaliser pour ses utilisateurs »

Cette perspective permet une analyse fonctionnelle. Elle est construite à partir de l'analyse opérationnelle et des *exigences* du client. L'analyse de besoin système permet alors de modéliser les *fonctions* et les *services* du systèmes nécessaires à ses utilisateurs.

L'architecture logique

« Comment le système va fonctionner pour répondre aux attentes. »

En fonction des besoins recueillis par l'analyse des besoins systèmes, l'architecture logique porte les premiers grands choix de conception de la solution. Elle se construit via une analyse fonctionnelle interne du système et par l'identification des *composants* de principe de la solution.

L'architecture physique

« Comment le système va être construit. »

Cette perspective représente l'architecture du système tel qu'il doit être réalisé et intégré. Elle ajoute les fonctions requises par l'implémentation et les choix techniques. Elle fait apparaître des composants comportementaux qui réalisent ces fonctions. Ces composants comportementaux sont ensuite déployés sur des composants d'implémentation qui leur fournissent les ressources nécessaires.

La stratégie de construction du produit

« Ce qui est attendu de chaque composant et les conditions de son intégration dans le système. »

Cette perspective déduit de l'architecture physique les conditions que doit remplir chaque composant pour satisfaire aux contraintes et choix de l'architecture du système. Elle définit aussi la stratégie d'intégration, vérification, validation du système dans son ensemble.

Vue d'ensemble des principaux concepts

Différentes vues cohabitent au sein de chaque perspective (figure 1.14) :

- vue fonctionnelle (activités, fonctions),
- vues qui décrivent le comportement (chaînes fonctionnelles, scénarios, modes et états)
- vue structurelle comportementale (composants comportementaux),
- vue structurelle d'implémentation (composants physiques hôtes),
- représentation des éléments échangés (data),
- point de vue d'analyse de spécialité (viewpoints).

Les différentes vues et perspectives sont modélisées à l'aide d'un ensemble de diagrammes dont Arcadia définit la syntaxe et la sémantique. Cet ensemble forme le langage d'Arcadia. La méthode et le langage Arcadia sont implémentés dans l'outil Capella [41].

1.5 Exemple d'applications

L'ingénierie système basée sur les modèles est aujourd'hui déployée dans de nombreuses industries. Les langages de modélisation sont alors souvent associés à des méthodes d'ingénierie systèmes et des outils de modélisations [42] tel que par exemple : le langage et la méthode ARCADIA implémentés dans l'outil Capella ou le langage SysML et la méthode Harmonie-SE implémentés avec l'outil Rhapsodie (cf figure 1.15).

On trouve ainsi des applications dans les secteurs de l'aéronautique [43], du nucléaire [44], de l'énergie [45].

SysML et ARCADia disposent de sémantiques permettant de décrire des systèmes complexes sous différents points de vues, cependant la sémantique de ces langages

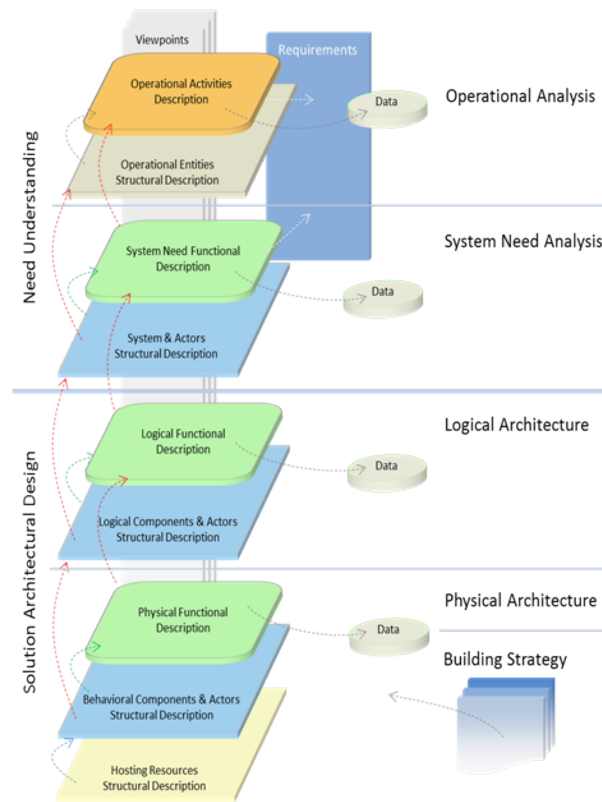


FIGURE 1.14 – Principales vues et perspectives structurant la démarche Arcadia [1].

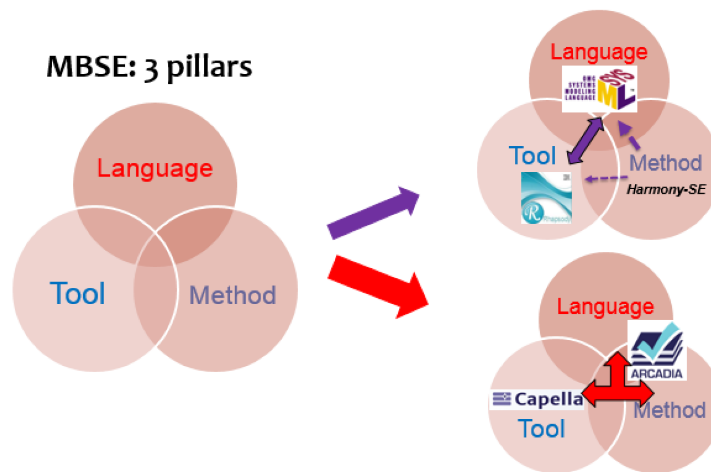


FIGURE 1.15 – Langage, méthodes et outils d'ingénierie système issu de [42].

comportent de nombreux points de variations. Par exemple la sémantique des machines à états de SysML et ARCADIA basée sur UML2 comportent des points de variations [46]. Pour Exécuter et simuler le comportement de ces modèles il faut pré-

ciser la sémantique des langages de modélisation. Nous verrons dans le chapitre 4 qu'il existe plusieurs stratégies permettant de préciser la sémantique d'un langage de simulation [47].

1.6 Conclusion du premier chapitre

Dans ce premier chapitre consacré aux modèles et aux systèmes, nous avons présenté les notions de systèmes et de modèles. Nous avons ensuite présenté les concepts et des outils utilisés dans l'ingénierie système basée sur les modèles. Dans le chapitre suivant nous présentons la formalisation des modèles et des transformations de modèles à l'aide des grammaires de graphes. Nous utiliserons alors les grammaires de graphes en seconde partie de ce manuscrit pour formaliser notre méthode de transformation de modèles.

CHAPITRE 2

FORMALISATION DES MODÈLES PAR GRAMMAIRE DE GRAPHES

2.1 Grammaire de graphes

Les graphes peuvent être utilisés pour décrire des structures statiques comme des structures de classes ou d'objets. Ils peuvent être aussi utilisés pour décrire les évolutions dynamiques de ces structures. Dans ce chapitre, nous étudions à travers des articles de la littérature, l'utilisation des grammaires de graphes pour la formalisation des transformations de modèles. Dans un premier temps nous reprenons la définition d'un graphe et du morphisme de graphes. Dans un second temps nous présentons le parallèle qui est fait entre les grammaires de graphes et les modèles de classes. Ces définitions nous permettent d'introduire le concept de grammaire de graphes triples afin de formaliser les transformations de modèles.

2.1.1 Formalisation par grammaire de graphes

La théorie des grammaires de graphes et des systèmes de transformations de graphes trouve son origine dans les années 1970. Elle a été développée par les équipes de TU Berlin (H. Ehrig, M. Pfender, and H.J. Kreowski), l'université de Erlangen (H.J. Schneider), et IBM Yorktown Heights (B. Rosen). La théorie a été complétée dans les année 80 et 90 par l'équipe de TU Berlin (H. Ehrig, G. Taentzer, M. Löwe, and R. Heckel). Les résultats de ces travaux sont regroupés dans deux volumes du *"Handbook of Graph Grammars and Computation by Graph Transformation"* [48] [49] Enfin la théorie a été étendue par la même équipe aux transformations de graphes typés attribués dans les années 2000-2010 [50] et [51]. Les définitions que nous présentons dans cette partie sont principalement issues de ces travaux[52] [53] [50].

2.1.2 Graphes et morphismes de graphes

Définition 1 (Graphe) *Un graphe $G = (V, E, s, t)$ est constitué d'un ensemble de nœuds V, d'un ensemble d'arcs E, ainsi que des fonctions sources et cibles $s, t : E \to V$.*

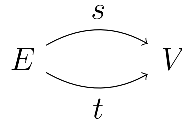


FIGURE 2.1 – Graphe

Remarque : Nous notons qu'un arc $e \in E$ est associé au plus, à un sommet source $v_s \in V$ et au plus à une cible $v_t \in V$. Un sommet $v \in V$ peut être associé à n arcs sources et n arcs cibles avec $n \in \mathbb{N}$.

Définition 2 (Morphisme de graphes) *Soit deux graphes $G_i = (V_i\ ,\ E_i\ ,\ s_i\ ,\ t_i)_{i\in\{1,2\}}$, un morphisme de graphes $f : G_1 \to G_2,\ f = (f_v\ ,\ f_e)$ est composé de deux fonctions $f_v : V_1 \to V_2$ et $f_e : E_1 \to E_2$ préservant les fonctions sources et cibles. Tel que $f_v \circ s_1 = s_2 \circ f_e$ et $f_v \circ t_1 = t_2 \circ f_e$.*

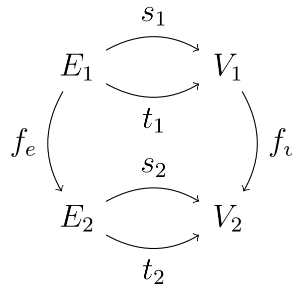


FIGURE 2.2 – Morphisme de graphes

Nous pouvons assigner des types à chaque élément du graphe [54]. Cela peut être réalisé par la création d'un "graphe de type" TG. Ce graphe contient les différents types et leurs relations. Alors le tuple $(G, type_G)$ d'un graphe G associé au morphisme de graphe $type_G : G \to TG$, est un graphe typé.

Définition 3 (Morphisme de graphes typés) *Soit un graphe G instance du graphe TG par le morphisme de typage $type_G$, et le graphe H instance du graphe TH par le morphisme de typage $type_H$. De plus, soit le morphisme $g : TG \to TH$. Un morphisme $f : G \to H$ est une instance du morphisme g si : $type_H \circ f \leq g \circ type_G$ (figure 2.3).*

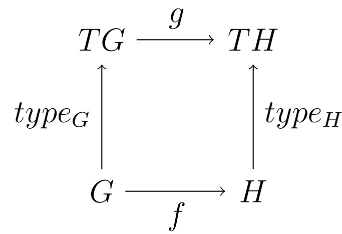


FIGURE 2.3 – Morphisme de graphes typés

La figure 2.4 montre un exemple de morphisme de graphes typés. Les graphes TG et TH sont des graphes de définition de type, les graphe G et H sont respectivement les instances des graphes TG et TH les rectangles représentent les nœuds des graphes, les flèches pleines représentent les arcs. Les fléches pointillées les morphismes. Pour des raisons de lisibilité, seules les correspondances entre les arcs *« manage »* des graphes TG et TH et les arcs *« manage »* des graphes G et H ont été représentées.

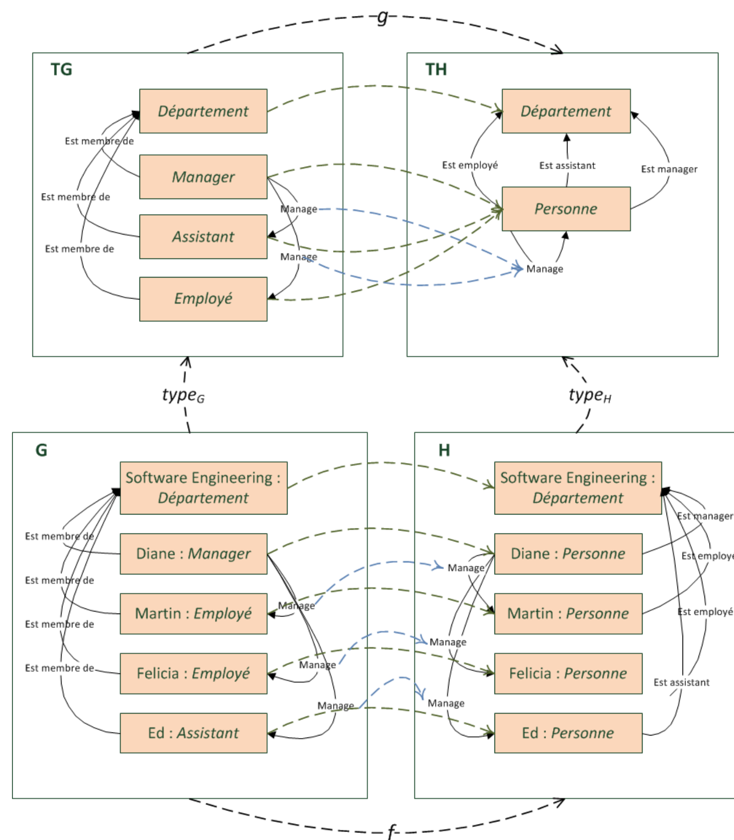


FIGURE 2.4 – exemple de Morphisme de graphe Typé

2.1.3 Graphes et morphisme de graphes avec héritage, contenance et attributs.

Les structures que nous manipulons en IDM sont plus complexes que les graphes typés. Elles possèdent en plus les notions d'héritage, de contenance, et d'attributs. Pour que les grammaires de graphes soient applicables aux méta-modèles et aux modèles manipulés en IDM, l'article de Enrico Biermann et al. [52] propose une définition de graphes avec liens de contenance (définition 4) et une définition de la racine d'un graphe (définition 5)

Définition 4 (Graphe avec liens de contenance (C-graphe)) *Un graphe avec liens de contenance en abrégé C-graph est un graph G avec un ensemble distinct d'arcs de contenance $E_{cont} \subseteq E$. Les liens de contenance induisent la relation transitive binaire $contient_G$ suivante : $contient_G = \{(x, y) \in V \times V \mid \exists e \in E : (s(e) = x \wedge t(e) = y)\} \cup \{(x, y) \in V \times V \mid \exists z \in V : (x\, contient_G\, z \wedge z\ \ contient_G\, y)\}$*

Définition 5 (Racine d'un Graphe) *Un C-graphe possède un nœud racine r si $\exists! r \in V$, tel que $\forall x \in V,\ x \neq r \wedge r\, contient_G\, x$.*

Enfin la notion d'héritage couplée aux liens de contenance a été formalisée par [55]. Les définitions 6 et 7 sont issues de cet article.

Définition 6 (Graphe avec héritage et liens de contenance) *Le tuple $G = (T, I, Vabs, C)$ est appelé graphe avec relations d'héritage et liens de contenance, en abrégé IC-graphe. où*

- *T est un graphe $T = (V,\ E,\ s,\ t)$,*
- *I est l'ensemble des arbres d'héritage $I = (I_V, I_E, s, , t)$ avec $I_V = V$ et $I_E \cap E = \varnothing$,*
- *$Vabs$ un ensemble $Vabs \subseteq V$ de nœuds abstraits,*
- *C un ensemble $C \subseteq E$ d'arcs de contenance.*

avec :

- *pour chaque noeud $v \in I_V$ le clan[1] d'héritages est défini par $clan_I(v) = \{v' \in I_V \mid \exists\, un\ chemin\, v' \xrightarrow{*} v \in I\} \subseteq I_V$ avec $v \in clan_I(v)$*
- *$\forall v, w \in I_V : v \in clan_I(w) \wedge m \in clan_I(v) \Rightarrow v = w$ (absence de cycle d'héritage)*

1. Ici un clan est un sous-ensemble d'une clique tel-que défini dans [56]

De plus les propriétés suivantes sont définies :

- $clan_I(M) = \bigcup_{v \in M} clan_I(v)$,
- $contient_G$ *définit la relation de contenance correspondant à* C *:* $contient'_G = \{(v,w) \in V \times V | \exists c \in C \wedge (x,y) \in V \times V : s_T(t) = x\ avec\ v \in clan_G(x) \wedge t_T(t) = y\ avec\ w \in clan_G(y)\}$ $contient_G$ *est une fermeture transitive de* $contient'_G$
- *Un graphe G possède une racine s'il existe un nœud concret (non-abstrait)* $r \in V - Vabs$ *appelé nœud racine, lequel contient transitivement tous les autres nœuds concrets :* $\forall v \in V - Vabs - \{r\} : r\ contient_g\ v$.

Les éléments du tuple $T, I, Vabs, C$ *du graphe* G *peuvent aussi être notés respectivement :* $T(G), I(G), Vabs(G), C(G)$.

Définition 7 (Morphisme de graphes avec héritage et liens de contenance) *Soit deux IC-graphes* G *et* H *un morphisme* $f : G \to H$, *un morphisme de graphes avec relations d'héritage et liens de contenance, en abrégé IC-morphisme, est défini sur le morphisme de graphes* $f_T : T(G) \to T(H)$. *Le mapping entre les ensembles* $Vabs(G)$ *et* $Vabs(H)$ *est induit par* $f_{T,V}$. *Le mapping entre les nœuds des arbres d'héritage* $I(G)_V$ *et* $I(h)_V$ *est induit par* $f_{T,V}$. *Le mapping entre les ensembles* $C(G)$ *et* $C(H)$ *est induit par* f_E. *Nous utiliserons désormais les abréviations :*

- f *pour désigner* f_T,
- f_V *pour désigner* $f_{T,V}$
- f_E *pour désigner* $f_{T,E}$

De plus les propriétés suivantes doivent être vérifiées :

1. $f_V \circ s_{T(G)} \subseteq clan_{I(H)}(s_{T(H)} \circ f_E)$ *et* $f_v \circ t_{T(G)} \subseteq clan_{I(H)}(c_{T(H)} \circ f_E)$ *: les mappings entre la source et la cible doivent être clan-compatibles.*
2. $f_V(clan_{I(G)}) \subseteq clan_{I(H)}(f_V)$ *: le mapping des nœuds est clan-compatible.*
3. $f_E(C(G)) \subseteq C(H)$. *Le mapping des arcs conserve les liens de contenance.*

Enrico Biermann et al. étendent ensuite leurs définitions aux graphes typés :

Définition 8 (IC-Graphe typé) *Soit un graphe simple* $G = (V, E, s, t)$ *avec héritage et liens de contenance* GC' *et un IC-graphe* $GC = (G, I(GC'), E(GC'), C)$ *appelé IC-graphe typé, (ou instance de IC-Graph) est définie par le morphisme complet total IC-morphism* $type_{GC} : GC \to TGC$ *appelé typage de IC-morphisme, avec :*

1. $C = \{e \in E | type_{GC}(e) \in C(TGC\}$ C *est induit par le morphisme de typage*
2. $e1, e2 \in C : t_G(e1) = t_G(e2) \Rightarrow e1 = e2$ *au plus un conteneur*
3. $(x, x) \notin contains_{GC}$ $\forall x \in V$ *pas de cycle de contenance*

Définition 9 (IC-Morphisme typé) *Considérons les IC-graphs et le IC-Morphisme : Soit un graphe G instance du graphe TG par le morphisme de typage $type_G$, et le graphe H instance du graphe TH par le morphisme de typage $type_H$. De plus, soit le morphisme $g : TG \to TH$. Un IC-Morphisme $f : G \to H$ appelé IC-morphisme est une instance du morphisme g si : $type_H \circ f \leq g \circ type_G$ (figure 2.5)*

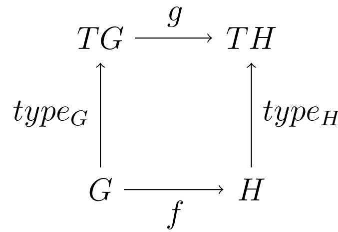


FIGURE 2.5 – IC-Morphisme typé

La figure 2.6 présente un exemple d'IC-graphes typés de leurs instances ainsi qu'un morphisme d'IC-graphe (en lignes pointillées)

Dans l'exemple de la figure 2.6 deux graphes, représentent la structure d'un département et de ses employés(les deux carrés supérieurs de la figure). Les deux représentations sont mappées par un morphisme (lignes pointillées). Les deux carrés inférieurs représentent quant à eux les instances de ces IC-graphes mappés par le morphisme d'IC-graphe typé (lignes pointillées). Le mapping de cet exemple est partiel, le concept d'assistant n'est pas transposé dans le graphe cible. Le concept de manager est lui fusionné avec le concept d'employé.

Enfin, nous définissons la notion de graphes attribués. Un graphe attribué est un graphe dont les arcs et les nœuds (concrets ou abstraits) peuvent posséder des attributs. Dans leur article [53] Juan de Lara et al. proposent une définition des graphes et des transformations de graphes typés et attribués. Ils définissent un nouveau type de graphe qu'ils appellent E-Graph. pour obtenir un E-graph ils ajoutent au graphe de la définition1 un noeud attribut.

Définition 10 (E-graphe et morphisme de E-graphe) *Un E-graphe G avec*
$G = (V_G, V_D, E_G, E_{NA}, E_{EA}, (source_j, target_j)_{j \in \{G,NA,EA\}}$ *est constitué des ensembles :*

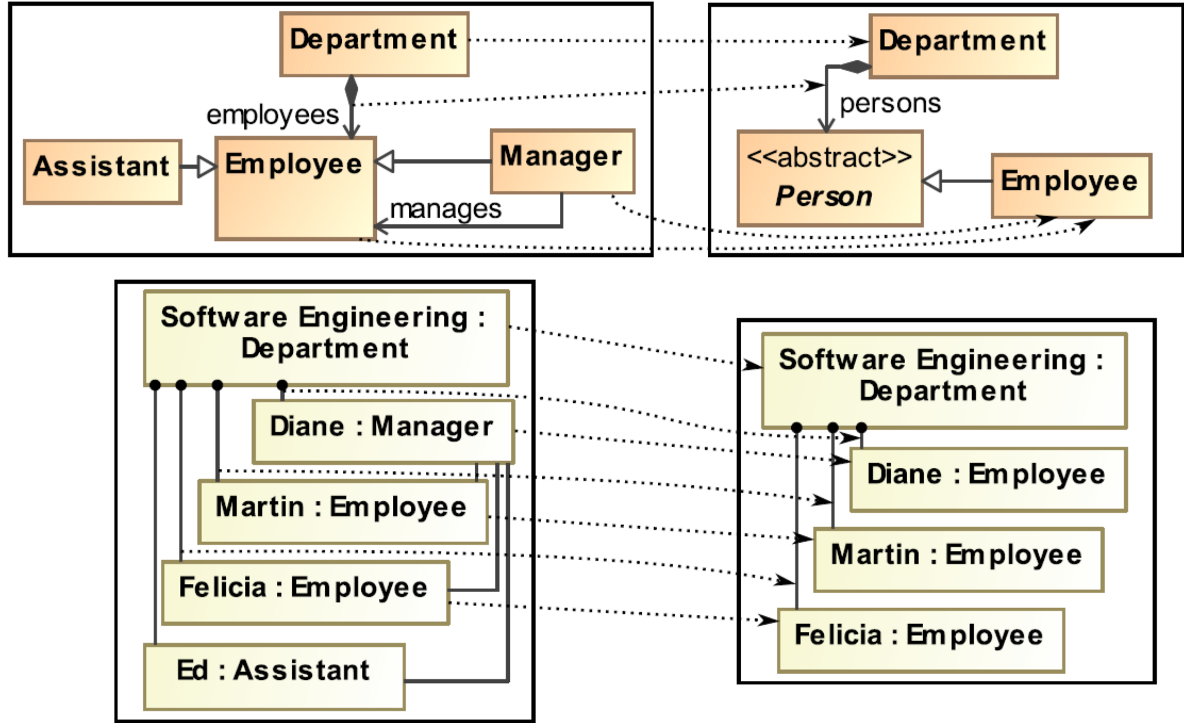


FIGURE 2.6 – Exemple d'IC-graphe typé et de morphisme d'IC-graphe donné par [55]

- *V_G et V_D appelés respectivement nœuds du graphe et nœuds de données,*
- *E_G, E_{NA}, E_{EA}, appelés respectivement arcs du graphe, arcs de nœuds attribués et arcs d'arcs attribués,*

ainsi que des fonctions sources (source) et cibles (target) :

- *$source_G : E_G \to V_G, target_G : E_G \to V_G$ pour les arcs du graphe.*
- *$source_{NA} : E_{NA} \to V_G$, $target_{NA} : E_{NA} \to V_D$ pour les arcs de noeuds attribués,*
- *$source_{EA} : E_{EA} \to E_G$, $target_{EA} : E_{EA} \to V_D$ pour les arcs d'arcs attribués*

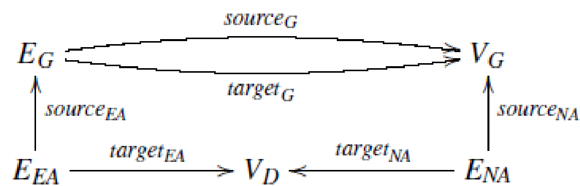


FIGURE 2.7 – E-graphe

Soit $G^k = (V_G^k, V_D^k, E_G^k, E_{NA}^k, E_{EA}^k, (source_j^k, target_j^k)_{j \in \{G, NA, EA\}}$ pour $k = 1, 2$ deux

E-graphes. Un morphisme de E-graphe $f : G^1 \to G^2$ est un tuple $(f_{V_G}, f_{V_D}, f_{R_{NA}}, f_{E_{EA}})$ avec $f_{V_i} : V_i^1 \to V_i^2$ et $f_{E_j} : E_j^1 \to E_j^2$ pour $i \in \{G, D\}, j \in \{G, NA, EA\}$ tel que f commute avec toutes les fonctions sources et cibles (par exemple $f_{V_G} \circ source_G^1 = source_G^2 \circ f_{E_G}$)

Les ensembles E_{NA} et E_{EA} sont nécessaires pour allouer plusieurs attributs aux arcs et aux nœuds. Les E-graphes et les morphismes de E-graphes forment la catégorie des **EGraphs**. Cependant l'ajout d'un nœud d'attribut V_D au graphe n'est pas suffisant pour que le graphe devienne un graphe "attribué". Il faut ajouter à ce nœud des données. Ainsi Juan de Lara et al. proposent dans [53] de définir le graphe attribué comme un E-graphe combiné avec une algèbre sur des signatures de données [57]

Définition 11 (Graphes attribués et morphisme de graphes attribués) *Soit $DSIG = (S_D, OP_D)$ une signature de données avec tri des valeurs d'attributs. $S'_D \subseteq S_D$. Un graphe attribué $AG = (G, D)$ consiste en un E-Graphe G avec une algèbre-DSIG D tel que $\biguplus_{s \in S'_D} D_s = V_D$.*
Pour deux graphes attribuées $AG^i = (G^i, D^i)$ avec $i = 1, 2$ un morphisme de graphe attribué $f : AG^1 \to AG^2$ est le couple $f = (f_G, f_D)$ avec un morphisme de E-Graphe $f_G : G^1 \to G^2$ et un homomorphisme $f_D : D^1 \to D^2$ tel que (1) commute pour tout $s \in S'_D$.

$$\begin{array}{ccc} D_s^1 & \xrightarrow{f_{D,s}} & D_s^2 \\ \downarrow & (1) & \downarrow \\ V_D^1 & \xrightarrow{f_{G,V_D}} & V_D^2 \end{array}$$

FIGURE 2.8 – Morphisme de graphes attribués

Étant donné une signature de données DSIG, les graphes attribués et les morphismes de graphes attribués forment la catégorie **AGraphs**.

Pour que le formalisme soit complet il reste à ajouter la notion de type pour les graphes attribués.

Définition 12 (Graphes attribués typés et leurs morphismes) *Etant donné une signature de donnée $DSIG$ un graphe attribué de type $ATG = (TG, Z)$ ou Z est une*

algèbre-DSIG finale. avec $Z_s = \{s\} \forall s \in s_D$. Un graphe attribué typé (AG, t) instance de ATG consiste en un graphe attribué AG et un morphisme de graphe attribué $t : AG \rightarrow ATG$. Un morphisme de graphe attribué typé $f : (AG^1, t^1) \rightarrow (AG^2, t^2)$ est un morphisme de graphe attribué $f : (AG^1 \rightarrow AG^2)$ tel que $t^2 \circ f = t^1$.

Les auteurs présentent dans leur article un exemple de graphe typé attribué en explicitant différentes notations (figure 2.9) Enfin les auteurs ajoutent la notion de nœud

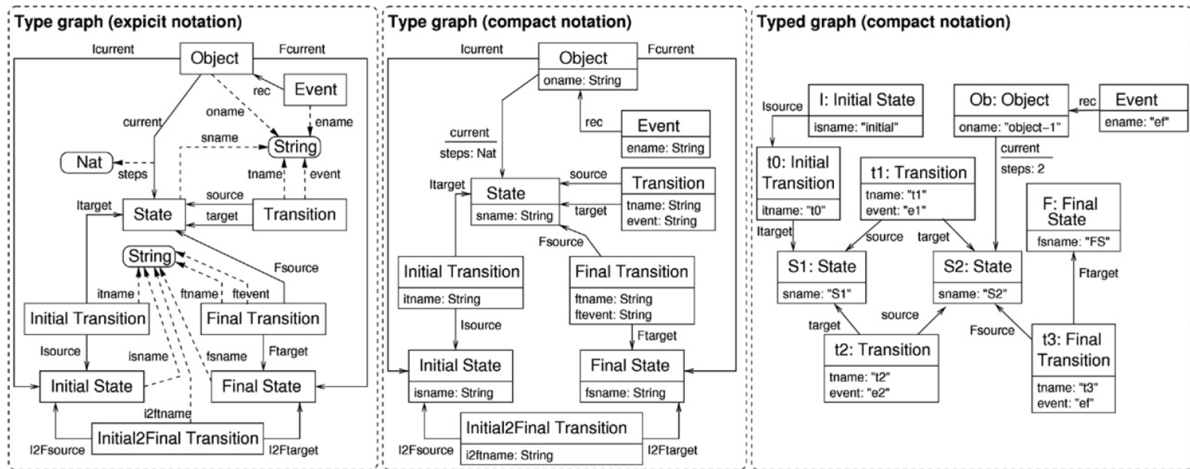


FIGURE 2.9 – Exemples de graphes Attribués typés

abstrait et d'héritage aux graphes attribués typés.

Définition 13 (Graphes attribués de définition de type avec héritage) *Un graphe attribué de type avec héritage $ATGI = (TG, Z, I, A)$ consiste en un graphe attribué TG, un graphe d'héritage $I = (I_V, I_E, s, t)$ avec $I_V = TG_{V_G}$, et un ensemble $A \subseteq I_V$ appelé nœuds abstraits. Pour chaque nœud $n \in I_V$ le clan d'héritage est défini par :*
$clan_I(n) = \{n' \in I_V | \exists path\, n' \xrightarrow{*} n \in\ I\} \subseteq I_V$ *avec* $n \in clan_i(n)$.

Remarque : $x \in clan_I(n) \Rightarrow clan_I(x) \subseteq clan_I(y)$.

L'exemple de la figure 2.10 reprend le graphe de la figure 2.9 en le modifiant pour lui adjoindre des nœuds abstraits.

Nous possédons désormais l'outillage conceptuel permettant de formaliser les graphes. On s'intéresse maintenant à la formalisation de la transformation de graphe.

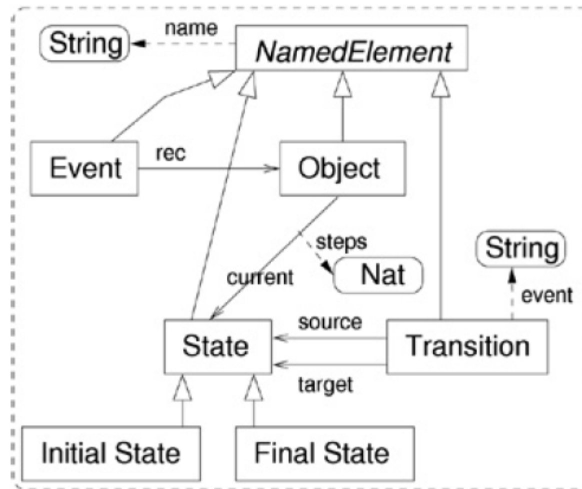


FIGURE 2.10 – Exemples de graphes attribués avec héritage

2.2 Approche algébrique de la transformation des graphes.

Les graphes et les morphismes de graphes définissent la catégorie algébrique des $Graphs$, et $Graphs_{TG}$ pour les graphes typés. Les constructions et les résultats de cette catégorie sont applicables dans une approche algébrique de la transformation des graphes. Un des concepts importants de la catégorie de $Graphs$ (resp. $Graphs_{TG}$), est la réécriture de graphes par l'approche de "pushout" (ou "somme amalgamée"). L'idée principale est de construire des graphes en utilisant un graphe intermédiaire appelé "gluing" (cf figure 2.11). Soit deux graphes G_1 et G_2 possédant une intersection commune G_0 tel que $G_0 = G_1 \cap G_2$, le graphe "gluing" G_3 de G_1 et G_2 par rapport à G_0, est donné par $G_3 = G_1 \cup G_2$ Ce concept est la base de l'approche algébrique de la transformation des graphes. Dans cette approche, une transformation de graphes est basée sur l'application d'une opération de "*production*" sur un graphe source.

Définition 14 (production) . *Une production $p = (L \stackrel{l}{\leftarrow} K \stackrel{r}{\rightarrow} R)$ est constituée de trois graphes L, K et R, appelés respectivement "gauche" (left hand side), "interface" (glueing graph) et "droite" (right hand side), ainsi que de deux morphismes de graphes injectifs l et r.*

Le graphe gauche L représente la pré-condition de la règle, tandis que le graphe droit R décrit la post-condition. Le graphe K est l'interface, il est inclus dans L et R. Dans la pratique, K est filtré lors de l'application de p mais n'est pas modifié [58].

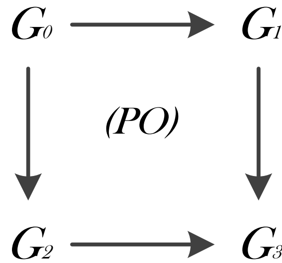


FIGURE 2.11 – diagramme de pushout

Définition 15 (Transformation directe de graphes) *. Soit une production $p = (L \stackrel{l}{\leftarrow} K \stackrel{r}{\rightarrow} R)$, et un graphe G associé à un morphisme de graphe : $m : L \rightarrow G$ appelé filtre (ou "match"). Une transformation $G \stackrel{\rho,m}{\Longrightarrow} H$ d'un graphe source G vers un graphe cible H est donnée par le diagramme de "double pushout" (DPO) (ou "double somme amalgamée") figure 2.12 , où (PO1) et (PO2) sont deux pushout de la catégorie $Graphs$, ($Graphs_{TG}$ si les graphes sont typés)*

La figure 2.13 issue de [59] montre un exemple de transformation de graphes par un diagramme DPO. Dans cette exemple, G est le graphe source et H le graphe cible. G possède 5 noeuds (1,2,3,6 et 7) reliés par des arcs. H possède 6 noeuds (1,2,4,5,6,7) reliés par des arcs. La production est donnée par $p = (L \stackrel{l}{\leftarrow} K \stackrel{r}{\rightarrow} R)$. Elle permet la transformation de G vers H. Le graphe K est le graphe "interface" pour les deux pushout. Il est à l'intersection de l'ensemble des graphes. Ainsi, pour le premier pushout (PO1), $K = L \cap D$ et $G = L \cup D$. Pour la seconde pushout (PO2), $K = R \cap D$ et $H = R \cup D$. Une propriété importante est que la transformation de graphe par DPO est réversible. La transformation peut se lire de droite à gauche et de gauche à droite.

Juan de Lara et al. proposent dans [53] un exemple de transformation de graphes typés par DPO (figure 2.14 L'exemple décrit une machine d'états-transitions contenant deux états et deux transitions. La transformation permet de déplacer l'arc pointant vers l'état courant de l'état S1 vers l'état S2. Dans cet exemple le graphe source et le graphe cible sont le même graphe : la transformation est endogène, le graphe est modifié par la transformation. [59] étend la définition de la production pour lui adjoindre une condition d'application.

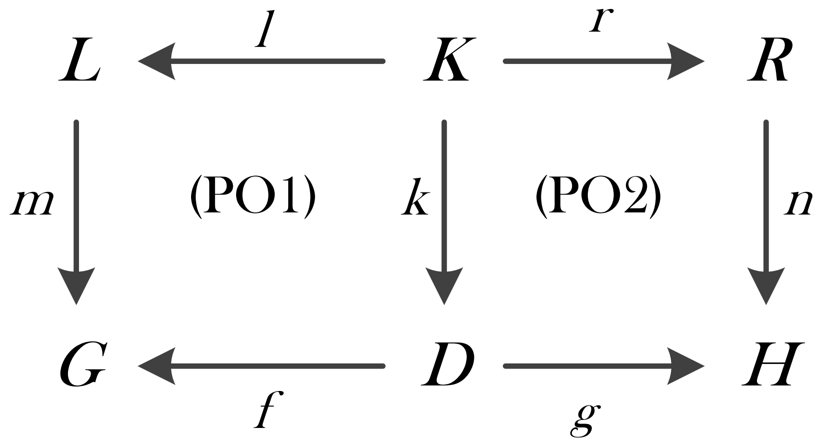


FIGURE 2.12 – diagramme de double pushout

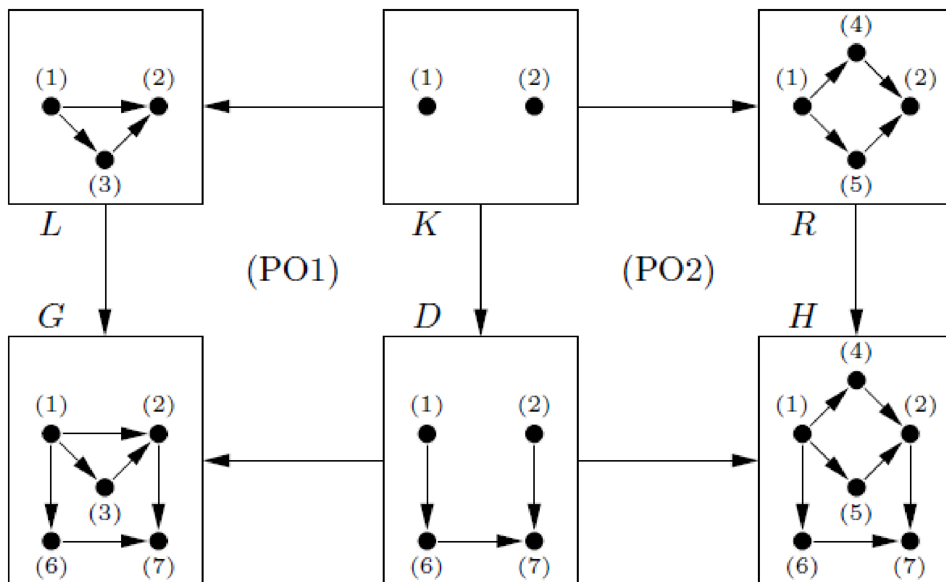


FIGURE 2.13 – Exemple de transformation par DPO extrait de [59]

Définition 16 (Règle) *Une règle* $\rho = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R, ca)$ *est constituée de trois graphes L,K,R appelés respectivement Gauche (Left hand side), Colle (Glueing) et Droite (Right*

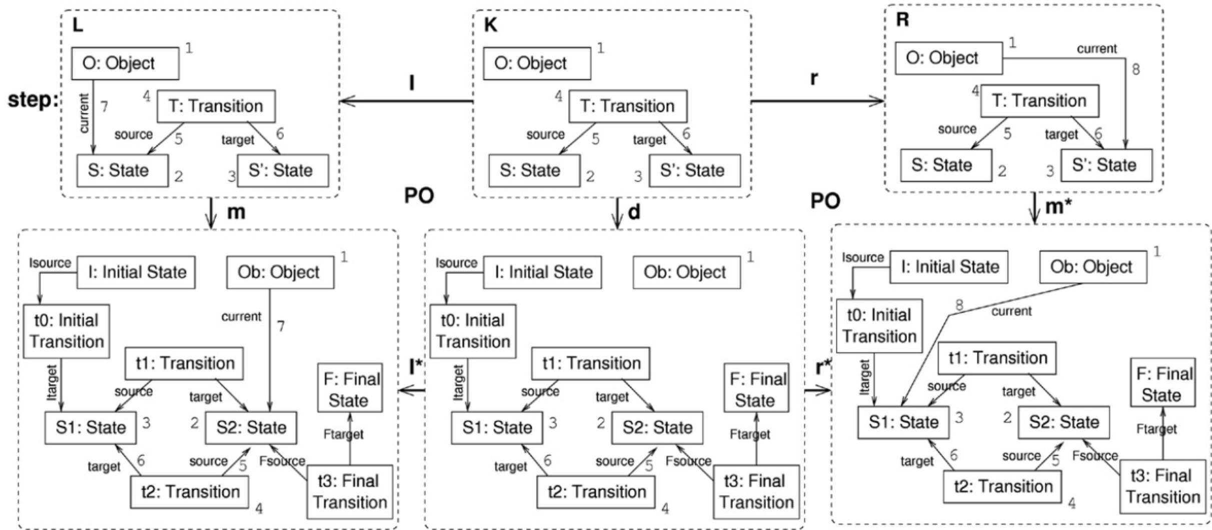


FIGURE 2.14 – Transformation directe d'un graphe typé par DPO extrait de [53]

hand side) ; de deux morphismes de graphes l et r ; ainsi que d'une condition ca sur L, appelée condition d'application.

Une transformation de graphes décrit alors l'application d'une règle sur un graphe via une production satisfaisant la condition d'application.

Définition 17 (Transformation suivant une règle) *Soit une règle $\rho = (L \stackrel{l}{\leftarrow} K \stackrel{r}{\rightarrow} R, ca)$, un graphe source G, un graphe cible H. Le morphisme $m : L \rightarrow G$, est appelé $match$ (ou "filtre"), tel que $m \models ca$. La transformation élémentaire $G \stackrel{\rho,m}{\Longrightarrow} H$, de G vers H forme un Double Pushout (DPO) figure 2.15.*

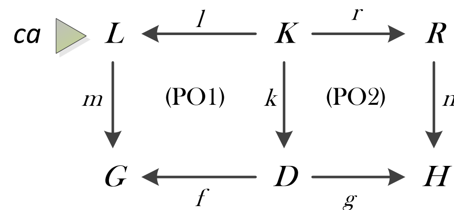


FIGURE 2.15 – Transformation avec condition d'application

Dans la pratique la condition d'application ca est un booleen qui peut être appliqué sur les éléments du graphe de pré-condition K alors le morphisme de filtre m doit

satisfaire ca, $m \models ca$. La condition d'application peut aussi être appliquée aux éléments du graphe de post-condition R alors $n \models ca$.

Il est important de noter que dans une transformation par un DPO, la structure du graphe source n'est pas conservée. Elle est modifiée par la transformation tel que montré dans l'exemple de la figure 2.13 .

2.2.1 Transformation de modèles et grammaire de graphes triples

Afin de formaliser les transformations de modèles il est possible de considérer les modèles comme des graphes. En ce sens, [51] définit un lien entre les notions de méta-modèle et les notions de la théorie des graphes. Il définit ensuite les concepts généraux de la transformation de modèles basée sur la transformation de graphes.

Transformations de modèles basées sur la transformation de graphes

Ehrig Hartmut et al. montrent dans [51] la relation entre les notions de graphes et les notions de modèles. Ainsi la notion de méta-modèle peut être reliée à la notion de type-graphe définie précédemment. Le tableau 2.1 montre les correspondances entre les notions de modèles et les notions de graphes.

Notions des modèles	Terminologie des graphes
Méta-Modèle	Type-graphe (TG)
Classe	Nœud $\in TG$
Classe abstraite	Nœud abstrait $\in A \in ATGI$
Association	Arc du type-graphe TG
Attribut	Nœud attribut $\in TG$
Instance de modèle	Graphe (G) typé sur TG
Objet	Nœud $\in G$
Référence	Arc $\in G$

TABLE 2.1 – Terminologie des modèles vs graphes

[51] définit, page 51, les concepts généraux de la transformation de modèles basée sur la transformation de graphes :

Définition 18 (Transformation de modèles basée sur la transformation de graphes)
*Soit **GRAPHS** la catégorie des graphes complets **AIC-TrGraph**.*

1. *Soit un méta-langage source $\mathcal{L}_s \subseteq$ **GRAPHS**$_{TG^S}$ et un méta-langage cible $\mathcal{L}_t \subseteq$ **GRAPHS**$_{TG^t}$, une transformation de modèle $TM : \mathcal{L}(TG^S) \Rrightarrow \mathcal{L}(TG^T)$ de $\mathcal{L}(TG^S)$ vers $\mathcal{L}(TG^T)$ est définie par $TM = (\mathcal{L}(TG^S), \mathcal{L}(TG^T), TG^c, s_G, t_G, SGT)$ où TG^c est un graphe de correspondance, s_G et t_G deux morphismes $(TG^s \leftarrow TG^c \rightarrow TG^t)$, où SGT est un système de graphes de transformations composé de règles non destructives R typées sur TG^c ainsi que de conditions d'application.*
2. *Soit $\mathcal{M}(G)$ un modèle induit par le AIC-graphe G conforme au langage $\mathcal{L}(TG)$ noté $\mathcal{M}(G) \models \mathcal{L}(TG)$. Une séquence de transformation de modèle via MT en abrégé $MT - sequence$ est donnée par $(G^S, G_1 \overset{*}{\Longrightarrow} G_n, G^T)$ où $\mathcal{M}(G^S) \models \mathcal{L}(TG^S)$, $\mathcal{M}(G^T) \models \mathcal{L}(TG^T)$.*
3. *La relation de transformation de modèles $MT_R \subseteq \mathcal{L}(TG^S) \times \mathcal{L}(TG^T)$ définie par MT est donnée par $(G^S, G^T) \in MT_R \Leftrightarrow \exists MT - sequence(G^S, G_1 \overset{*}{\Longrightarrow} G_n, G^T)$.*
4. *$TM : \mathcal{L}(TG^S) \Rrightarrow \mathcal{L}(TG^T)$ est :*
   (a) *syntaxiquement correcte si pour tout $(G^S, G^T) \in MT_R$ nous avons $\mathcal{M}(G^S) \models \mathcal{L}(TG^S) \wedge \mathcal{M}(G^T) \models \mathcal{L}(TG^T)$,*
   (b) *totale si : $\forall \mathcal{M}(G^S) \models \mathcal{L}(TG^S) \Rightarrow \exists!(G^S, G^T) \in MT_R$,*
   (c) *surjective si : $\forall \mathcal{M}(G^T) \models \mathcal{L}(TG^T) \Rightarrow \exists!(G^S, G^T) \in MT_R$,*
   (d) *complète, si MT_R est totale et surjective*

Les transformations de graphes via un double "pushout" sont destructives. Le graphe d'origine est réécrit et devient le graphe cible. Les éléments du graphe source absent dans le graphe cible ne sont pas préservés, tel que dans l'exemple présenté en figure 2.13.

Pour appliquer la transformation de graphes à la transformation de modèles, il faut pouvoir préserver la structure du graphe source lors de la transformation. C'est l'idée de la grammaire de graphes triples.

Grammaire de graphes triples

La grammaire de graphes triples a été introduite par Schürr dans [60] et formalisée dans [61] puis étendue et généralisée aux graphes typés et attribués dans [62]. La grammaire de graphes triples a pour avantage de pouvoir spécifier des transformations de modèles de manière intuitive sur la base d'un formalisme.

Définition 19 (Graphes Triples) *Un triple graphe $G = (G^s \xleftarrow{s_G} G^c \xrightarrow{t_G} C^t)$, consiste en trois graphes : G^s, relatif au domaine source, G^t, relatif au domaine cible et un graphe de correspondance G^c relatif aux règles de transformations. Ces graphes sont reliés entre eux par deux morphismes de graphes $s_G : G^s \rightarrow G^c$ et $c_G : G^c \rightarrow G^t$. Un morphisme de graphes triples $m = (m_s, m_c, m_t) : G \rightarrow H$ matche les graphes et préserve les correspondances entre ces graphes. Formellement un morphisme de graphes triples m consiste en des morphismes de graphes $m_s : G^s \rightarrow H^s$, $m_r : G^r \rightarrow H^r$, $m_t : H^t \rightarrow G^t$ tel que $m_s \circ s_G = s_H \circ m_c$ et $m_t \circ t_G = t_H \circ m_c$.*

$$
\begin{array}{ccccccc}
G & = (G^S & \xleftarrow{s_G} & G^C & \xrightarrow{t_G} & G^T) \\
m \downarrow & m^S \downarrow & & m^C \downarrow & & m^T \downarrow \\
H & = (H^S & \xleftarrow[s_H]{} & H^C & \xrightarrow[t_H]{} & H^T)
\end{array}
$$

FIGURE 2.16 – Morphisme de graphes triples

Un graphe triple typé $(G, type_G)$ est donné par le morphisme de type $type_G : G \rightarrow TG$ du graphe triple G vers le graphe triple TG.

Définition 20 (système de transformation et grammaire de graphes triples) *Un système de transformation de graphes triples $SGT = (G, \rho)$ est un couple (G, ρ), où G est un graphe triple et ρ un ensemble de règles de transformations.*
Une grammaire de graphes triples $GGT = (SGT, S)$est un couple (SGT, S) où SGT est un système de transformations de graphes triples et S un graphe source.
Le langage $\mathcal{L}(GGT)$ de la grammaire de graphes triples est défini par :
$\mathcal{L}(GGT) = \{G | \exists\, transformation\; S \overset{*}{\Longrightarrow} G\; via\; \rho\}$

Les graphes triples et les graphes triples typés forment (avec leur règles de composition et d'identités) les Catégories $TrGraphs$ et $TrGraphs_{TG}$

La figure 2.17 tirée de [51] montre un exemple de grammaire de graphes triples d'une transformation d'une structure de diagramme de classes (TG^S) en une structure de base de donnée (TG^T) par le graphe de correspondance (TG^C). Les nœuds *"classe"* du graphe source correspondent aux nœuds *"table"* du graphe cible. Les nœuds *"Attribute"* du graphe source correspondent aux nœuds *"Column"* du graphe

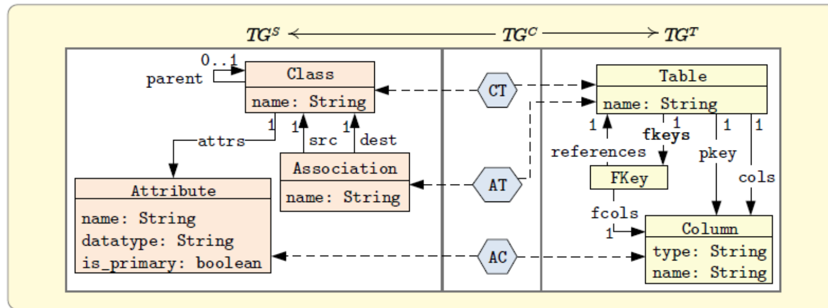


FIGURE 2.17 – Exemple de graphes triples [51]

cible. Enfin les nœuds *"Association"* du graphe source correspondent à des clés étrangères associées aux nœuds *"Table"* du graphe cible.

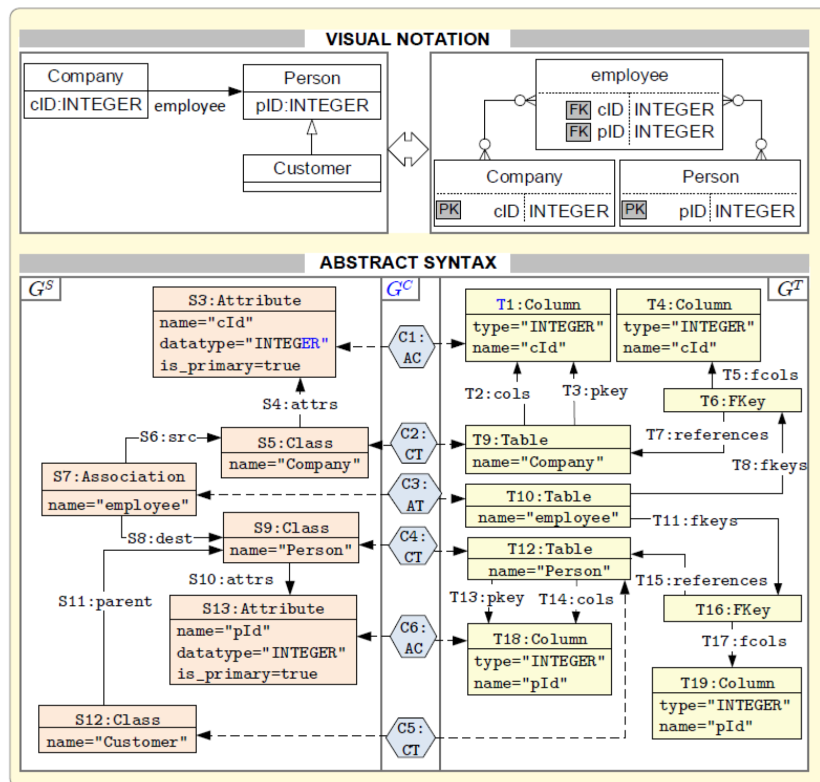


FIGURE 2.18 – Exemple d'instance de graphes triples [51]

Les auteurs présentent (figure 2.18) une instance du graphe triple $G = (G^S \leftarrow G^C \rightarrow G^T)$ par rapport à TG de la figure 2.17. La partie supérieure de l'exemple montre une notation visuelle possible, alors que la partie inférieure spécifie la syntaxe abstraite

du diagramme de classes, du modèle de base de données, ainsi que des liens de correspondance.

A l'image de la règle (définition 16) qui définit la règle de transformation pour le DPO, il existe une règle triple qui définit les transformations pour les graphes triples. Une règle triple est la construction de la source, de la cible et de la correspondance en une étape. Chaque règle triple établit un patron de correspondances entre les différents graphes. Formellement une règle triple est définie par :

Définition 21 (règle triple et transformation) *Une règle triple* $tr = (tr : L \to R, ac)$ *est constituée de deux graphes triples* L *et* R*, d'un* $\mathcal{M}-morphisme$ tr *et d'une condition d'application* ac *sur* L*. Étant donné un graphe triple* G*, une règle triple* $tr = (tr, ac)$ *et un filtre* $m : L \to G$ *avec* $m \models ac$*, une transformation triple directe* $G \overset{tr,m}{\Longrightarrow} H$ *de* G *par* tr *est donnée par le pushout de la figure 2.19*

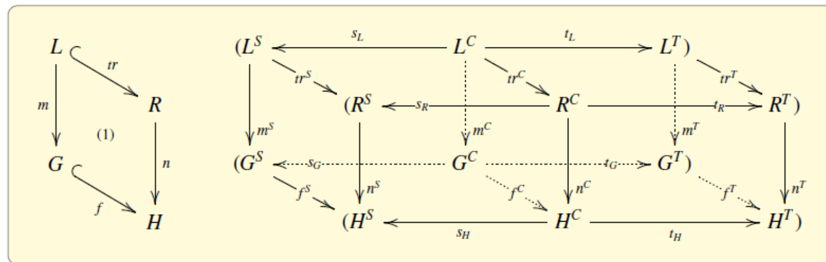


FIGURE 2.19 – Etapes d'une transformation triple [51]

La figure 2.20 montre un exemple de grammaire de règles triples. Les éléments de gauche et de droite de la règle sont représentés en un graphe triple. Les éléments créés sont représentés avec le symbole ++ et sont entourés de vert. Cinq règles composent la transformation complète : **Class2Table**, **Attr2Column**, **PrimaryAttr2Column**, et **Association2Table**.

- **Class2Table** génère dans le graphe source G^S un nœud *Class* possédant un attribut *name*, un nœud *CT* dans le graphe de correspondance G^C. Puis un nœud *Table* dans le graphe cible G^T.
- **Subclass2Table** génère un nœud de type *Class* dans G^S relié par un arc parent à un nœud de type *Class* préexistant.
- **Attr2Column** génère à partir d'un nœud *Class* et d'un nœud *Table* pré-existant, un nœud *Attribut* dans le graphe G^S, un nœud *AC* dans G^C et un nœud *Column* dans G^T.

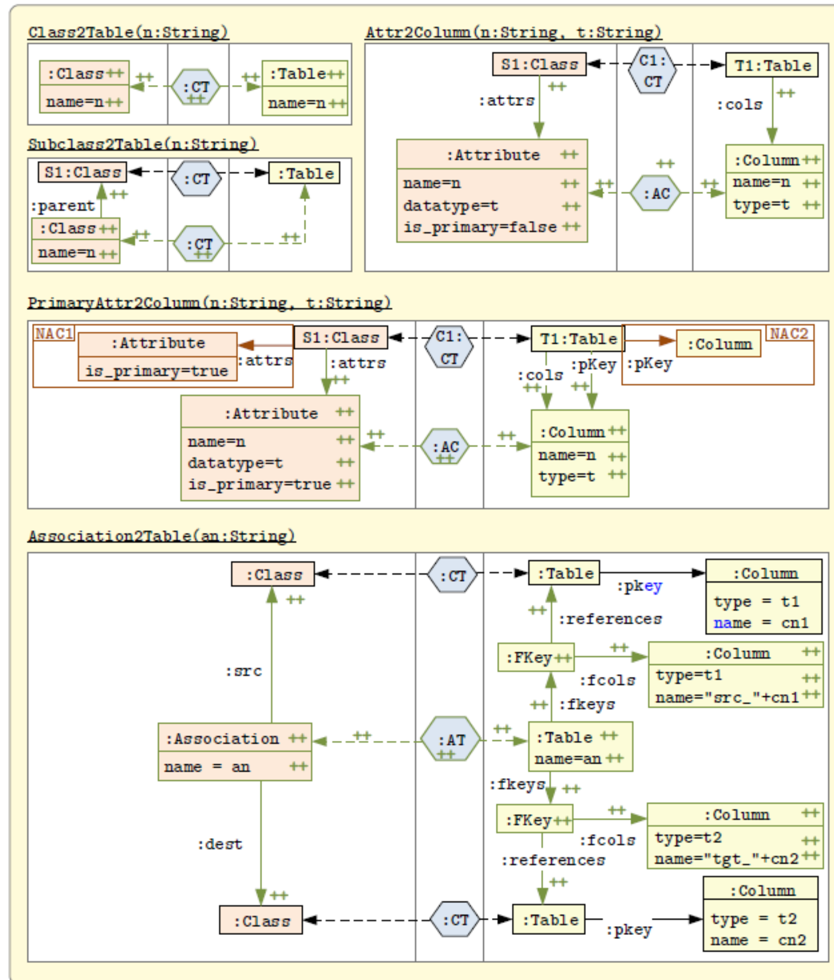


FIGURE 2.20 – Exemple de règles triples [51]

- **PrimaryAttr2Column** doit générer un nœud unique attaché à chaque nœud *Class* de G^S et à chaque nœud *Table* de G^T. L'unicité de la relation est assurée par la mise en place d'une condition d'application négative (NAC) [63] en rouge dans l'exemple.
- **Association2Table** crée une association dans G^S entre deux nœuds de type *Class*, composée d'un nœud de type *Association* et de deux arcs ; un *src* et un *dest*. Un nœud de type *Table* et deux nœud de type *Fkey* sont créés dans G^T. Le tout est relié à deux tables préexistantes par des arcs *references*.

Selon la définition 20, le graphe triple de la figure 2.18 instance du langage de graphes triples $\mathcal{L}(GGT)$, est formé à partir de la grammaire de graphes triples de la figure 2.20, et par la séquence : $\varnothing \overset{Class2Table}{\Longrightarrow} G_1 \overset{Class2Table}{\Longrightarrow} G_2 \overset{Subclass2Table}{\Longrightarrow} G_3 \overset{Attr2Column}{\Longrightarrow}$

$G_4 \overset{PrimaryAttr2Column}{\Longrightarrow} G_5 \overset{Association2Table}{\Longrightarrow} G_6$

Dans cet exemple les éléments du graphe source et du graphe cible sont générés de manière synchrone suivant les règles de transformations triples. Or dans les applications de transformations de modèles qui nous intéressent, les éléments du graphe source existent et nous voulons créer des éléments dans un graphe cible en accord avec des règles de transformations. Il s'agit d'un cas particulier de grammaire de graphes triples appelé "transformation avant" *(forward transformation)*.

[64] propose la définition suivante pour la règle de "forward transformation".

Définition 22 (Source dérivée, Règle de transformation avant) *Soit une règle triple $tr = (tr^S, tr^C, tr^T) : L \rightarrow R$ la règle source $tr_S : L_S \rightarrow R_S$ est dérivée par extension du morphisme de graphe $tr^S : L^S \rightarrow R^S$ constitués de graphes et de morphismes de graphes vides $L_S^C = L_S^T = R_S^T = R_S^T = \emptyset$. La règle de transformation avant $tr_F = (tr_F^S, tr_F^C, tr_F^T)$ est dérivée en prenant tr et en redéfinissant les éléments suivants : $L_F^S = R^S, tr_F^S = id$ et $S_{L_F} = tr^S \circ S_L$.*

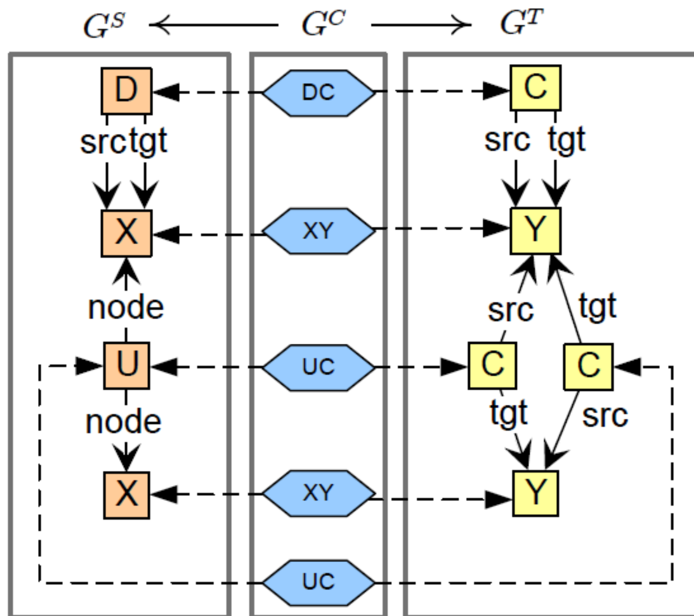


FIGURE 2.21 – graphe triple [64]

Les figures 2.21 et 2.22 illustrent un exemple de règles triples "avant". la première figure représente un graphe triple dans lequel les nœuds "X" du graphe source correspondent à des nœuds "Y" du graphe cible. et les nœuds "D" (connexion directe)

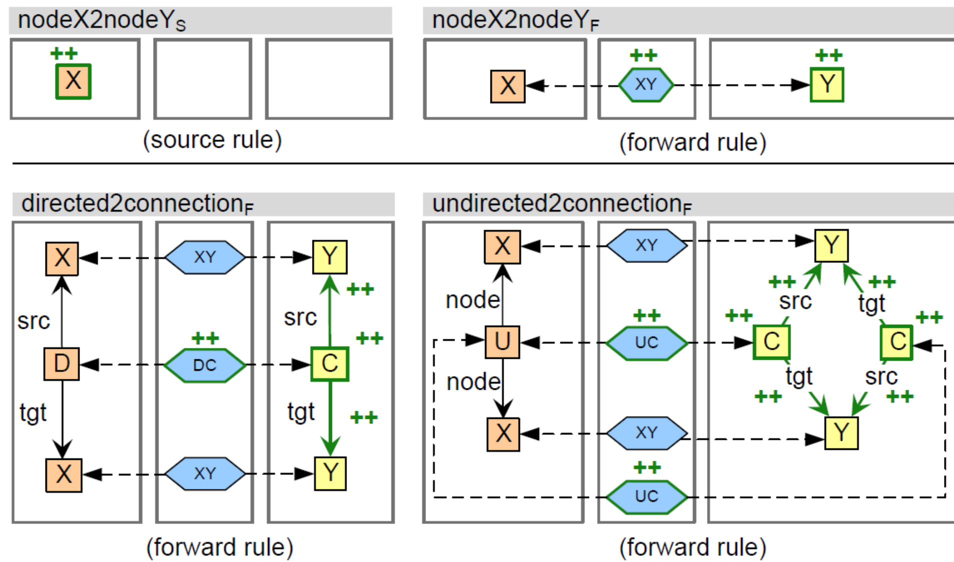


FIGURE 2.22 – Règle "avant" [64]

correspondent un seul nœud "C". Les nœuds "U" (connexion indirecte) correspondent chacun à plusieurs nœuds "C" du graphe cible. La seconde figure illustre la grammaire de graphes triples pour une transformation avant. La première règle triple **nodeX2nodeYs** crée les nœuds "X" du graphe source. La règle triple **nodeX2nodeYf** crée les nœuds "Y" du graphe cible à partir de nœuds pré-existants. La règle triple **directed2Connectionf** crée un nœud "C", un arc "src" et un arc "tgt" à partir d'un nœud "D" et de deux nœuds "Y" préexistants. Enfin la triple règle **undirected2connectionf** crée deux nœuds "C" et les arc associés à partir d'un nœud "U" et de deux nœuds "Y" dont nous dérivons les catégories $AIC-TrGraphs$ et $AIC-TrGraphs_{TG}$ des graphes triples composés d'AIC-Graphes définis en 25.

Applications des grammaires de graphes triples.

Les grammaires de graphes triples trouvent des applications en IDM pour la transformation de modèles, la synchronisation de modèles et en traçabilité des transformations de modèles. Elles ont été implémentées au travers des outils suivants pour permettre des transformations ou synchronisations de modèles basées sur des grammaires de graphes triples AGG 2.0 [65], Viatra [12] GReAT [66].

2.2.2 Conclusion sur les grammaires de graphes

Dans cette partie consacrée aux grammaires de graphes nous avons formalisé, à travers une étude de la littérature la notion de graphes typés avec racine et lien de contenance **(IC-Graphe**)et ses morphismes, ainsi que la notion de graphes typés attribués avec arbre d'héritage **(ATGI)** et ses morphismes. Nous avons établi le lien entre les transformations de graphes et les transformations de modèles. Enfin, nous avons passé en revue les définitions des grammaires de graphes triples permettant une formalisation des transformations de modèles telles que nous les pratiquons en IDM. L'ensemble de ces définitions doit nous permettre pour la suite de ce manuscrit, de formaliser la méthode de transformation de modèles proposée.

CHAPITRE 3

TRAÇABILITÉ

3.1 Traçabilité

La traçabilité est née dans les années 70 de la volonté des concepteurs de logiciels d'établir un lien entre les exigences formulées par les clients et la réalisation de la solution logicielle. Cependant, les champs d'applications de la traçabilité ont évolué pour devenir aujourd'hui des méthodes permettant de suivre le développement de logiciels dans un contexte plus général. Stefan Winkler et Jens Von Pilgrim dans "A survey of traceability in requirements engineering and model-driven development" [67] définissent deux champs principaux de la traçabilité appliquée au domaine logiciel :

- Dans le champ de l'ingénierie des exigences et plus généralement de l'ingénierie logicielle, la traçabilité permet d'assurer la validation et la vérification d'artefacts liés entre eux.
- Dans l'IDM, la traçabilité appliquée aux transformations de modèles permet de conserver la cohérence et de propager les modifications entre modèles dérivés les uns des autres.

Il existe bien sûr bien d'autres domaines dans lesquels les concepts de traçabilité sont utilisés. Nous pouvons citer par exemple l'ingénierie de la connaissance et le management de projets logiciels. Toutefois nous nous intéresserons ici à la traçabilité appliquée à l'ingénierie des exigences et plus particulièrement à la traçabilité appliquée aux transformations de modèles.

3.1.1 Définition et Terminologies

Les documents, les éléments de modèles ou bien encore les éléments de code sont souvent désignés par le terme d'artefact dans la littérature consacrée à la traçabilité [68], [69]. Nous utiliserons nous aussi dans cet exposé le terme d'artefact lorsque nous parlerons d'éléments reliés par un lien de traçabilité.

Gotel et FinKektein proposent une définition de la traçabilité dans [70] *". . the ability to describe and follow the life of a requirement, in both a forward and backward direction; i.e., from its origins, through its development and specification, to its subsequent deployment and use, and through periods of ongoing refinement and iteration in any of these phases."*

Cette définition définit la relation de traçabilité uniquement en lien avec les exigences. L'IEEE Standart Glossary of Software Engineering Terminology [71] généralise la définition de Gotel et Finkeltein pour définir la traçabilité entre artefacts quels que soit ces artefacts :

- *"The degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another; for example, the degree to which the requirements and design of a given software component match;"*
- *"The degree to which each element in a software development product establishes its reason for existing; for example, the degree to which each element in a bubble chart references the requirement that it satisfies."*

Enfin N. Aizenbud-Reshef et al. généralisent la définition de la traçabilité et considèrent dans [72] la traçabilité comme *"as any relationship that exists between artifacts involved in the software-engineering life cycle."*. Pour les auteurs, cette définition inclut :

- les liens explicites générés par des transformations (telle que la génération de code ou le reverse engineering),
- les liens calculés sur la base d'informations existantes,
- les liens déduits statistiquement sur la base d'historiques.

[67] précisent une terminologie de la traçabilité et définissent les termes de *Trace* et *lien de traçabilité*

Définition 23 (Trace) *Une trace est un élément d'information (implicite ou explicite) indiquant ou prouvant qu'un événement a existé ou s'est réalisé.*

Définition 24 (Lien de traçabilité) *Un lien de traçabilité est une relation utilisée pour relier des artefacts entre eux.*

3.1.2 Modèles de traçabilité

L'un des premiers modèles conceptuels de traçabilité a été proposé par Ramesh et Jarke dans [69]. Ce modèle montre les relations entre les différentes entités impli-

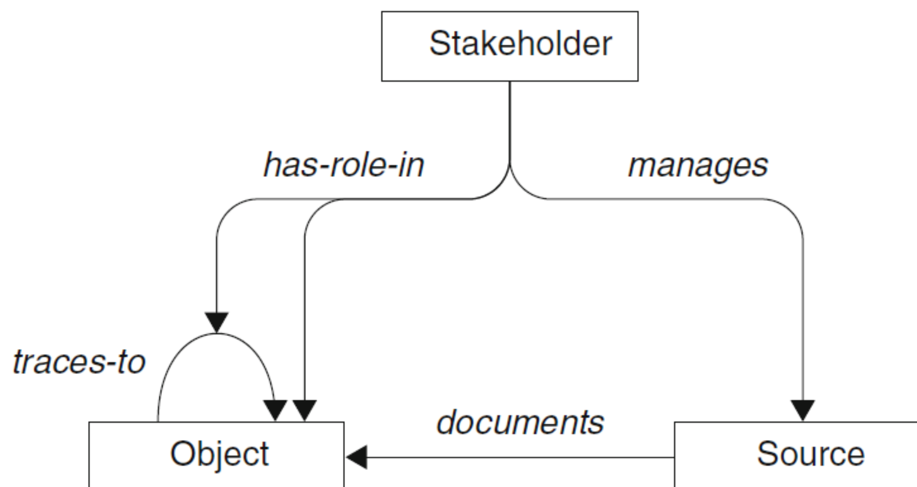


FIGURE 3.1 – Modèle conceptuel de traçabilité selon Ramesh et Jarke

quées dans un processus de traçabilité. En premier lieu les *Stakeholders* représentent les parties prenantes du processus de développement du logiciel. Les *Sources* représentent toutes les sources d'informations qu'elles soient formelles ou non. Ce sont par exemple les procédures, les documents formalisés, mais aussi les sources d'information non formelles comme les appels téléphoniques. Les *Objects* sont relatifs à tous les artefacts sur lesquels nous pouvons appliquer de la traçabilité, tels que les éléments d'exigences et les éléments de conception du logiciel. Ce modèle n'est pas un modèle formel au sens de l'IDM, mais il permet de poser les bases d'un premier raisonnement sur les relations entre les éléments mis en œuvre dans un processus de traçabilité. Ce modèle a été utilisé par le passé pour explorer les différentes formes et types de traçabilité.

Partant de ce modèle, et complété par une analyse de la littérature consacrée à ingénierie des exigences S. Winkler et J.V. Pilgrim proposent un modèle (figure 3.2) conceptuel plus exhaustif dans [67]. Ce diagramme organise les caractéristiques qu'un modèle de traçabilité doit posséder.

Au sommet de ce modèle, le *Context Container* stocke les méta-données relatives au contexte de la traçabilité tels que les méta-modèles sur lesquels la traçabilité est ap-

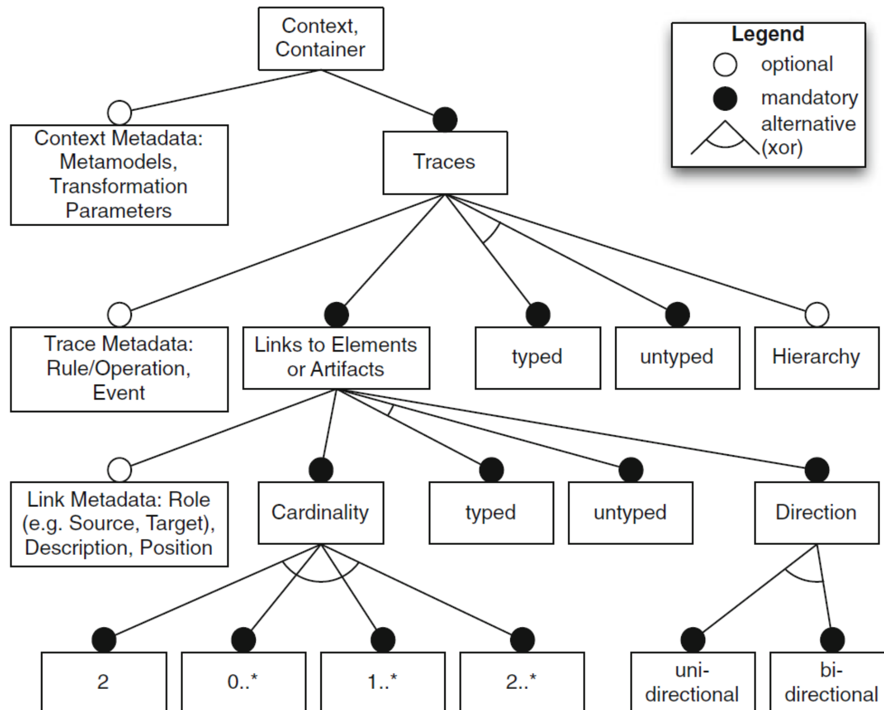


FIGURE 3.2 – Diagramme des caractéristiques de modèles de traçabilité de S. Winkler et J.V. Pilgrim

pliquées, les paramètres de transformation. De plus le contexte sert de conteneur pour le stockage des *Traces*. Les *Traces* peuvent être typées ou non. Elles peuvent aussi être représentées dans une structure hiérarchisée. Cependant, la caractéristique la plus importante des *Traces* concerne les liens entre les artefacts *Links to Elements or Artifacts*. En effet, une trace préserve un lien entre différents artefacts. Le diagramme 3.2 montre que ces liens peuvent être typés ou non et peuvent être unidirectionnels ou bidirectionnels.

S.Winkler et J.V. Pilgrim complètent leur modèle conceptuel en reprenant une classification des liens de traçabilité établie par Spanoudakis et Zisman dans [73]

- **Liens de dépendance :** un lien de dépendance entre deux artefacts e1 et e2 indique que e2 dépend de l'existence de e1. Une modification sur e1 implique une modification potentielle sur e2.
- **Liens de raffinement :** les liens de raffinement sont utilisés dans des hiérarchies d'abstractions pour décrire comment des artefacts complexes sont raffinés en artefacts plus petits, plus concrets ou plus utilisables.

- **Liens d'évolution :** les liens d'évolution sont utilisés quand un artefact en remplace un autre, par exemple lorsqu'une nouvelle version remplace une version plus ancienne, ou bien lorsqu'un ensemble d'exigences est remplacé par un ensemble d'exigences plus précises.
- **Lien de satisfaction :** les liens de satisfaction sont utilisés pour tracer un artefact avec un artefact amont dans le cycle de développement, par exemple un élément de conception satisfaisant une ou plusieurs exigences.
- **liens de chevauchement :** les liens de chevauchement peuvent être utilisés pour tracer deux artefacts relatifs à une caractéristique ou à aspect commun du système, par exemple tracer la représentation d'une exigence exprimée dans un langage naturel et dans un langage formel.
- **Liens de conflit :** les liens de conflit tracent un conflit existant entre deux artefacts. Pour être utiles, ces liens doivent contenir une information sur comment résoudre le conflit en donnant les alternatives possibles.
- **Liens de rationalisation :** les liens de rationalisation sont utilisés pour tracer les décisions prises concernant l'évolution d'un projet. Ils sont utilisés pour documenter la justification concernant la création ou l'évolution d'artefacts.
- **Liens de contribution :** Les liens de contribution tracent les différentes relations entre les parties prenantes et les artefacts.

3.1.3 La traçabilité dans l'IDM.

Comme nous l'avons vu précédemment, l'IDM utilise les modèles pour représenter les différents artefacts impliqués dans le développement d'un système complexe. Ces modèles peuvent être décrits avec différents langages et sur plusieurs niveaux d'abstraction. Dans ce contexte, la traçabilité joue un rôle clé pour préserver la cohérence entre les différents artefacts représentés dans une multitude de modèles. Pour stocker les liens de traçabilité, N.Drivalos et al. [74] [75] montrent que deux stratégies sont appliquées dans l'IDM :

1. **Le stockage intra-modèles des liens de traçabilité.** Les informations de traçabilité sont stockées avec les artefacts référencés. Ces informations prennent la forme d'éléments de modèle ou d'attributs.
2. **Le stockage inter-modèles des liens de traçabilité.** Les informations de traçabilité sont stockées dans un modèle tiers conforme à un méta-modèle définis-

sant une sémantique dédiée à la traçabilité.

Un exemple de traçabilité intra-modèle est proposé par [76]. Les auteurs utilisent des diagrammes UML et proposent une traçabilité basée sur un ensemble de dépendances stéréotypées, liant les artefacts entre eux.

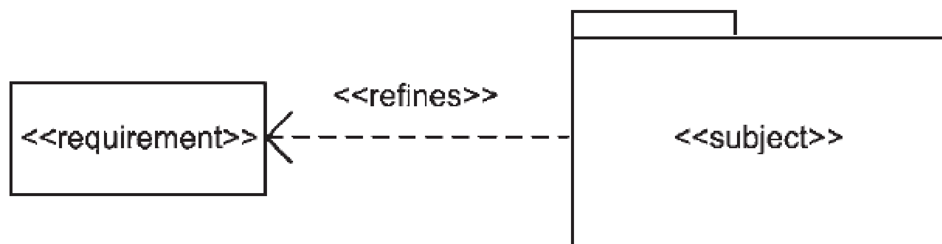


FIGURE 3.3 – Trace entre un package sujet et son exigence [76]

Cette stratégie de traçabilité a l'avantage d'une mise en œuvre rapide. Cependant lorsque les modèles se multiplient et deviennent plus complexes, se pose très vite la question de "où" stocker les liens de traçabilité. S'ils sont stockés dans le modèle source, alors ils ne sont pas visibles par le modèle cible (et réciproquement). Si les liens de traçabilité sont stockés par les modèles sources et cibles, il faut s'assurer du maintien de la cohérence de ces liens lorsque les modèles évoluent [72]. De plus, préserver les informations de traçabilité implique de modifier la sémantique des modèles concernés pour y inclure les informations de traçabilité. Cela induit une "pollution" au sens de [74]. Cette "pollution" rend moins compréhensible les modèles d'origine. Il devient parfois difficile de distinguer les informations de traçabilité des informations propres aux systèmes modélisés. Enfin les analyses automatiques des liens de traçabilité deviennent plus difficiles à mettre en œuvre à mesure que les modèles augmentent en complexité.

Une stratégie de traçabilité inter-modèle implique de définir un méta-modèle tiers possédant une sémantique uniquement dédiée à la traçabilité. Les liens de traçabilité référencent les artefacts des modèles sources et cibles sans les modifier. Les modèles ne sont pas "pollués" par des informations annexes. Avoir une sémantique de traçabilité dédiée permet aussi de réaliser des analyses de traçabilité automatiques plus aisées et cela alors même que les systèmes modélisés deviennent complexes. Cependant [74] note que le pré-requis à un stockage inter-modèle est l'existence d'un identifiant unique pour chaque artefact impliqué dans la traçabilité.

Dans cet esprit, N .Mustaf et Y. Labiche proposent dans [77] un méta-modèle générique de traçabilité figure 3.4. L'objectif de N .Mustaf et Y. Labiche est de proposer

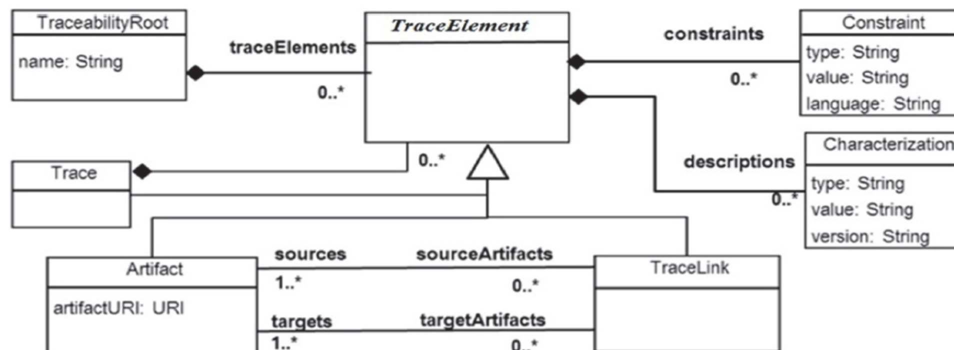


FIGURE 3.4 – Méta-modèle générique de traçabilité de N .Mustaf et Y. Labiche [77]

une sémantique générique capable d'assurer la traçabilité entre les différents artefacts d'un système hétérogène. La *TraceabilityRoot* est la méta-classe racine du système de traçabilité, elle regroupe les informations du système. Le système de traçabilité est composé d'un ensemble de *TraceElement* :

- *Trace*
- *Artifact*
- *TraceLink*.

Chaque *Trace* du système de traçabilité associe deux *Artifact* (source et target) par l'intermédiaire d'une *TraceLink*. A chaque élement de trace(*TraceElement* est associé une *Characterization* et des *Constraint* permettant de caractériser chaque élément suivant une taxonomie qui lui est propre. L'intérêt de la solution proposée par [77] est de pouvoir établir un système de traçabilité applicable quel que soit le système d'application. La solution a l'avantage d'une facilité de mise en œuvre et offre une standardisation de la traçabilité (tous les modèles sont conformes au même méta-modèle de traçabilité). Cependant comme le notent N. Drivalos et al. dans [75], un modèle générique peut potentiellement établir des liens entre des artefacts sans que cela soit légitime. Pour éviter cette situation, Drivalos et al. proposent d'établir un méta-modèle de traçabilité spécifique à chaque transformation de modèle. Le méta-modèle de traçabilité proposé dans [75] est un méta-modèle fortement typé doté d'une sémantique riche. La figure 3.5 présente un méta-modèle de trace. Comme dans le méta-modèle précédant, les informations générales sont regroupées dans une classe racine *TraceModel*. Par contre ici, chaque lien de traçabilité est fortement typé. Chaque type de lien est

uniquement utilisé pour établir la traçabilité entre deux artefacts de types prédéfinis. Ainsi le lien de traçabilité *ComponentPackageTraceLink* établit des traces entre les instances de la classe *Package* des modèles *ClassMetamodel* et les instances de la classe *Component* des modèles *ComponentMetamodel*

FIGURE 3.5 – Méta-modèle de traçabilité dédié aux liens entre *ClassMetamodel* et *ComponentMetamodel* [75]

Ici un ensemble de contraintes inter-modèles peuvent être mises en place, par exemple pour s'assurer qu'un seul et unique objet *Package* est lié à un seul et unique objet *Component*. Ces contraintes sont exprimées dans le langage Epsilon Validation Langage (EVL). Nous pouvons noter que le méta-modèle de traçabilité fortement typé doté d'une sémantique riche développé par [75] est proche du concept de graphe de correspondance des grammaires de graphes triples. Un autre exemple de méta-modèle s'inspirant de la grammaire de graphe triple est proposé par Zinovy Diskin et

al. [78]

3.1.4 Conclusion sur la traçabilité

Dans ce chapitre nous avons évoqué différentes approches de traçabilité. Au vue de l'approches de formalisation des transformations de modèles que nous utilisons, l'approche de traçabilité proposée par N.Drivalos et al. dans [75] nous semble la mieux adaptée à notre application de transformation de modèles. En effet les concepts et les liens fortement typés du méta-modèle de traçabilité, ont une proximité avec le graphe de correspondance de la grammaire des graphes triples, présenté au chapitre précèdent. Ils permettent de créer un système de traçabilité non ambiguë.

Dans la seconde partie de ce manuscrit nous allons présenter une chaîne de transformation de modèles permettant de transformer des modèles d'ingénierie vers des modèles dotés d'une sémantique d'exécution à des fins de simulation. Nous utiliserons les grammaires de graphes pour formaliser notre démarche. Enfin la traçabilité étant un élément essentiel pour établir le lien entre les concepts des modèles sources et le modèle de simulation, nous utiliserons un méta-modèle de traçabilité tel que proposé par N.Drivalos et al. dans [75].

DEUXIÈME PARTIE

État de l'art

CHAPITRE 4

TRANSFORMATION ET EXÉCUTION DE MODÈLES

4.1 Définition d'une sémantique en IDM pour la vérification de modèles

La problématique abordée dans cette thèse concerne la simulation du comportement de modèles d'ingénierie systèmes, à des fins de vérification, lesquels ne possèdent pas de sémantique exécutable. En ce sens, Benoît Combemal et al. proposent dans *"Essay on Semantics Definition in MDE, An Instrumented Approach for Model Verification"* [47] une taxonomie des approches pour l'exécution d'un Domaine de Langage Spécifique (DSL). Ce travail a abouti à la mise en évidence de trois approches :

- par la définition d'une sémantique reposant sur des axiomes, appelée : *sémantique axiomatique*,
- par l'extension du DSL avec une sémantique opérationnelle, appelée : *sémantique opérationnelle*,
- par la transformation du DSL vers un DSL doté d'une sémantique exécutable (*axiomatique* ou *opérationnelle*), appelée *sémantique translationnelle*.

La première approche (*"sémantique axiomatique"*) consiste à définir un ensemble de propriétés sous la forme de pré-condition P, d'instruction I et de post-condition Q. Le modèle pour être valide doit satisfaire l'ensemble des axiomes $\{P\}I\{Q\}$ aux différentes étapes de son exécution. L'approche *sémantique opérationnelle* étend le DSL avec des informations permettant de décrire l'état du modèle lors de son exécution. Benoît Combemale et al. ont identifié plusieurs techniques. La première consiste à étendre le langage de méta-programmation pour exprimer la sémantique comportementale, en ajoutant un ensemble d'opérations relatives à chaque concept. Ces opérations peuvent être exprimées par exemple à l'aide de Kermeta [79], ou le MOF Action

Langage [80]. Une seconde technique consiste à réaliser une série de transformations endogènes à partir du DSL source. Ceci permet de transformer l'état du modèle. Cette seconde technique a largement été implémentée par les techniques de grammaire de graphes [48].

La dernière approche décrite par les auteurs comme *sémantique translationnelle* transforme un modèle conforme à un DSL source en un modèle conforme à un DSL cible doté d'une sémantique d'exécution (*axiomatique* ou *opérationnelle*).

José E. Rivera ans Antonio Vallecillo proposent dans [81] d'ajouter une sémantique comportementale aux modèles en utilisant le langage Maude [82]. En ce sens ils suivent l'approche (*sémantique axiomatique*) de la taxonomie de [47]. Les auteurs développent sous Maude leur méta-modèle et l'étendent avec un ensemble de règles d'exécution suivant une sémantique axiomatique. Si cette approche permet de définir une sémantique comportementale formelle, la solution semble moins bien adaptée lorsqu'on considère des systèmes complexes exprimés sous forme de modèles hétérogènes. L'interconnexion des axiomes devenant plus complexe à mettre en œuvre à mesure que le système à modéliser se complexifie.

Dans [83], Benoît Combemale et al. proposent une méthode et des outils "GEMOC" [84], pour exécuter des modèles avec des sémantiques hétérogènes. Dans leur article les auteurs suivent l'approche (*sémantique opérationnelle*). Ils étendent le metamodèle de Capella afin d'y ajouter une sémantique opérationnelles pour simuler le comportement des diagrammes de data-flow et des machines à états (appelé "mode automata" dans l'article). Le méta-modèle doté d'une sémantique exécutable est appelé xCapella.

La figure 4.1 montre l'architecture de xCapella dans l'environnement GEMOC. Les méta-modèles Capella, *"Capella Data Flow (EMF)"* et *"Capella Mode Automata (EMF)"* sont étendus à l'aide de fonctions et de données d'exécution *"Execution Data & Functions"*. Chaque méta-modèle étendu a son propre moteur d'exécution *"Execution engine "* l'exécution concurrente de l'ensemble étant synchronisé par le *"Heterogenous Coordination Engine"*. L'environnement d'exécution de GEMOC permet de conserver la trace d'exécution de l'espace des états possibles du système concurrent. Il est alors possible d'analyser les différents chemins d'exécution. La figure 4.2 montre un exemple de l'environnement d'exécution de GEMOC.

L'approche proposé par Benoît Combemale et al. permet en suivant l'approche (*sémantique opérationnelle*) de définir une sémantique d'exécution sur des modèles hété-

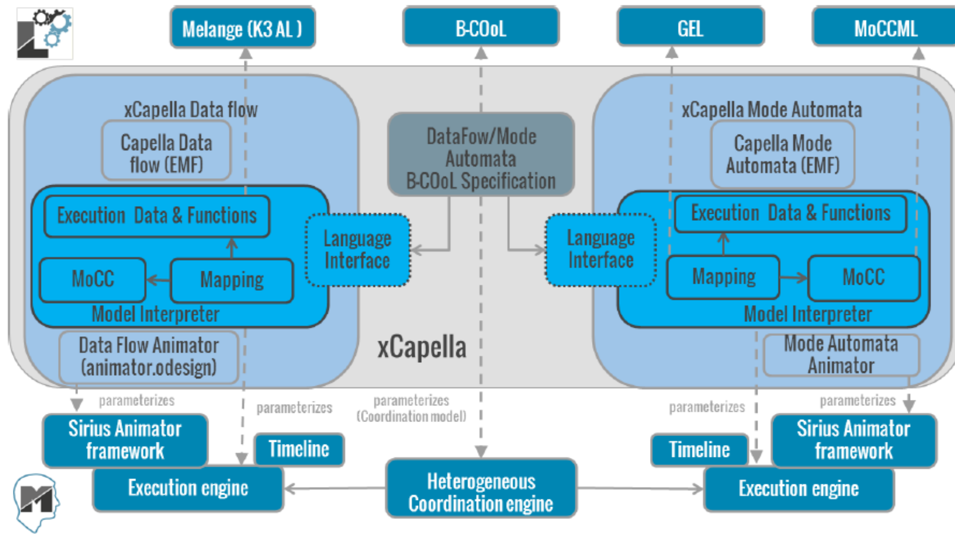


FIGURE 4.1 – xCapella dans l'environnement GEMOC (issu de [84])

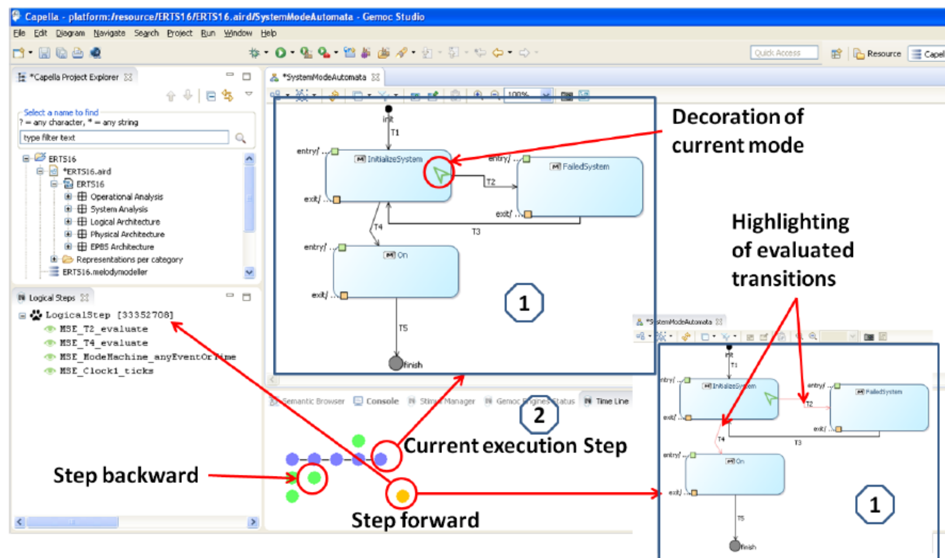


FIGURE 4.2 – GEMOC Workbench (issu de [84])

rogènes tout en proposant une traçabilité de l'exécution. Cependant cette approche impose de fixer la sémantique comportementale de l'environnement de modélisation. Or comme nous l'avons abordé lors de l'exposé de notre problématique, la grande diversité des systèmes modélisés implique de garder une grande expressivité de l'outil de modélisation. Par conséquent, un grand nombre de points de variation sémantique ne peuvent être fixés qu'au dernier instant en fonction du système modélisé. Nous conser-

verons de cette approche l'idée forte de s'appuyer sur les concepts source pour définir la sémantique cible. Dans la thèse, nous utiliserons la troisième approche *l'approche translationnelle* . En effet, cette approche permet de garder une séparation entre les langages de modélisation système et les langages de simulation. Le domaine de modélisation garde alors une sémantique descriptive dédiée à la spécification du système étudié et le domaine de simulation une sémantique exécutable dédiée à la simulation et à l'analyse du système ou d'une partie du système étudié. Cette séparation des préoccupations permet de choisir le domaine d'exécution en fonction d'objectifs spécifiques de simulation et de vérification. Cependant pour être interprété dans le domaine d'exécution les concepts du modèle source doivent être arrangés, transformés et adaptés. De plus un lien formel doit être établie entre les concepts sources et cibles pour pouvoir interpréter sans biais les résultats de la simulation. Dans la section suivante, nous présentons des exemples de transformations de modèles de système complexe vers des domaines de simulation.

4.2 Transformation de modèles de système vers un environnement de simulation

L'intention des approches présentées dans ce chapitre est de simuler le comportement d'un système décrit dans un langage de modélisation n'ayant pas de sémantique d'exécution. Cette intention peut être classée suivant la taxonomie proposée par [27] (cf chapitre 1.3.3) comme « semantic definition ».

Selon notre connaissance de la littérature, il est possible de classer les transformations de modèles de système vers un environnement de simulation suivant trois approches :

- **Par transformation de modèles directe**. Les concepts du domaine de modélisation sont alignés directement avec les concepts du domaine de simulation.
- **Par une extension du langage de modélisation suivie d'une transformation de modèles.** Le langage de modélisation est étendue pour que les concepts de modélisation puissent être alignés avec les concepts du langages de simulation
- **Par une chaîne de transformations de modèles** Les concepts du langage de modélisation sont arrangés par des transformations de modèles successives, pour pouvoir être aligné avec les concepts de simulation.

4.2.1 Transformation directe du modèle vers le domaine de simulation

Andrea Sindico et al. proposent dans [85] un framework permettant une simulation complète du système modélisé en SysML, et des communications avec son environnement (figure 4.3. L'article décrit uniquement la transformation d'un modèle du sous-système fonctionnel SysML vers un modèle de simulation MATLAB/Simulink [86].

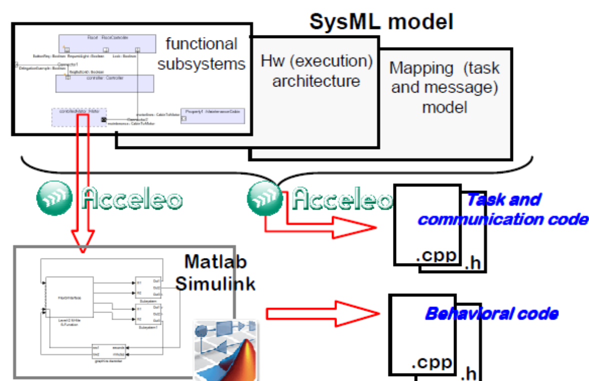


FIGURE 4.3 – Génération de code MATLAB/Simulink à partir de SysML proposé par [85]

Les auteurs proposent de transformer le modèle d'un sous-système fonctionnel modélisé avec SysML en un modèle Simulink. Dans cette approche, le langage SysML n'est pas étendue comme dans les approches précédentes. Les auteurs utilisent les *« Definition Block Diagram »* (ddb) et les *« Internal Block Diagram »* (idb) de SysML pour modéliser le système. Les auteurs réalisent ensuite une transformation directe du sous-système fonctionnel vers un script MATLAB. La transformation est réalisée grâce à un ensemble de template Acceleo [87]. Le comportement de chaque bloc pour la simulation est ensuite renseigné manuellement. Dans cette approche une première phase de sélection est réalisée (sélection du sous-système fonctionnel). Le modèle SysML est réalisé de façon à permettre un alignement direct des concepts vers les concepts SysML par exemple les noms des blocks SysML doivent être conformes à la syntaxe MATLAB/Simulink. Enfin une phase d'ajout et d'adaptation est réalisée manuellement pour permettre la simulation du modèle dans Simulink. La traçabilité entre les concepts source et cible est limitée aux seuls noms des concepts.

4.2.2 Extension du langage de modélisation suivie d'une transformation de modèles

Une approche communément suivie consiste à étendre le langage de modélisation pour pouvoir ensuite aligner directement les concepts de modélisation avec les concepts du langage de simulation [88]. Par exemple SysML est étendu pour un alignement vers Modelica [89] avec *« SysML4Modelica »* [90] [91] [92] ou vers MATLAB/Simulink avec *« SysML4Simulink »* [93] [94]. Nous présentons deux exemples de cette approche : un exemple de transformation SysML vers Simulink et un exemple de transformation SysML vers Modelica

Bassim Chabibi et al. proposent dans [93] de générer des modèles MATLAB/Simulink à partir de modèles SysML. L'idée de Bassim Chabibi et al. est dans un premier temps d'étendre le langage SysML pour pouvoir ajouter dans le modèle source des informations nécessaires à une simulation dans MATLAB/ Simulink. Cette opération est réalisée par l'ajout de trois stéréotypes SysML (cf figure 4.4). Le profile optenu est appelé *« SysML4Simulink »*.

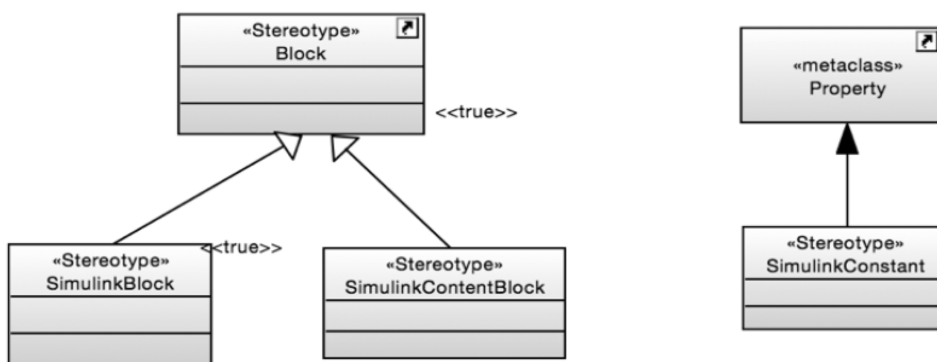


FIGURE 4.4 – Extention de SysML pour SysML4Simulink [93]

- le stéréotype **« SimulinkConstant »** permet de créer une distinction entre des variables et des constantes. Cette indication permet de préciser si une valeur doit rester constante lors de chaque simulation. Ce stéréotype hérite de la métaclasse « Property » de sysML,
- le stéréotype **« SimulinkBlock »** permet une description du modèle source en des termes permettant un alignement directe avec le concept de « Subsystem block » de simulink,
- le stéréotype **« SimulinkContenBlock »** permet un alignement direct avec le

concept de « System » de simulink.

Le comportement du système est modélisé avec les stéréotypes « constraint block » de SysML à l'aide d'équations dynamiques. Dans un second temps Bassim Chabibi et al., proposent une transformation de modèles de « SysML4Simulink » vers MATLAB/Simulink via une transformation de modèles vers un script Matlab généré par des templates Acceleo . (figure 4.5).

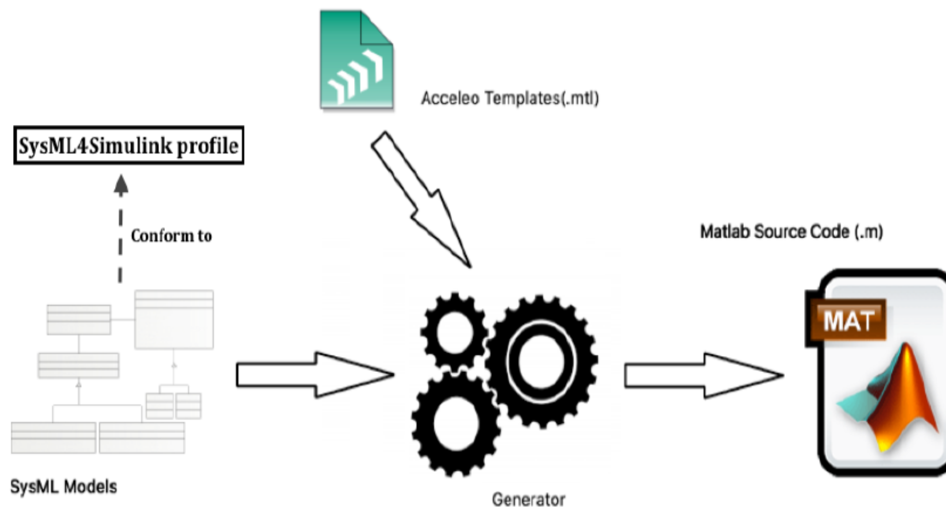


FIGURE 4.5 – Génération de code MATLAB/Simulink à partir de SysML proposé par [93]

La transformation de modèles est facilitée par « SysML4Simulink » qui permet un alignement direct des concepts de ce langage vers les concepts Simulink. L'approche présentée ne comporte pas d'élément de traçabilité en dehors de la conservation des noms de concepts.

Les auteurs appliquent leur approche sur un modèle de circuit électrique développé sous SysML. Ce modèle est ensuite transformé et simulé sous MATLAB. Dans cette approche, la sémantique du modèle source est fortement contrainte par le langage « SysML4Simulink ». Le système est modélisé a priori, avec l'intention de le simuler avec Simulink. Sans étape d'adaptation du modèle source, le nommage des concepts dans « SysML4Simulink » doit être conforme à la syntaxe de Simulink. L'étape d'ajout est implicite dans cette approche mais nécessaire pour définir par exemple les typages des données de simulation.

Thomas Johnson et Christiaan J.J. Paredis proposent dans [95] et [91] de transformer un modèle SysML vers un modèle Modelica. Les auteurs étendent dans un

premier temps SysML pour permettre de modéliser les informations nécessaires aux objectifs de simulation puis utilisent une transformation de modèles basée sur une grammaire de graphes triples implémentée avec le Framework de transformation Viatra [96]. (cf figure 4.6).

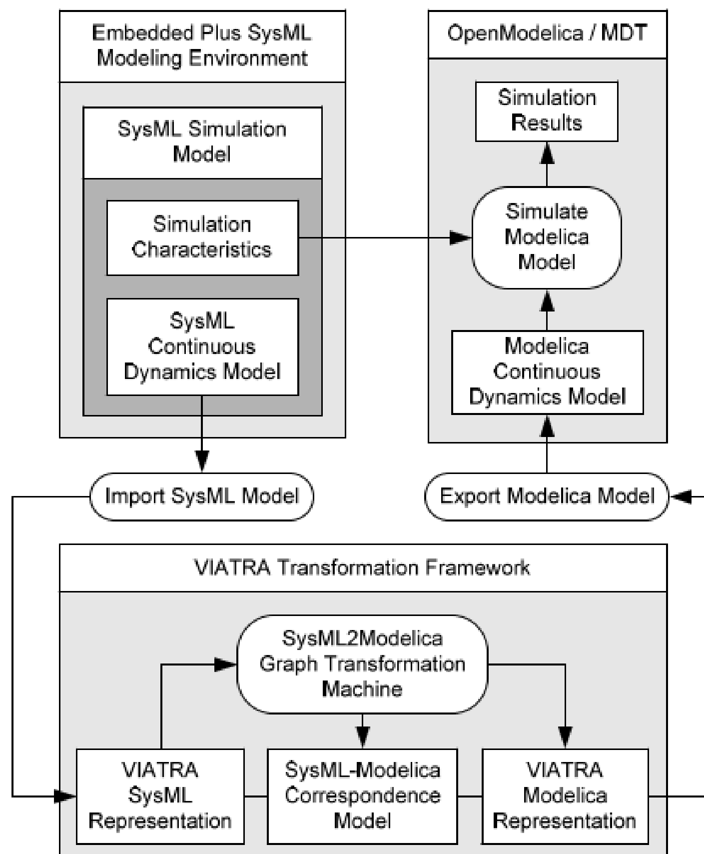


FIGURE 4.6 – transformation de modèles SysML vers Modelica, proposée par [91]

Les auteurs appliquent leur méthode sur la modélisation et la simulation d'une suspension de voiture. Dans cette approche, comme dans l'approche précédente la sémantique du modèle source est fortement contrainte par le langage« SysML4Modelica ». Le système est modélisé a priori, avec l'intention de le simuler avec Modelica. La différence avec l'approche de Bassim Chabibi et al. est l'utilisation de la grammaire de graphes triples (cf chapitre 2.2.1) pour réaliser la transformation de modèles.

Le graphe de correspondance (figure 4.7) permet alors d'assurer la traçabilité entre les concepts source et cible, par la création d'un méta-modèle de traces fortement typé tel que présenté par [75] (cf chapitre 3.1.3).

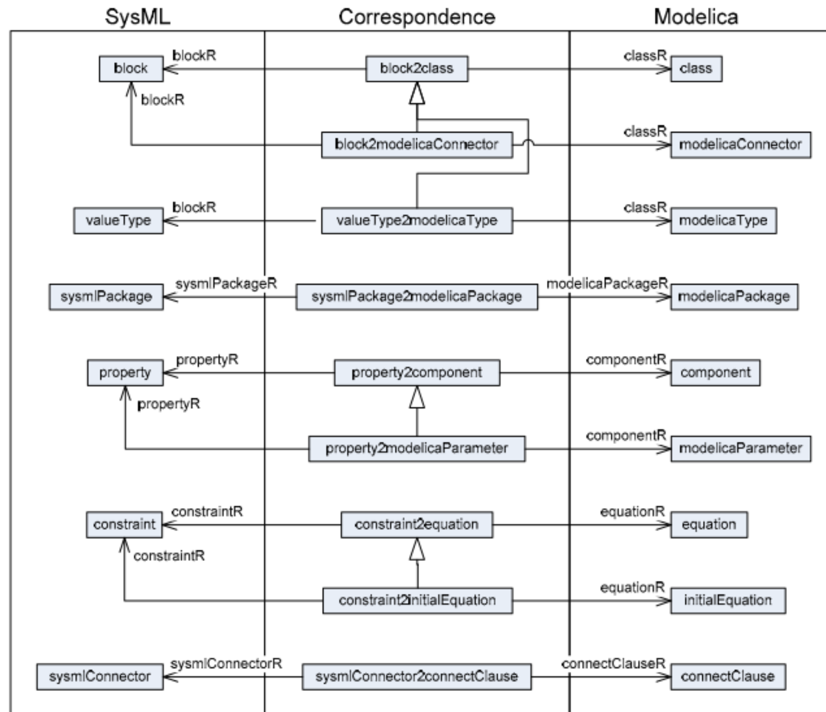


FIGURE 4.7 – Grammaire de graphe triple simplifiée SysML vers Modelica, proposée par [95]

4.2.3 Chaîne de transformations de modèles

Daniel Chaves Cafe et al. [97] proposent de simuler un modèle SysML contenant des diagrammes écrits dans des sémantiques hétérogènes. La simulation est réalisée avec le langage SystemC-AMS [98]. Pour ce faire, les auteurs définissent une chaîne de transformation de modèles composée de deux transformations 4.8. La première est une transformation de modèle à modèle qui permet de réorganiser les concepts afin qu'ils soient conformes au méta-modèle SystemC-AMS. Cette transformation correspond à l'intention *« Langage Translation »* de la taxonomie de [27]. La seconde est une transformation de modèle à texte qui permet de générer le code exécutable en langage SystemC-AMS. Cette seconde transformation permet d'adapter et d'enrichir le modèle source afin qu'il soit exécutable (intention *« Langage Translation »*).

Daniel Chaves Cafe et al. illustrent leur approche par la simulation d'un modèle SysML composé de deux sous-systèmes concurrents ; l'un ayant un comportement en temps continu et l'autre suivant une sémantique de machine à états finis. Dans l'application, une partie du comportement du sous-système continu est annotée sous forme

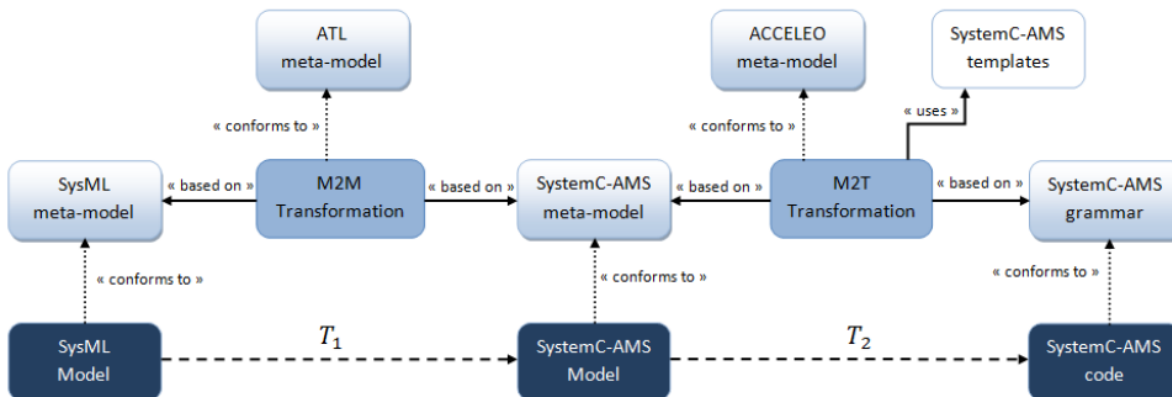


FIGURE 4.8 – Chaîne de transformations de modèles proposée par [97]

d'équation dans des blocks de stéréotype *"constraint"*. L'adaptation et l'enrichissement sémantique du modèle sont réalisés par des objets de la classe "Adaptator" du méta-modèle SystemC-AMS.

Ainsi, [97] réalisent une première phase implicite de sélection des concepts nécessaires à la simulation. Puis la chaîne de transformation permet la réorganisation, l'alignement, l'adaptation et l'enrichissement du modèle source pour le traduire en un code exécutable. Les auteurs ne traitent pas dans leur article la problématique de la traçabilité.

4.3 Synthèse des approches

Dans les travaux cités on retrouve cinq phases de transformation afin de transformer un modèle d'ingénierie en modèle exécutable. Il est nécessaire en premier lieu de réaliser une sélection des concepts à analyser. Toutes les approches précédentes considèrent de manière plus ou moins implicite un sous-ensemble limité de SysML. Vient ensuite une opération d'alignement pour définir comment les concepts sources sont traduits en concepts cibles. Lorsque l'alignement est réalisé directement, il est considéré implicitement que les concepts du domaine cible existent (au nom près) dans le domaine source. Cependant dans le cas général, il est nécessaire d'ajouter avant l'alignement, une opération de réorganisation, pour organiser les concepts sources en concepts transférables dans le domaine cible. Enfin les opérations d'enrichissement et d'adaptation permettent d'ajouter des informations de sémantiques opérationnelles

absentes dans le modèle source et de les adapter pour que le modèle obtenu soit pleinement conforme au domaine cible et puisse être exécuté. [47] démontre la nécessité de créer une relation d'équivalence entre les concepts des modèles sources et cibles. Si cette relation d'équivalence n'est pas établie, les résultats de la simulation obtenus dans le domaine cible ne peuvent pas être interprétés dans le domaine source. Toutes les approches étudiées dans ce chapitre sont adaptées mais aussi limitées à un seul domaine cible. La plupart des approches réalisent les différentes opérations de définition de sémantique en une seule transformation. Enfin le lien de traçabilité entre les concepts sources et cibles n'est pas toujours établi.

Dans le chapitre suivant nous proposons et formalisons une méthode de transformation de modèles, afin de définir une sémantique d'exécution pour un modèle d'ingénierie système. Pour ce faire, nous définissons une chaîne de transformation impliquant les intentions "Restrictive query", "Model refactoring", "Translation", "Translational Semantic", "Simulation" ou "Synthesis", au sens de [27].

TROISIÈME PARTIE

Proposition et application

CHAPITRE 5

MÉTHODE DE TRANSFORMATION

5.1 Introduction a ModelRun

5.1.1 Principes généraux

Dans ce chapitre nous proposons et formalisons une méthode à base de transformations de modèles permettant de traduire un DSML possédant une sémantique descriptive d'un modèle d'ingénierie système, vers un DSML doté d'une sémantique opérationnelle, permettant la simulation et la vérification de propriétés du modèle d'ingénierie source. Notre méthode appelée ModeRun est composée de 5 étapes de transformation (cf figure : 5.1) :

1. **La sélection** définit le sous-ensemble du modèle d'ingénierie pour lequel nous voulons vérifier des propriétés.
2. **La réorganisation** prépare les concepts sélectionnés pour qu'ils puissent être alignés avec les concepts du domaine cible.
3. **L'alignement** traduit les concepts du DSML source sélectionné en concepts du DSML cible.
4. **L'enrichissement** ajoute les informations et concepts inexistants dans le domaine source, mais nécessaires au domaine cible.
5. **L'adaptation** arrange les concepts pour que le modèle soit conforme au langage défini par le méta-modèle du domaine cible.

Afin que les conclusions établies sur le modèle de simulation puissent se reporter sur le modèle d'ingénierie source, il est important d'établir un lien fiable entre les concepts du modèle source et les concepts du modèle cible. C'est pourquoi nous complétons notre méthode de transformation par une méthode de traçabilité des transformations de modèles.

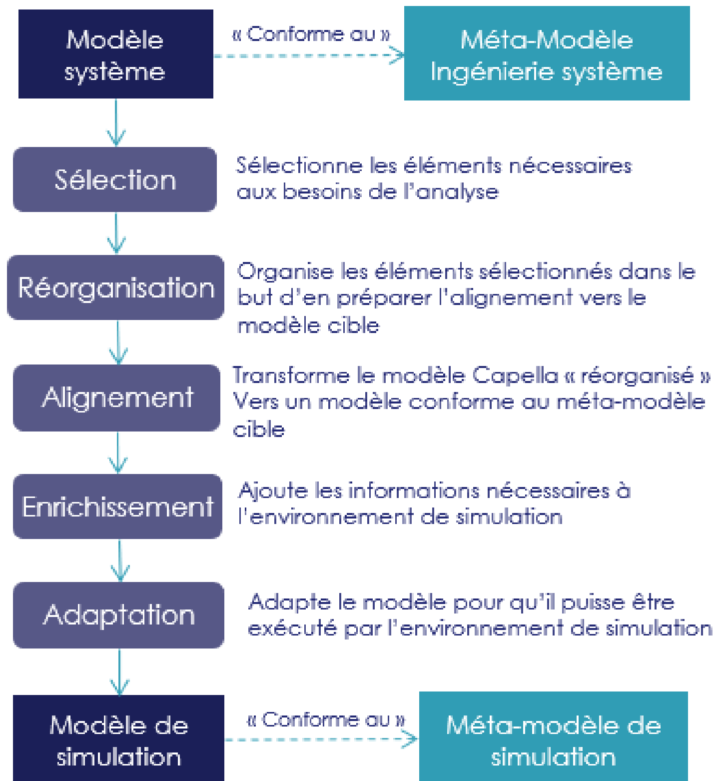


FIGURE 5.1 – Méthode de transformation

5.1.2 Formalisation des étapes de transformation

Nous avons dans le chapitre 2.1 défini d'un coté les graphes avec héritage et lien de contenance (définition 6) et de l'autre les E-graphes et les graphes attribués (respectivement définitions 10 et 13). Cependant nous utilisons dans notre application des modèles ECore composés de liens d'héritages, de liens de contenance et d'attributs. Nous proposons dans la définition suivante d'unir ces deux définitions pour définir des graphes attribués avec héritages et liens de contenances soit en abrégé des AIC-graphes.

Rappel de la définition 6 : Le tuple $G_{IC} = (T,\ I,\ Vabs,\ C)$ est appelé graphe avec relations d'héritage et liens de contenance, en abrégé IC-graphe. où

- T est un graphe $T = (V,\ E,\ s,\ t)$,
- I est l'ensemble des arbres d'héritage $I = (I_V,\ I_E,\ s,\ t)$ avec $I_V = V$ et $I_E \cap E = \varnothing$,
- $Vabs$ un ensemble $Vabs \subseteq V$ de nœuds abstraits,

- C un ensemble $C \subseteq E$ d'arcs de contenance.

Définition 25 (AIC-Graphe) *Le tuple* $G = (G_{IC}, V_D, E_{NA}, source_{NA}, target_{NA})$*, un IC-graphe attribué, en abrégé AIC-graphe avec :*
- $G_{IC} = (T,\ I,\ Vabs,\ C)$*, un IC-Graphe,*
- V_D *l'ensemble des nœuds attribués tel que* $V \cap V_D = \varnothing$*,*
- E_{NA} *l'ensemble des arcs de nœuds d'attribués tel que* $E \cap E_{NA} = \varnothing$*,*
- $source_{NA}$ *la fonction source qui associe un nœud à un arc de relation d'attribut* $source_{NA} : E_{NA} \to V$*,*
- $target_{NA}$ *la fonction cible qui associe un arc de relation d'attribut à un nœud attribut.* $target_{NA} : E_{NA} \to V_D$.

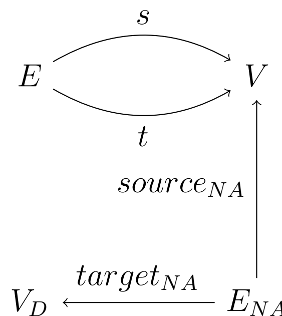


FIGURE 5.2 – Graphe attribué

Comme nous l'avons évoqué dans le chapitre 2.1, l'ajout d'un nœud d'attribut V_D au graphe n'est pas suffisant pour que le graphe devienne un graphe "attribué". Il faut aussi ajouter à ce nœud des données. Pour cela, nous combinons à ce graphe, une algèbre sur des signatures de données [57] (cf définitions 11 et 12) Dans ce chapitre nous considérerons désormais des AIC-graphes.
Note : A la différence de la définition du E-graphe proposé par Juan de Lara et al. [53], nous ne considérons que les nœuds de données associés aux nœuds du IC-Graphe. Ainsi dans nos applications les arcs ne sont pas attribués.

Les étapes de transformations sont illustrées par des graphes dont la figure 5.3 donne la sémantique des symboles utilisés dans nos représentations graphiques.

Enfin pour ne pas surcharger les représentations graphiques, les cardinalités des relations entre nœuds ne sont pas représentées.

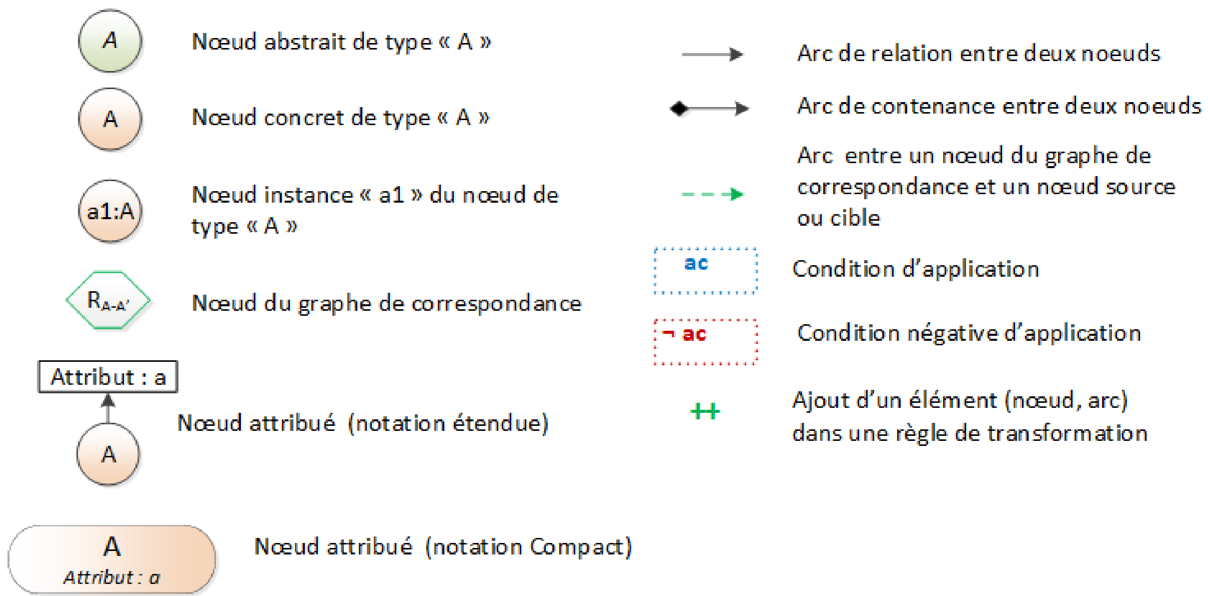


FIGURE 5.3 – Légende des symboles utilisés pour la représentation des AIC-graphes

5.1.3 Traçabilité des étapes de transformations

Nous proposons de créer une chaîne de 5 transformations $G^{source} \overset{G^{correspondance}_{selection}}{\Longrightarrow} G^{selection} \overset{G^{correspondance}_{reorganisation}}{\Longrightarrow} G^{organise} \overset{G^{correspondance}_{alignement}}{\Longrightarrow} G^{aligne} \overset{G^{correspondance}_{enrichissement}}{\Longrightarrow} G^{enrichi} \overset{G^{correspondance}_{adaptation}}{\Longrightarrow} G^{cible}$. Les graphes sont liés entre eux par des graphes de correspondance. Ces graphes de correspondance créent une chaîne de traçabilité entre les concepts sources et leurs correspondances dans le graphe cible. L'implémentation de ces graphes de correspondance peut ce faire à l'aide d'un modèle de classes fortement typées tels que proposé par [75] (cf chapitre 3.1.3). Il est alors possible de remonter la chaîne de transformation pour chaque instance du modèle cible et d'associer le ou les instances du modèle source qui en sont à l'origine. Seul les informations ajoutées pendant la phase d'enrichissement n'ont pas de correspondance dans le modèle source. Il est cependant important de les tracer comme éléments d'enrichissement exogènes au modèle source.

5.1.4 Exemple de cas d'application

Dans la suite de la présentation nous utiliserons un cas d'utilisation qui servira de fil rouge pour illustrer chaque étape de notre chaîne de transformation. Le langage

source de notre exemple permet de spécifier et modéliser un atelier de production industriel avec des lignes de production, des flux de matières et des opérateurs. Le méta-modèle figure 5.4 donne la syntaxe abstraite de ce langage.

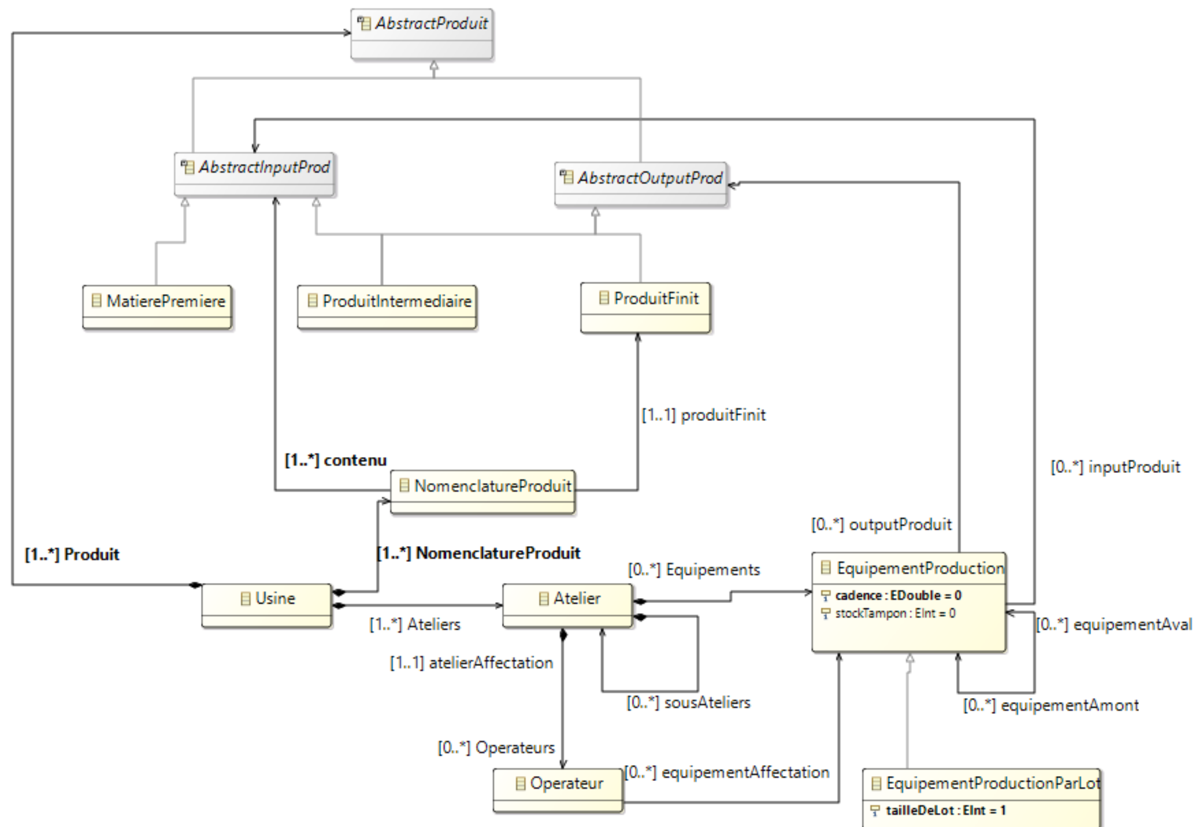


FIGURE 5.4 – méta-modèle modélisation des flux de production d'une usine

Le concept racine du méta-modèle est l'« *Usine* ». L'usine contient un ensemble d'« *Ateliers* » qui peuvent être des « *sous-ateliers* ». Aux ateliers sont attachées des équipements de production ainsi que des « *Opérateurs* ». Les équipements peuvent réaliser une production en flux (« EquipementProduction ») ou par lots (« Equipement-DeProductionParLot »). Ils portent une information de cadence de production en unité par minute et une information permettant de dimensionner un stock tampon en amont de l'équipement. Les équipements de production par lots possèdent une information sur la taille du lot de production. Le concept de « *NomenclaturesProduits* » est attaché au concept « *Usine* », il est composé d'un « *ProduitFinit* », de « *ProduitIntermédiaire* » et de « *MatièrePremière* ». La figure 5.5 propose un exemple de syntaxe concrète permettant une représentation de modèle d'unité de production. La figure 5.6 montre un

exemple d'utilisation de cette syntaxe.

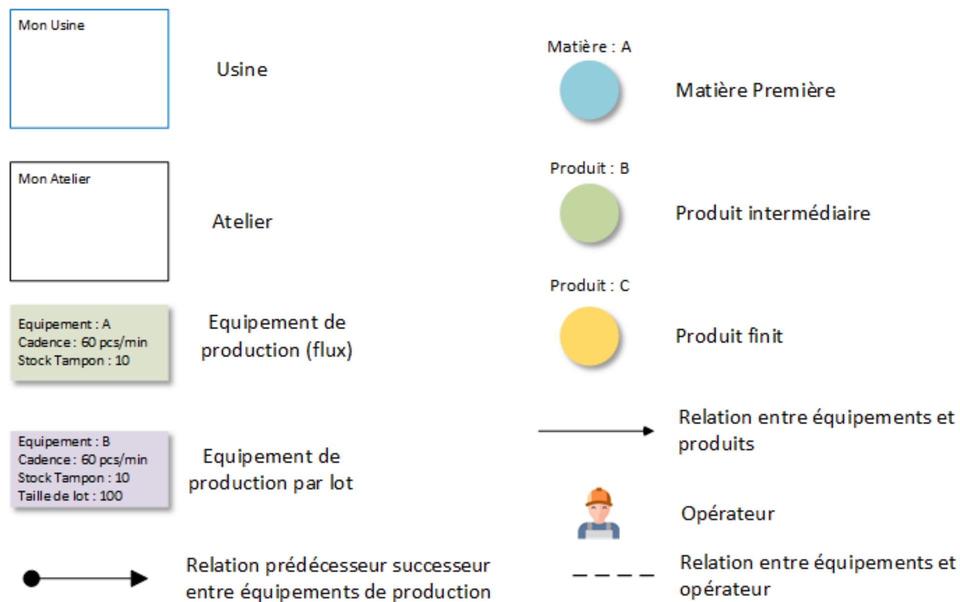


FIGURE 5.5 – Syntaxe concrète pour la modélisation d'une unité de production

La syntaxe concrète ainsi définie permet de modéliser visuellement une unité de production. Il est possible de représenter de manière schématique les interactions entre des équipements de production, des opérateurs, et les flux de produits. La figure 5.6 présente un exemple de modèle d'une unité de production agro-alimentaire destinée à la production de plats cuisinées de type *Pasta-box*. L'atelier *Pasta-Box* est composé de deux sous-ateliers, l'atelier de préparation et l'atelier de conditionnement. Dans l'atelier de préparation un équipement de cuisson et un équipement de préparation des garnitures préparent les recettes qui seront ensuite conditionnées. Ces équipements fonctionnent par lots. Dans ce modèle une pièce correspond à une dose de nourriture, ainsi le *cuiseur* prépare un lot de 1000 doses en 10 minutes. L'atelier de conditionnement est composé de deux équipements fonctionnant en flux, une *doseuse* permet d'assembler une dose de pâte et une dose de garniture dans une barquette. Une *operculeuse* scelle ensuite une opercule sur la barquette.

Cette modélisation d'une unité de production permet de spécifier de manière visuelle les équipements nécessaires à la production d'un produit. Elle permet de visualiser les opérateurs et les matières mises en oeuvre dans la réalisation d'un produit. Dans la phase de conception de l'unité de production ce modèle permet de confronter les points de vue avec les différentes parties prenantes du projets, que ce soit les déci-

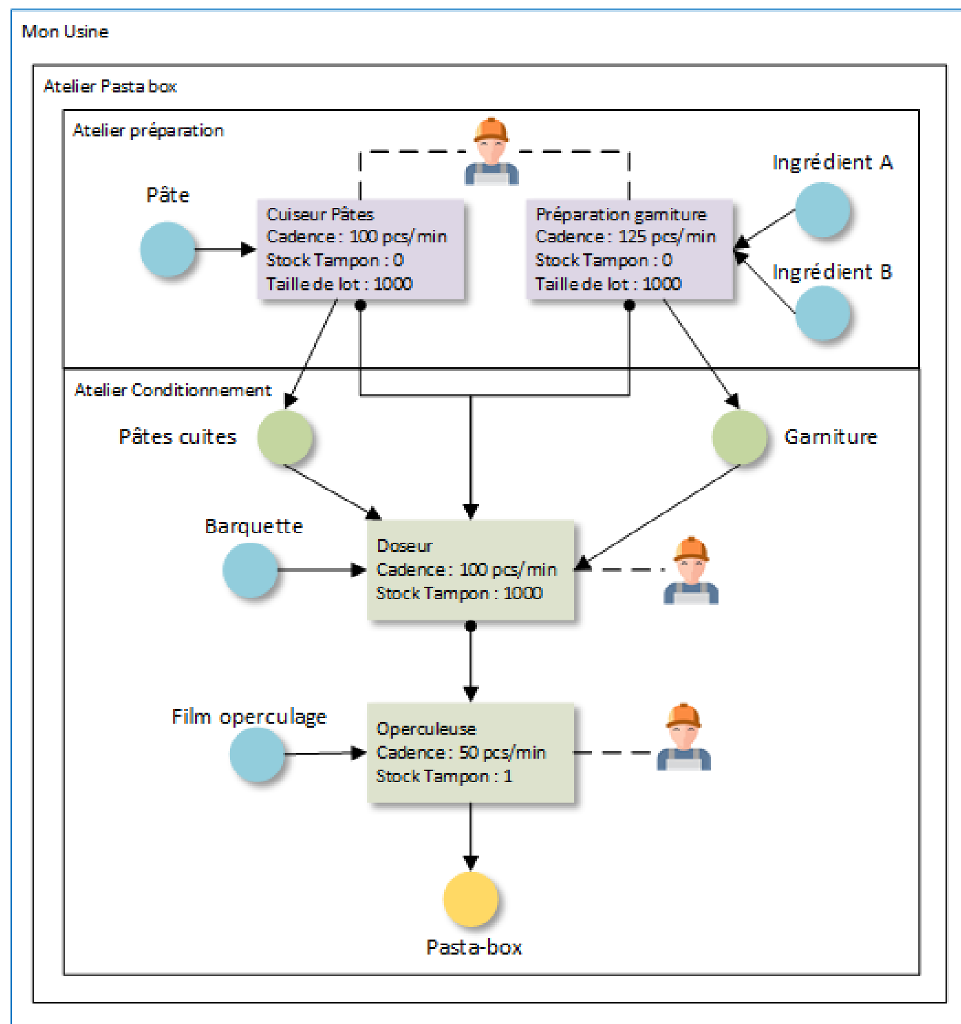


FIGURE 5.6 – Exemple de modèle d'une unité de production

deurs, les sous-traitants ou les futurs opérateurs. Cependant notre modèle ne possède pas de sémantique opérationnelle, notre modélisation ne peut pas répondre à des pré-occupations relatives au comportement de l'unité de production. Dans cet exemple, plusieurs comportements devraient pouvoir être simulés ; on peut vouloir par exemple simuler les interactions entre les opérateurs et les équipements pour vérifier le dimensionnement des équipes de production. On peut vouloir simuler les flux de produits et mesurer les risques de rupture de stock. Chacune de ces simulations implique de doter le modèle d'une sémantique exécutable spécifique. Dans ces conditions étendre le méta-modèle source pour lui adjoindre les sémantiques exécutables implique de créer une extension pour chaque sémantique exécutable ; ce qui abouti a complexifier

le modèle avec des informations uniquement dédiées aux différentes simulations. La méthode de transformation de modèle proposée dans cette thèse, permet de cibler un domaine de simulation spécifique, séparé du domaine source, pour chaque préoccupation de vérification, sans modifier le modèle source. Par la suite, nous aurons pour objectif de simuler les flux de production afin de vérifier la productivité globale de l'unité de production et pour vérifier si il existe des goulots d'étranglement. La sémantique de l'outil de simulation utilisée sera une sémantique de réseaux de pétri [99]. Ces derniers seront utilisés pour simuler les flux traversants les équipements de production (cf meta-modèle cible figure 5.28).

5.1.5 Présentation des étapes de transformation

La présentation de chaque étape de transformation est détaillée au travers de cinq items :

- **Objectif** : Nous décrivons les objectifs de l'étape considérée
- **Formalisation** : au travers de grammaire de graphes triples nous proposons une formalisation des opérations de transformation. Nous formalisons aussi les propriétés attendues des modèles après transformation.
- **Méthode** : nous proposons ici au utilisateurs de la méthode des « guidelines » pour réaliser l'étape considérée.
- **Exemple** : Nous illustrons l'étape de transformation sur l'exemple donnée en fils rouge.
- **Discussion** : Nous proposons un bilan de l'étape de transformation.

5.2 Les étapes de transformations

5.2.1 Sélection

Objectif

Le but de l'étape de sélection est de ne considérer que les éléments nécessaires à l'analyse du modèle. Nous sélectionnons donc le sous-ensemble de classes, attributs et références impliqués dans le sous-domaine à analyser.

Formalisation

Nous considérons la sélection comme une transformation triple directe (définition 21) $G^{source} \overset{\rho,m}{\Longrightarrow} G^{selection}$, avec :

- G^{source} le graphe source de la chaîne de transformation,
- $G^{selection}$ le graphe contenant les éléments sélectionnés dans le graphe source,
- ρ la règle triple de transformation avant (cf definition 22) avec $\rho = (tr : L \overset{tr}{\rightarrow} R, ca)$, avec ca la condition d'application et tr un $\mathcal{M}{-}morphisme\; tr = (tr^S, tr^C, tr^t)$.
- m le morphisme $m : L \rightarrow G^{source}$ appelé « $filtre$ » tel que $m \models ca$ (cf définition 17).

Le graphe sélection obtenu est un sous-ensemble du graphe source $\mathcal{L}(TG^{selection}) \subseteq \mathcal{L}(TG^{source})$ et $G^{selection} \subseteq G^{source}$. Dans ce cas, le graphe de correspondance ne conserve que les concepts concernés par la sélection. Le DMSL source est alors un sous type du DMSL sélectionné, au sens de [28] [29].

Il est possible de spécifier des conditions d'application sur les éléments que l'on veut sélectionner. Nous filtrons les éléments du graphe source suivant des critères relatifs à ses attributs ou à ses relations (figure 5.7) Les contraintes suivantes doivent être respectées pour que le modèle obtenu soit valide :

1. $G^{selection}$ doit posséder un nœud racine r (définition 5) et tous les nœuds sont contenus dans r. Tel que $\exists! r \in V$, $\forall x \in V,\; x \neq r \wedge r\; contien_G\; x$.
2. Les arbres d'héritages doivent être complets et respecter les contraintes de la définition 6 : Si $\exists v_{abs} \in I_V^{selection} \subseteq G_V^{selection} \Rightarrow \exists clan_{I^{selection}}(v)$.
3. $G^{selection} \subseteq G^{source}$, ce qui implique qu'aucune information ne peux être ajouter et que les concepts du graphe source ne sont pas ré-organisés. Le graphe sélection est une copie d'un sous ensemble du graphe source.

Ces contraintes se matérialisent par la satisfaction des conditions d'application sur le graphe R de post-condition. Tel que ; soit le morphisme $n = R \rightarrow G^{selection}$, $n \models ca$.

La figure 5.7 présente un graphe triple de type. Le graphe source TG^{Source} possède deux nœuds abstraits de type A et B et trois nœuds concrets de type C, D et E. Le nœud A est attribué ; l'attribut « attA » est de type booléen. Les nœuds du graphe de correspondance TG^C contiennent chacun une triple règle permettant de définir la transformation entre les nœuds du graphe source et du graphe cible. Les opérations de sélection portent sur les nœuds A, B, D, E, le graphe cible $TG^{Selection}$ contient les mêmes nœuds que le graphe source à l'exception du nœud de type C. La figure

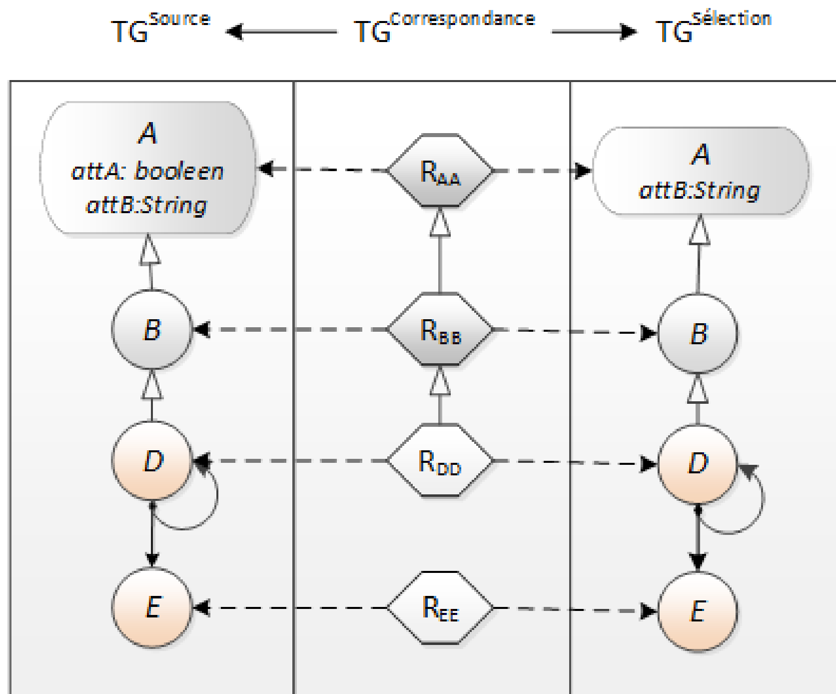


FIGURE 5.7 – Exemple de graphe triple sélection

5.8 présente les quatre règles triples de sélection. Les quatre règles sont des règles « avant » (cf définition 22).

- La première règle « R_{AA} » définit la création de nœuds de type abstrait *A*,
- la seconde règle « R_{BB} » définit la création de nœuds de type abstrait *B*,
- la troisième règle « R_{DD} » définit la création de nœuds de type *C*, pour cette règle il existe une condition d'application ; la valeur de l'attribut booléen *« attA »*, doit être vrai.
- la quatrième règle définit la création de nœuds de type *E*.

Il existe deux post-conditions pour cette dernière règle. Le graphe sélection doit posséder une racine et tous les nœuds doivent être contenus dans celui-ci *(ac2)*. Les arbres d'héritage doivent être complets et il n'y a pas de boucle sur les nœuds d'un arbre d'héritage *(ac3)*.

La figure 5.9 présente un exemple d'instance du graphe triple figue 5.7 auquel nous appliquons les règles triples de la figure 5.8. Les nœuds concrets *d1*, *d2*, *d3* de type *D* héritent de l'attribut *attA* du nœud abstrait *A*. La valeur de cet attribut est *« Vrai »* pour les nœuds d1 et d2 et *« Faux »* pour le nœud *d3*. Le nœud *d3* n'est donc pas sélectionné pour être membre du graphe sélection. Après application des règles

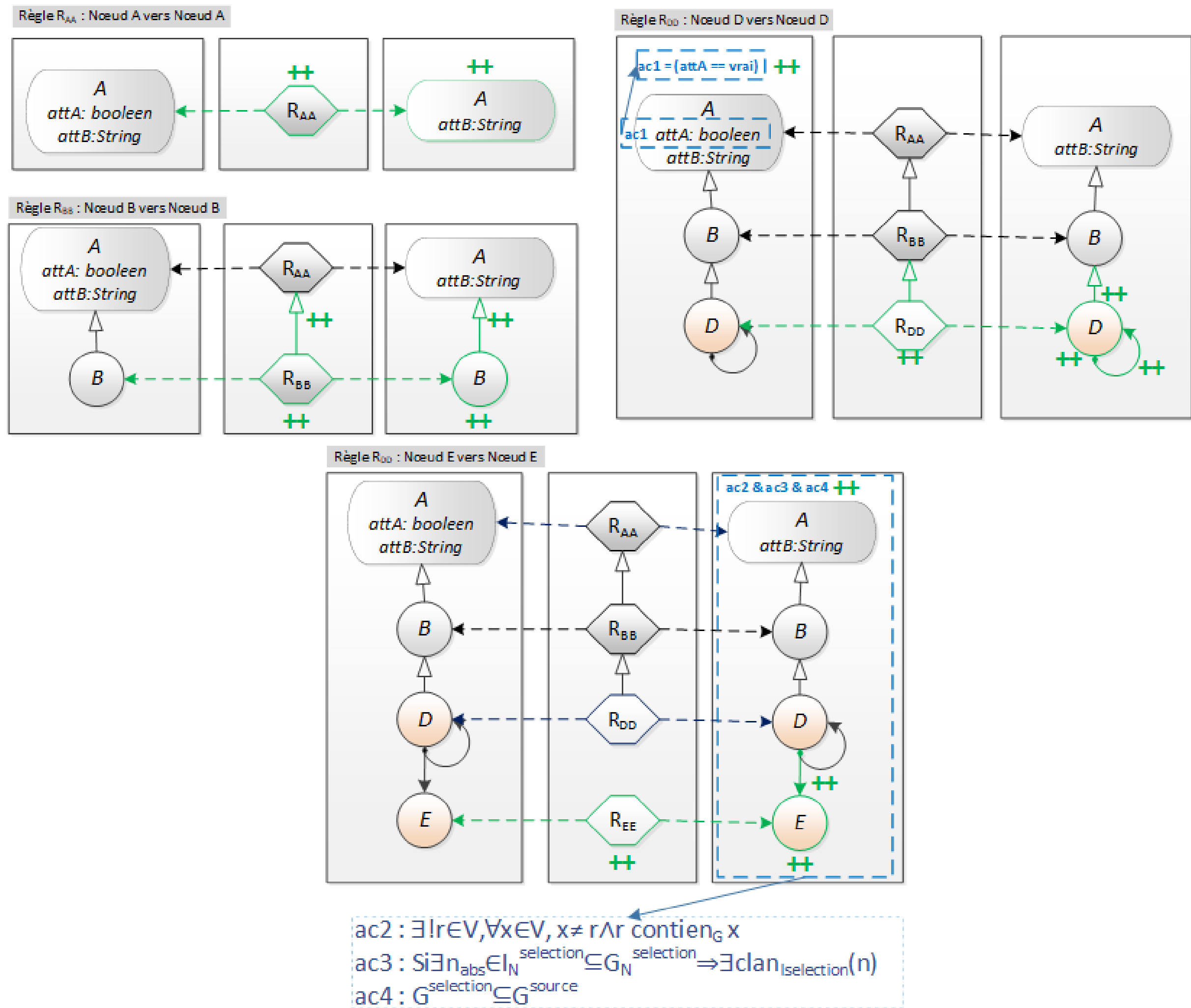


FIGURE 5.8 – Exemple de règles triples pour la sélection

triples le graphe $G^{Selection}$ contient trois nœuds, *d1, d2* et *e1*. *d1* est le nœud racine du graphe et il contient les nœuds *d2* et *e1*. Les post-conditions *ac2, ac3* et *ac4* sont vrais, $G^{Selection}$ est valide.

La figure 5.10 présente une instance pour laquelle la condition d'application *« ac2 »* est invalide. L'attribut *attA* du nœud *d2* porte la valeur *« Faux »* et n'est pas sélectionné pour être membre de $G^{Selection}$. Le graphe $G^{Selection}$ est alors composé de trois nœuds *d1,d3* et *e1*. Chacun de ces nœuds peut être candidat pour être nœud racine, mais aucun ne contient les autres. La post-condition ac2 n'est pas respectée le graphe $G^{Selection}$ est invalide. Dans ce cas la transformation s'arrête, il faut alors reconsidérer les critères de sélection pour aboutir à un modèle de sélection valide et considérer

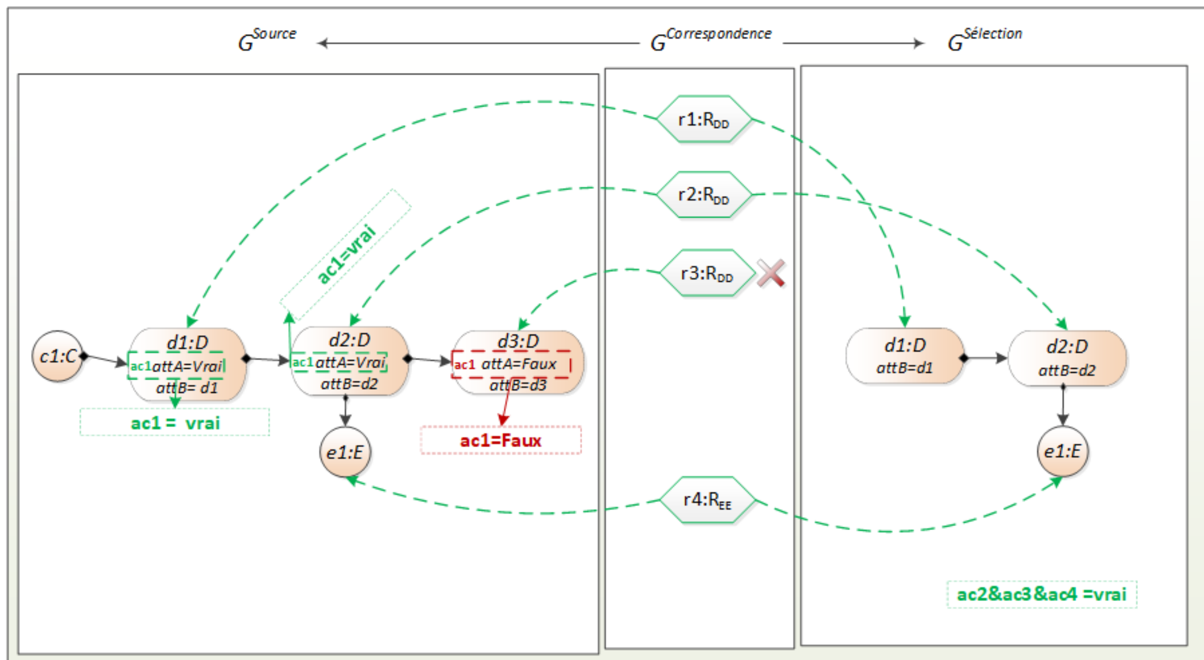


FIGURE 5.9 – Exemple d'instance du graphe triple de sélection pour laquelle les post-conditions sont vraies

une réorganisation du modèle dans l'étape de sélection. par exemple pour suprimer le nœud *d2* et relier les nœuds *d1* et *d3* .

Méthode Pour mettre en œuvre cette étape, nous proposons dans un premier temps de sélectionner l'ensemble des classes ainsi que les attributs pertinents pour la simulation. Pour garantir l'exactitude du méta-modèle obtenue, nous conservons aussi toutes les références entre les classes sélectionnées. Si des classes apparaissent dans une chaîne de référence entre des classes sélectionnées, nous les ajoutons. Nous sélectionnons également des classes abstraites contenant des attributs et des références utiles pour l'analyse. A partir de ces classes abstraites, nous sélectionnons tous les chemins d'héritage, y compris les classes intermédiaires, assurant une relation d'héritage entre les classes abstraites et les classes concrètes. Enfin, nous sélectionnons une classe jouant un rôle de classe racine. A partir de cette classe racine, nous sélectionnons toutes les classes et références assurant un chemin de contenance entre la classe racine et toutes les classes concrètes.

Après application de l'opération de sélection, les objets du domaine de sélection possèdent un antécédent et un seul dans le domaine source. Brian Henderson-Seller

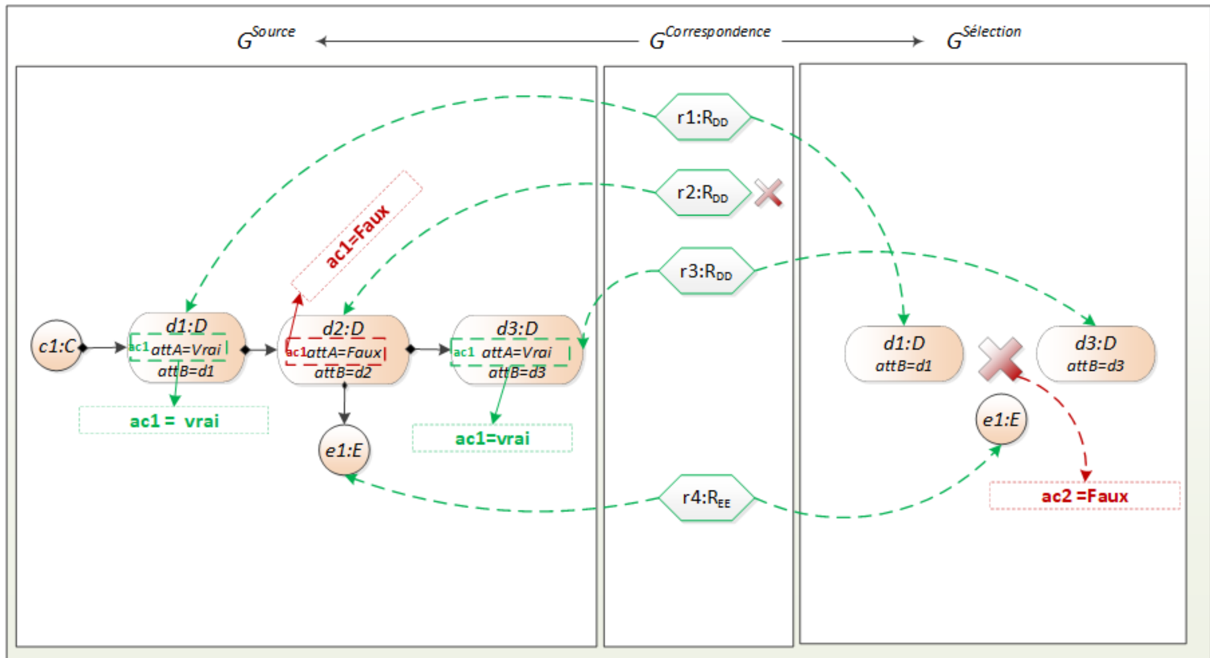


FIGURE 5.10 – Exemple d'instance du graphe triple de sélection pour laquelle la post-condition $ac2$ est fausse

montre dans [5] page 16, qu'il s'agit d'une opération bijective. (figure 5.11)

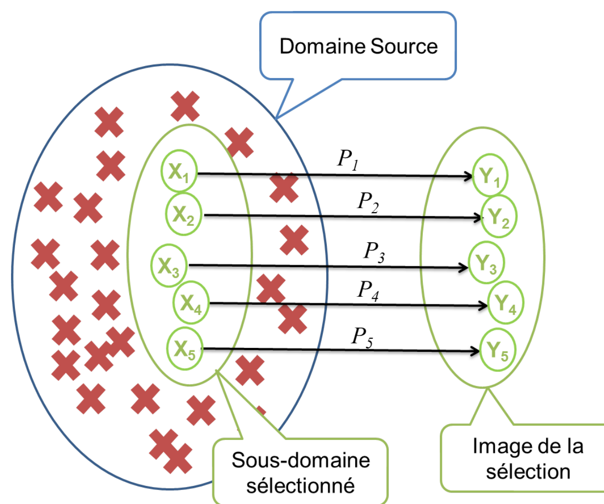


FIGURE 5.11 – Opération bijective

Exemple Dans l'application fil rouge, pour analyser les flux de production, nous avons besoin de sélectionner les équipements de production. Ces équipements sont attachés

à des ateliers nous devons donc sélectionner le concept « atelier ». Il est possible de préciser la sélection en ajoutant un critère de sélection sur le nom de l'atelier, dans notre application nous souhaitons sélectionner l'atelier (et ses sous ateliers) pour *Atelier.nom == "Atelier Pasta box"*. Le concept racine « Usine » n'est pas un membre de notre sélection, le nouveau concept « Atelier » devient le nouveau concept racine. Le méta-modèle sélection obtenu est donné par la figure 5.12. Après application de

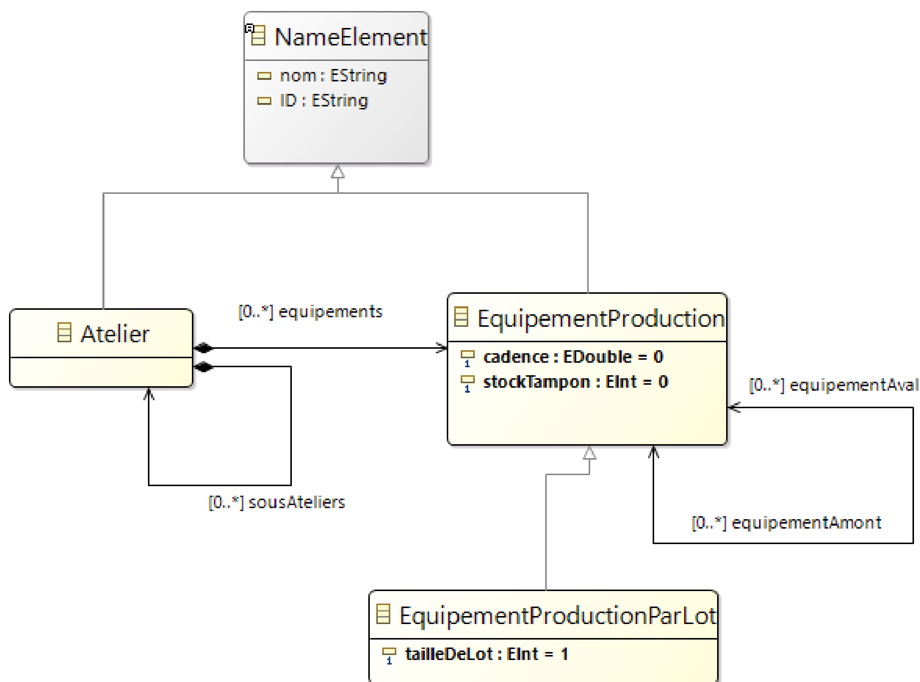


FIGURE 5.12 – méta-modèle Sélection

l'étape de sélection nous obtenons le diagramme d'objets figure 5.13, seul les objets de type *Atelier*, *EquipementDeProduction*, *EquipementDeProductionParLots* sont conservés avec leurs relations. La figure 5.14 montre une représentation des objets sélectionnés dans la syntaxe concrète de la figure 5.5

Discussion

Cette étape de sélection est généralement implicite dans la plupart des approches proposées dans la littérature. Dans notre approche, il nous apparaît fondamental d'expliciter le sous-ensemble impliqué dans l'analyse des propriétés du modèle. En effet, les propriétés seront vérifiées, sur un sous-ensemble du modèle et non sur le modèle

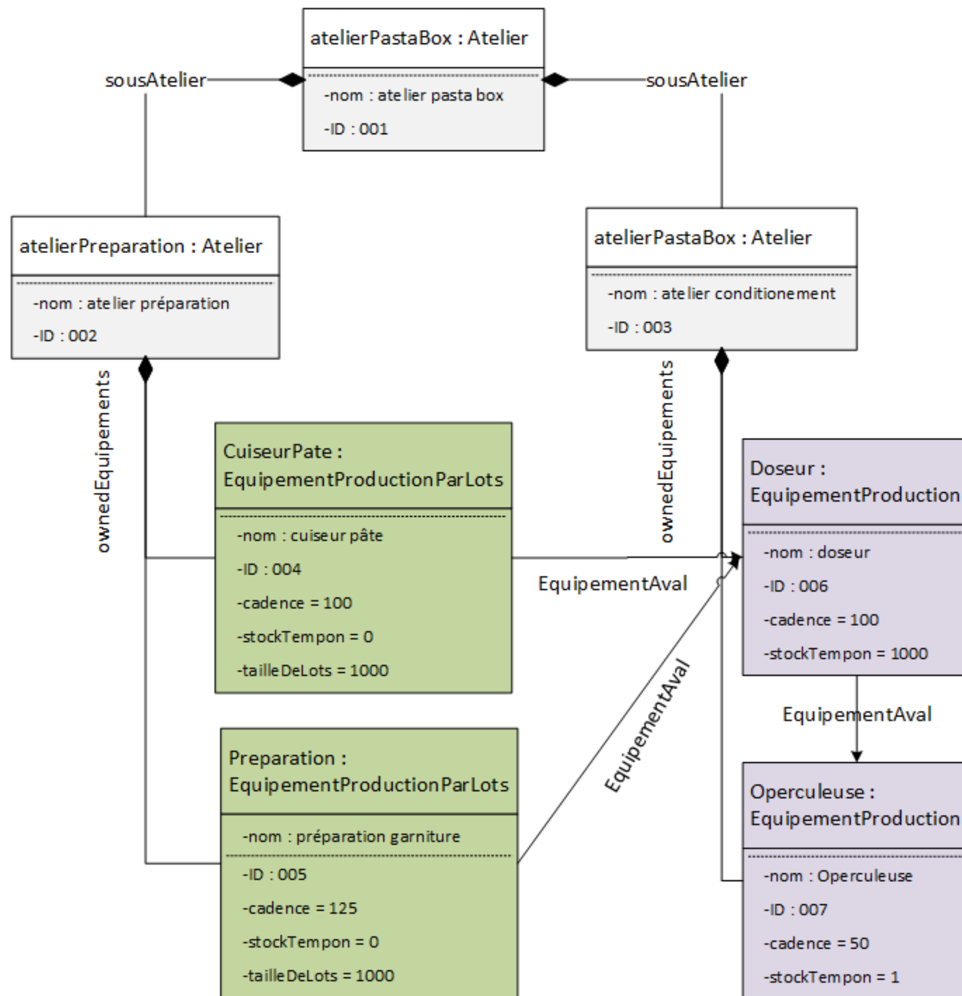


FIGURE 5.13 – Diagramme objet de la sélection

complet, ce qui n'est pas neutre d'un point de vue sémantique. Pour que les conclusions effectuées sur la base du modèle de simulation soient pertinentes il est impératif que les concepts non sélectionnés n'aient aucun impact sur le comportement des concepts sélectionnés au regard des objectifs de vérification. Nous faisons donc ici l'hypothèse forte que le sous-ensemble des concepts non sélectionnés n'a aucun impact sur le comportement de la simulation cible au regard des objectifs de vérifications choisis.

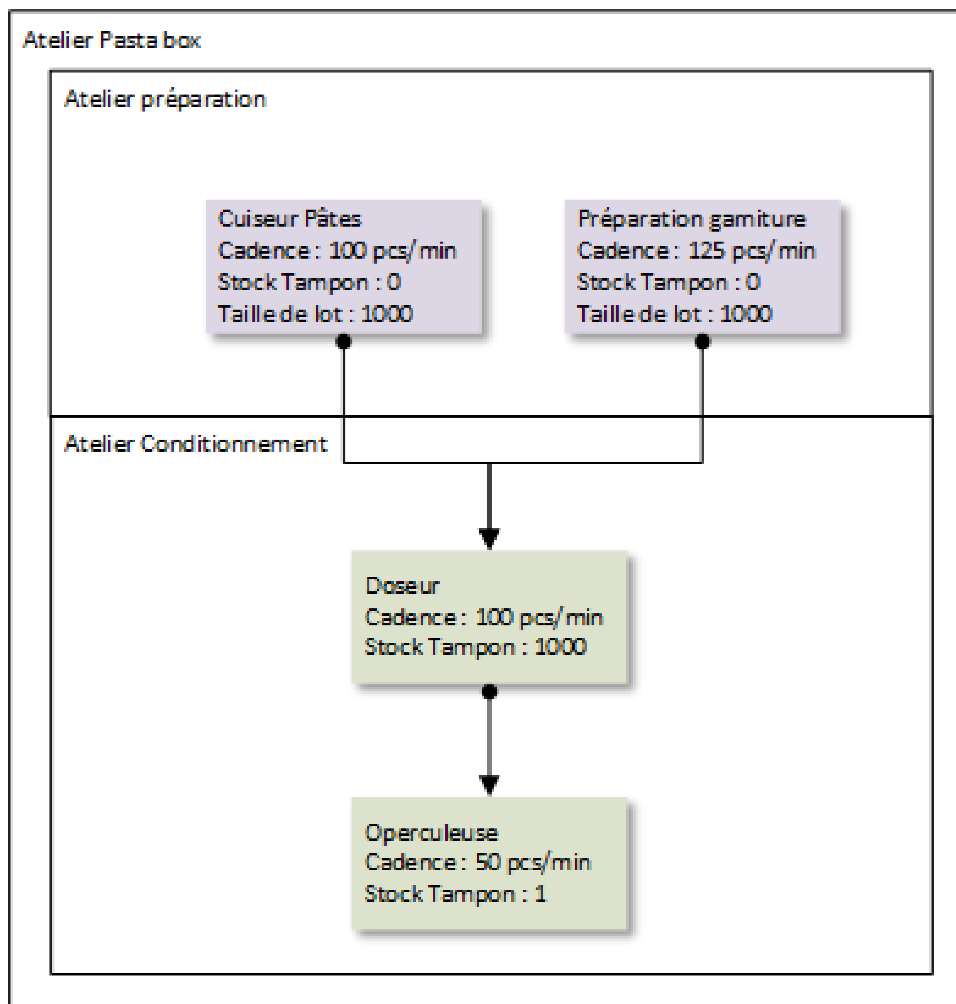


FIGURE 5.14 – représentation de la sélection suivant la syntaxe concrète de 5.5

5.2.2 Réorganisation

Objectifs

L'étape de réorganisation est une étape de préparation à l'étape d'alignement. Nous organisons les concepts de notre sélection précédente pour que les instances du sous-domaine sélectionné puissent être directement alignées sur le modèle cible. Dans cette étape, il n'y a pas de création d'information. Les nouvelles informations qui apparaissent le sont uniquement par calcul ou par réarrangement des informations disponibles dans le modèle source.

Formalisation

Dans cette thèse, nous considérons l'étape de réorganisation sous la forme de trois opérations de base :

- **La fusion de concepts :** Deux ou plusieurs concepts du modèle source sont réorganisés pour ne former qu'un seul concept.
- **La fission de concepts :** Les informations d'un concept du modèle source sont réparties entre plusieurs concepts du modèle cible.
- **La suppression de concept :** Un concept du modèle source est supprimé dans le modèle cible. Les liens qui unissent ce concept aux autres sont alors réorganisés.

Fusion de concept Deux ou plusieurs concepts du modèle source sont réorganisés pour ne former qu'un seul concept. La fusion peut s'appliquer à des objets de classes différentes ou de mêmes classes. Une fusion de nœuds est une transformation triple directe $G \overset{\rho,m}{\Longrightarrow} H$ qui transforme un ensemble de nœuds d'un graphe source $\{V_G\} \in G$ en un nœud unique $v_H \in H$ d'un graphe cible, par une règle triple ρ et d'un morphisme filtre m. Avec ρ constituée d'une production triple $p : G^n \rightarrow H, \forall n \in \mathbb{N}$ et d'une condition d'application ca tel que $m \models ca$. Nous présentons en exemple deux cas de fusion de concepts. Dans le premier cas, nous voulons supprimer dans l'opération de réorganisation, une relation de contenance d'un nœud sur lui même. Les instances de ce nœud liés par un arc de contenance sont fusionnées en un seul nœud. Dans le second cas la réorganisation consiste à fusionner un ensemble de nœuds en fonction d'une condition d'application.

Cas 1 : Suppression d'une relation de contenance d'un nœud sur lui même. Les instances de ce nœud liés par un arc de contenance sont fusionnées en un seul nœud. Si les nœuds sources possèdent des attributs et des relations, ces derniers sont fusionnés suivant la règle établie par le graphe de correspondance. Dans l'exemple, (figure 5.15). Le graphe source $TG^{Selection}$ est composé de deux nœuds de type *B* et *C*. *B* possède un arc de contenance sur lui même ainsi qu'un arc de relation sur *C*. Dans le graphe $TG^{ReOrganise}$ l'arc de contenance de *B* sur lui même n'existe pas. Le graphe d'instance $G^{Selection}$ est composé de quatre nœuds *b1, b2, b3* instances de *B* et *c1* instance de *C*. Les nœuds *b1, b2, b3* sont liés par des arcs de contenance tel

que $b1\, contien_G\, b2 \wedge b2\, contien_G\, b3$. *b2* et *c1* sont liés quant à eux par un arc de relation. Le graphe cible de la transformation, $G^{ReOrganise}$ contient deux nœuds ; un nœud issu de la fusion *"b1b2b3"* et un nœud *c1* lié par un arc de relation à *b1b2b3*.

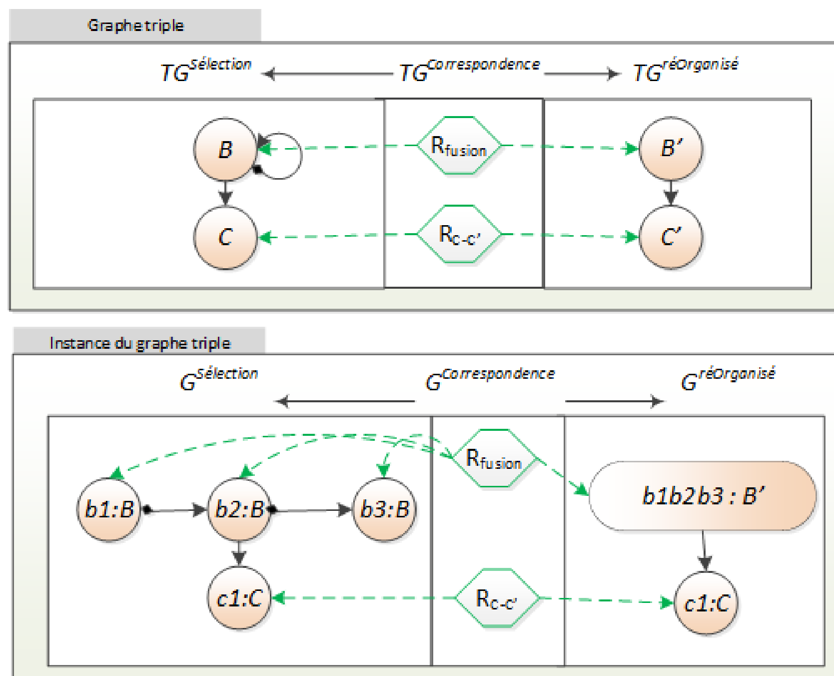


FIGURE 5.15 – Fusion cas 1

Cas 2 : Fusion d'un ensemble de nœuds avec une condition d'application. Les nœuds sont fusionnés suivant une règle et une condition d'application. La condition d'application permet de « matcher » les nœuds devant être fusionnés ensemble ($m \models ca$). Dans ce cas il est important de considérer le cas ou la condition d'application n'est pas satisfaite ($m \models \neg ca$) :

- Nous pouvons ignorer ces concepts. Dans ce cas les concepts ne sont pas présent dans $G^{ReOrganise}$ et les informations contenues par ces concepts ne sont pas conservées en vue de la simulation.
- Nous pouvons ajouter une règle pour $m \models \neg ca$ et organiser différemment ces concepts conformément au méta-modèle cible de simulation.

Dans l'exemple figure 5.16 les nœuds de type *C* et *D* sont fusionnés soit en un nœuds de type *CD'* soit transformés respectivement en un nœud *C'* et *D'* en fonction de la satisfaction (ou non) d'une condition d'application. La figure 5.17 montre un

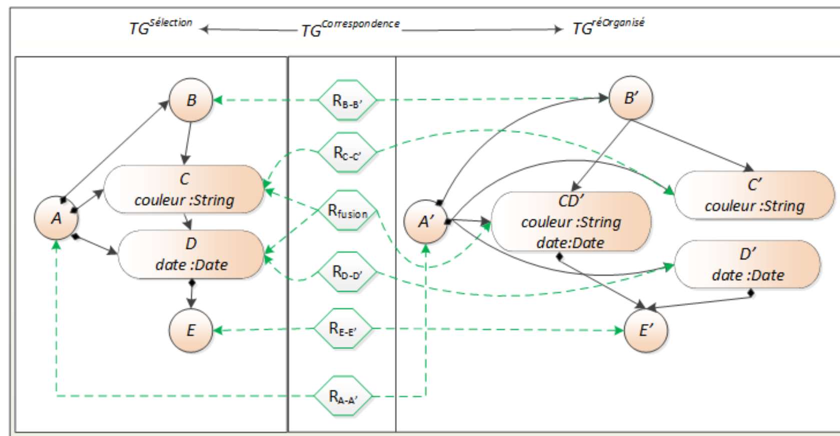


FIGURE 5.16 – Grammaire de Graphe Triple cas2

exemple de règles triple régissant une fusion de nœuds. Les nœuds sont fusionnés en fonction d'une condition d'application. Cette condition d'application peut s'appliquer soit sur les attributs ou soit sur les relations des nœuds considérés. Dans tous les cas, il faut prévoir une règle qui s'applique si la condition d'application est fausse tel que $m \models \neg ca$.

Fission de concepts. L'opération de fission est une transformation inverse de la transformation de fusion. La fission de nœuds est une transformation triple directe $G \overset{\rho,m}{\Longrightarrow} H$ qui transforme un nœud unique d'un graphe source $v_g \in G$ en un ensemble de nœuds $\{V_H\} \in H$ d'un graphe cible, par une règle triple ρ et d'un morphisme filtre m. Avec ρ constituée d'une production triple $p : G \to H^n, \forall n \in \mathbb{N}$ et d'une condition d'application ca tel que $m \models ca$. La répartition des attributs et des relations s'ils existent, entre les nouveaux nœuds du domaine cible, est réalisée suivant les règles établies par le graphe de correspondance. La figure 5.19 montre un exemple de graphe triple mettant en œuvre une fission de nœuds. Dans cet exemple, le nœud de type *A* possède deux attributs *a* et *b* et deux relations vers les nœuds de type *B* et *C*. Les informations contenus dans le nœud *A* sont réorganisées pour appartenir à deux nœuds séparés *A'* et *A"* dans le graphe réorganisé.

La figure 5.20 donne la régle triple pour notre exemple de fission de concept.

Suppression de concepts Certains concepts bien qu'ils ne soient pas utiles à la simulation du modèle ne peuvent être supprimés pendant l'étape de sélection. Soit

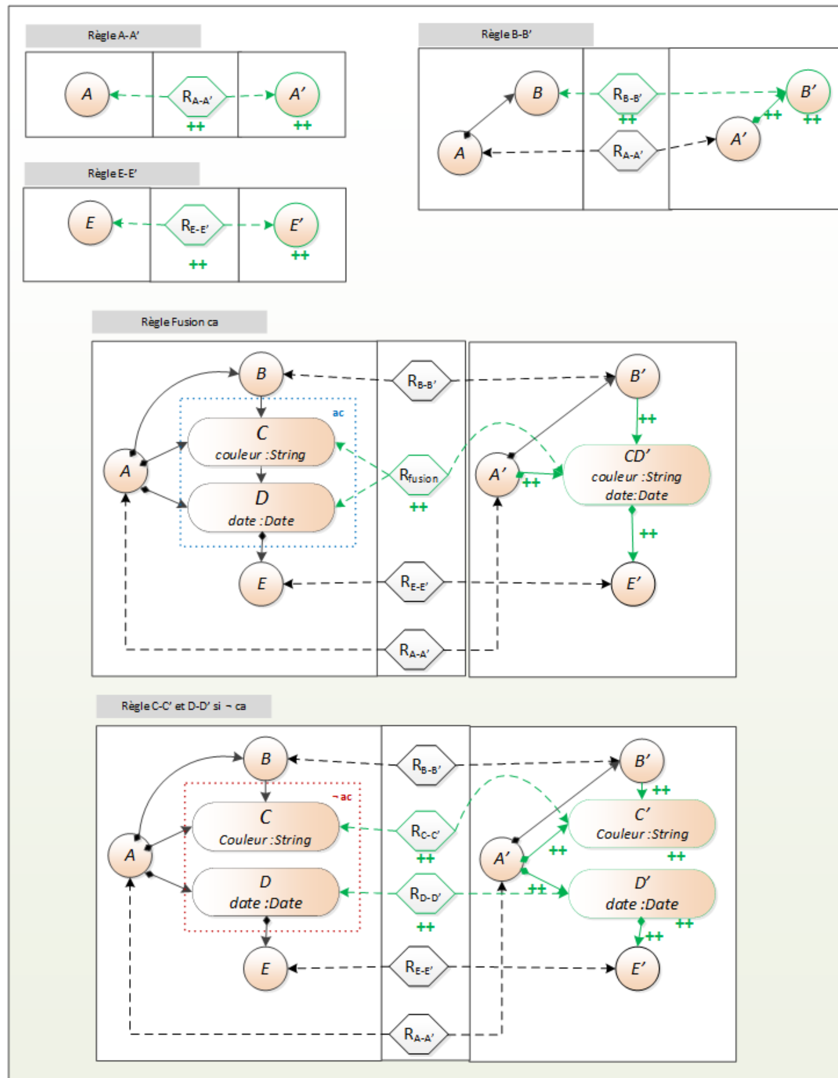


FIGURE 5.17 – Règle Triple cas2

parce qu'ils sont inclus dans le chemin de contenance, soit parce qu'ils sont inclus dans un arbre d'héritage. Le graphe réorganisé doit conserver les relations transitives de contenance existantes entre les nœuds conservés dans le graphe réorganisé. Soit $(a, b, c) \in (TG_V^{selection})^3$ trois nœuds liés par une relation de contenance $a\ contient_G\ b \wedge b\ contient_G\ c$ et $(a', c') \in TG_V^{reOrganise} \times TG_V^{reOrganise}$ tel que les morphismes $f_{reorg}(a) = a'$ et $f_{reorg}(c) = c'$, alors $f_{reorg}(b) = \varnothing \Rightarrow a'\ contient_G\ c'$.
Le même principe s'applique sur les héritages.

Les figures 5.21 et 5.22 présentent le graphe triple et les règles triples d'une suppression de nœuds. Le nœud de type *B* est membre de la relation transitive : $A contient_G B \wedge$

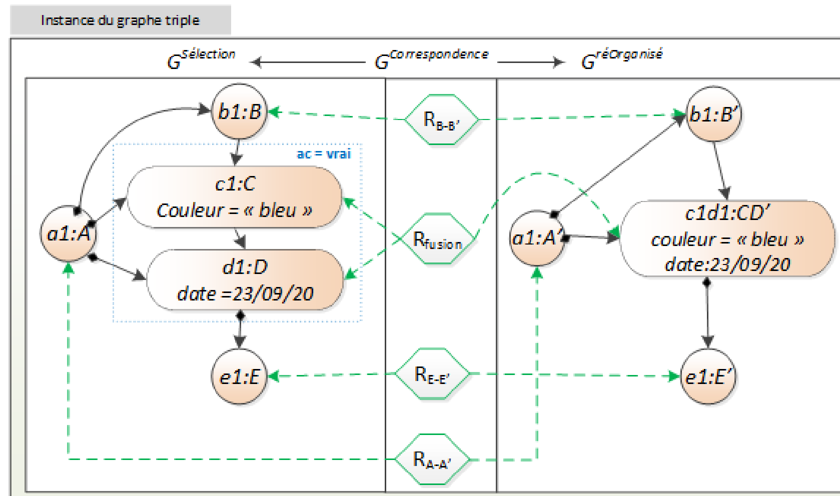


FIGURE 5.18 – instance du graphe Triple cas2

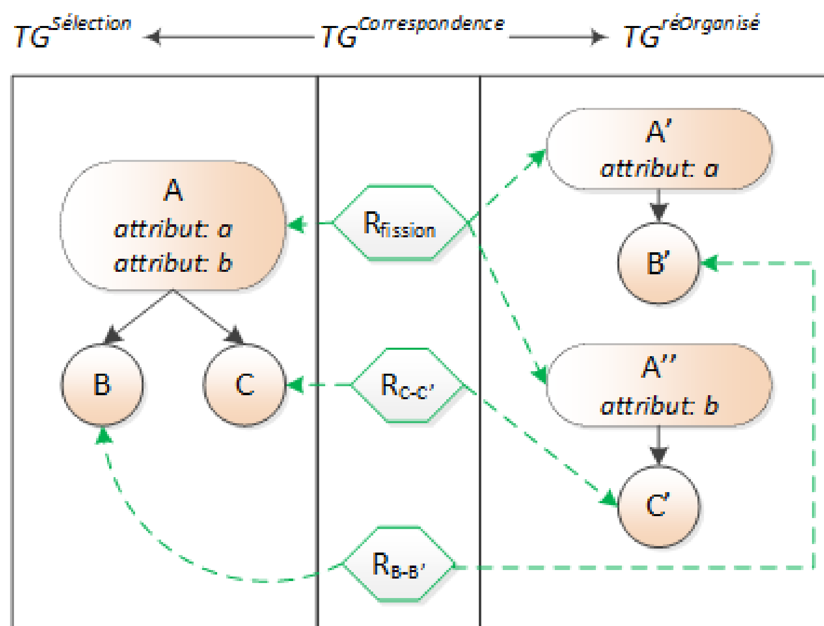


FIGURE 5.19 – Triple graphe fission de nœuds

$B\ contient_g\ C$. Lors de la transformation le nœud *B* est supprimé et remplacé par un arc de contenance entre *A* et *C* afin de conserver la relation transitive $A\ contient_G\ C$.

Fusion, Fission, Suppression Les trois opérations de bases peuvent être combinées sur un même nœud. Ainsi un nœud appartenant à un chemin de contenance peut être supprimé et ses attributs et relations fusionnés avec d'autres nœuds, tel que

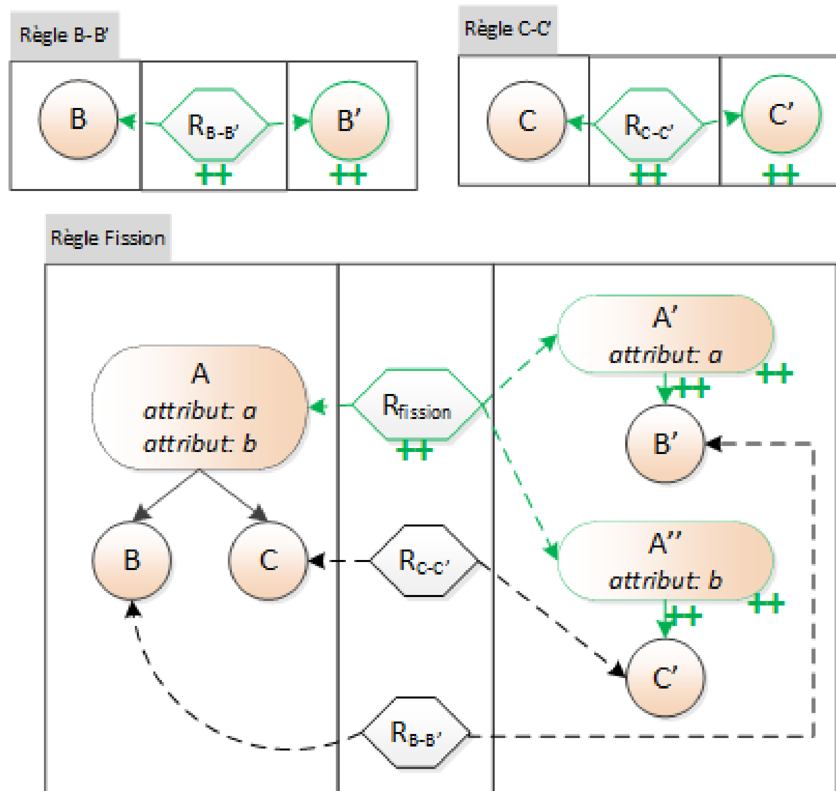


FIGURE 5.20 – Règle triple fission de nœuds

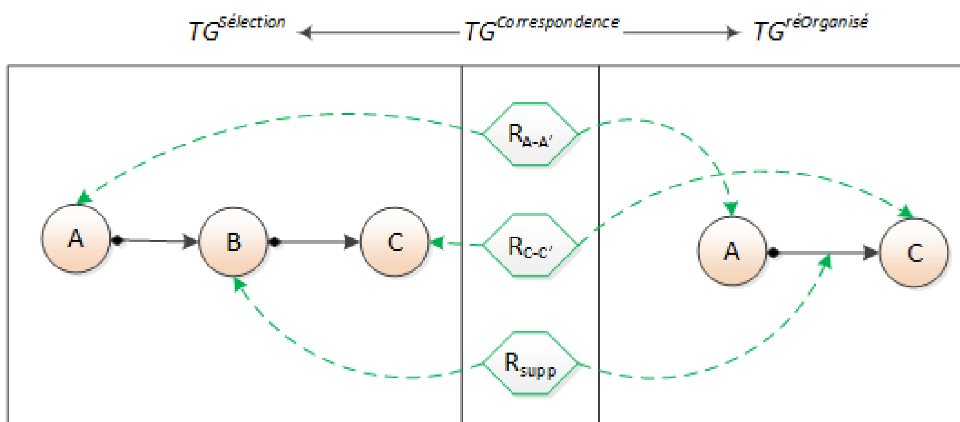


FIGURE 5.21 – Graphe triple de suppression de nœud

dans l'exemple figure 5.23

Si les nœuds candidats à la suppression possèdent des attributs ou des relations, ils doivent être soit supprimés, soit fusionnés avec d'autre nœuds.

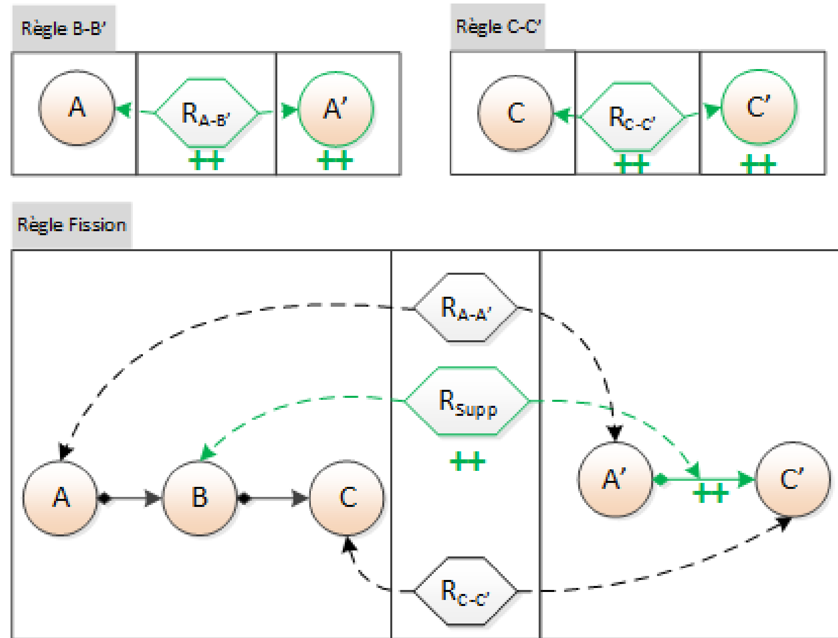


FIGURE 5.22 – Règle triple de suppression de nœud

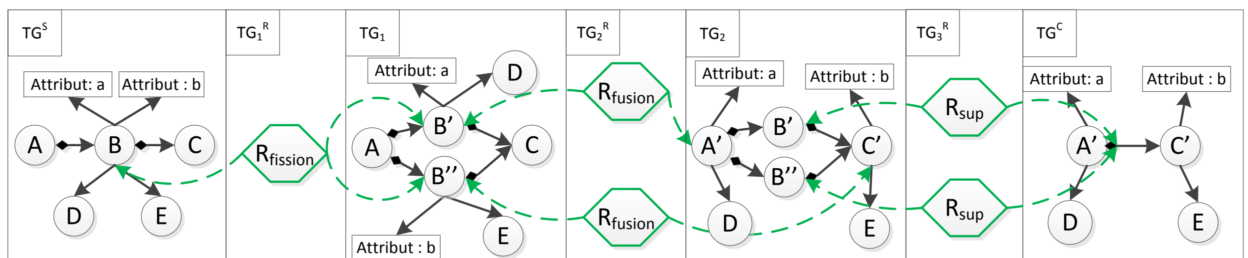


FIGURE 5.23 – Fission, fusion, suppression

Contraintes liées à la réorganisation

1. La transformation de modèles doit satisfaire aux contraintes définies pour les graphes et morphismes d'IC-Graphes (définition 7)

2. La transformation de modèles doit être complète, elle doit s'appliquer à tous les éléments du modèle sélectionné. $\bigcup_{i=1}^{i=n} ca_i(G^{selection}) = G^{selection}$

3. La syntaxe et la sémantique du modèle obtenu doivent être correctes : $\mathcal{M}(G^{reOreganise}) \models \mathcal{L}(TG^{reOreganise})$. Par exemple, si le modèle réorganisé possède des contraintes sur l'espace de nommage (absence de doublon dans les noms des objets), les

règles de transformation doivent en tenir compte pour qu'elles soient respectées.

Méthode

Nous réalisons la réorganisation des concepts avec l'objectif de pouvoir ensuite les aligner directement avec le modèle cible. Pour cette étape, il faut considérer le méta-modèle cible. Nous établissons d'abord un lien entre chaque concept du Méta-modèle source (MMS) et un concept du méta-modèle cible (MMC) :

- Si un concept du MMS n'a pas de correspondance dans le MMC il est supprimé.
- Si deux (ou plusieurs) concepts de MMS sont reliés au même concept du MMC, ils sont fusionnés.
- Si les informations d'un concepts du MMS sont réparties dans plusieurs concepts du MMC, nous utilisons une opération de fission.

Pour réaliser les opérations de réorganisation, nous pouvons utiliser des bibliothèques d'opérateurs de refactorisation (« flatten, hide, move ... ») fournies par des outils et langages de co-évolution telles que *ModifRoundtrip* [100] ou *Epsilon Flock* [101]. Nous pouvons, par exemple aplatir tous les attributs des classes abstraites avant de supprimer ces dernières (combinaison des opérateurs « flatten » et « hide » définis par l'outil de co-évolution *ModifRoundtrip*).

Exemple

Le but de la réorganisation est de préparer l'alignement des concepts sélectionnés avec les concepts du domaine cible (cf figure 5.28). Pour préparer l'alignement nous réorganisons les concepts sources tels que présentés en figure 5.24. La notion de sous-atelier est supprimée. Les objets et leurs relations ayant une relation d'agrégation avec l'objet racine de type *« Atelier »* sont fusionnés avec ce dernier. Le concept de *« EquipementDeProductionParLot »* est fusionné avec le concept de *« EquipementDeProduction »*. Enfin chaque connexion entre équipement de production est spécifié avec un concept *« ConnexionEquipementAmont »* et stocke les informations de l'attribue *« stockTampon »* (opération de fission).

Nous obtenons après l'étape de réorganisation le méta-modèle figure 5.25.

La figure 5.26 présente le diagramme objets après l'étape réorganisation. Il n'y a plus de distinction entre les objets représentant les équipements par lots et les équipe-

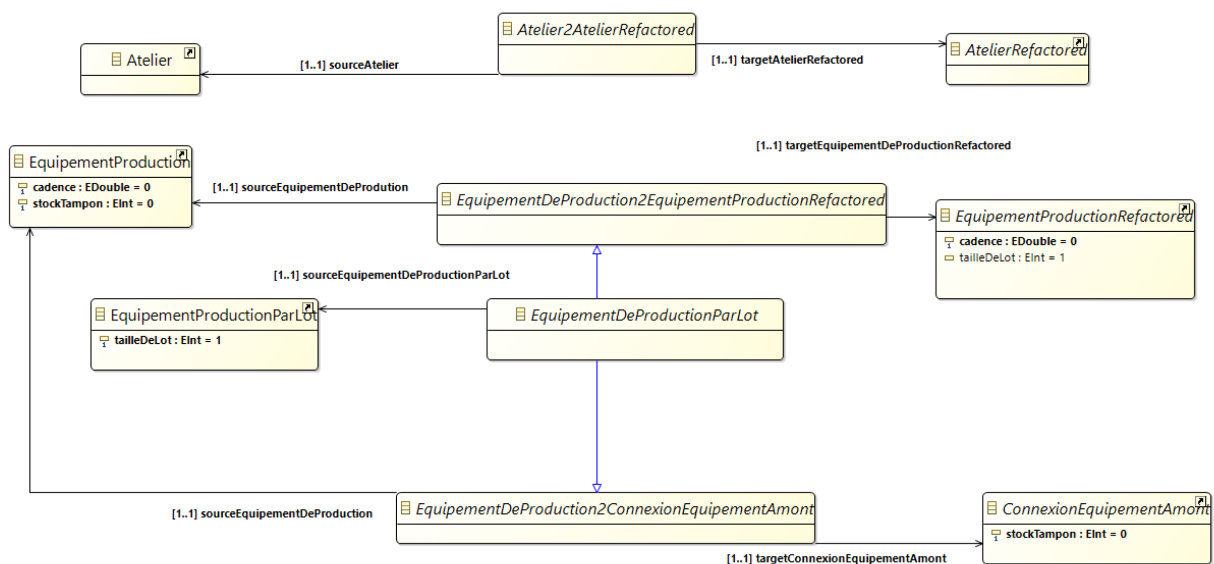


FIGURE 5.24 – Alignement des concepts sélectionnés avec les concepts réorganisés

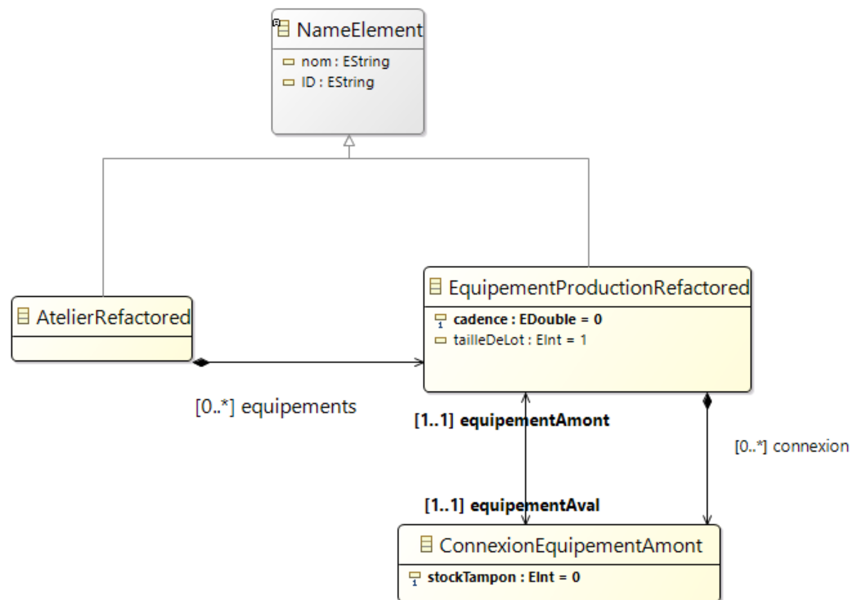


FIGURE 5.25 – Méta-modèle réorganisé

ment de production en flux. L'attribut « *tailleDeLot* » est donc fusionné avec « *EquipementDeProduction* ». Cependant cet attribut n'a de sens que pour les équipements de production par lots, pour les objets issues du concepts « *EquipementDeProduction* » cet attribut ne peut pas être initialisé. Il n'est en effet pas possible d'ajouter d'information lors de l'étape de réorganisation.

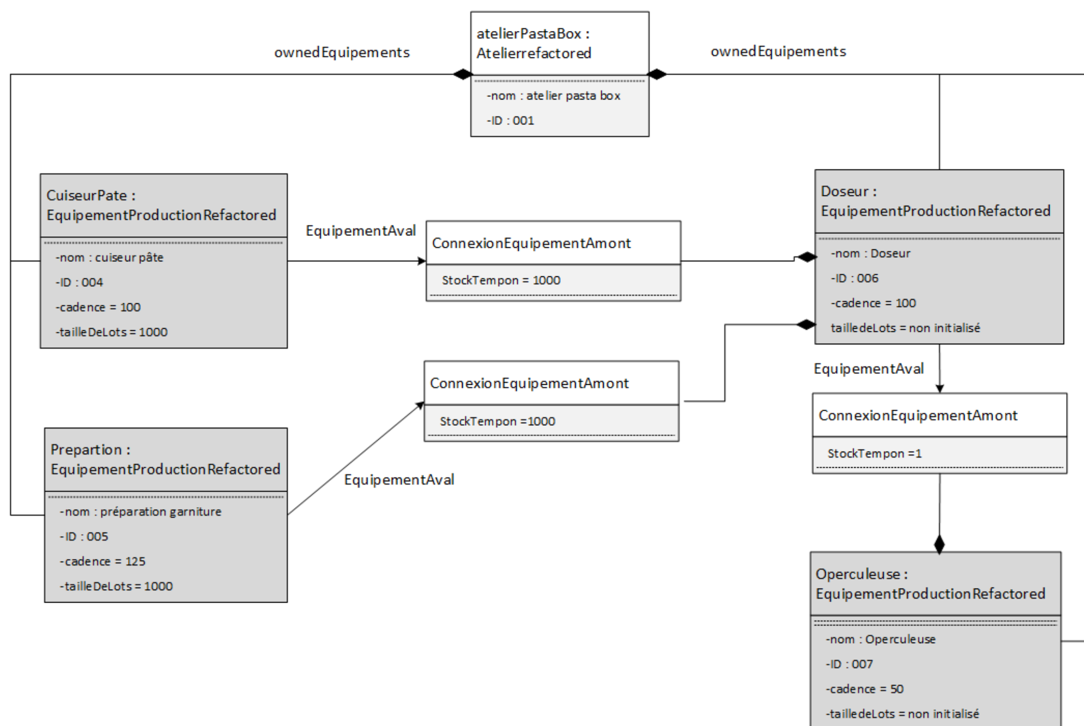


FIGURE 5.26 – Diagramme objet réorganisé

Discussion

Les étapes de sélection et de réorganisation préparent l'étape d'alignement. Le sous-typage induit par la sélection reste un idéal. Dans la réalité, il est généralement nécessaire de ré-organiser les concepts pour pouvoir ensuite les aligner directement sur le domaine cible. Lors de l'étape de réorganisation seul les informations présentes dans le domaine sources sont considérées. Il n'est pas possible d'ajouter d'information lors de cette étape.

5.2.3 Alignement

Objectifs

L'étape d'alignement traduit les concepts du domaine réorganisé dans le langage défini par le domaine cible. Le domaine cible possède la sémantique opérationnelle permettant la simulation du modèle.

Formalisation

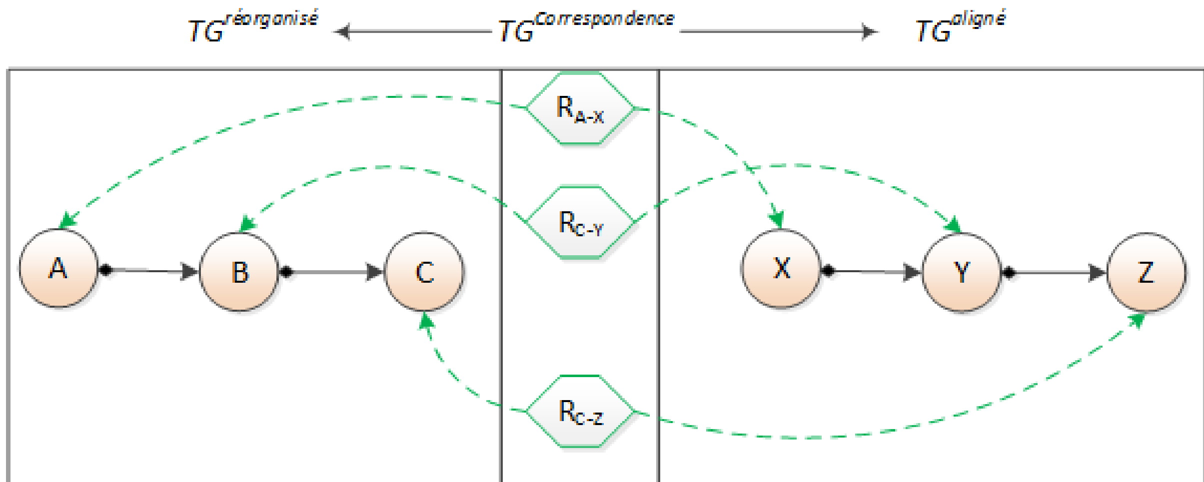


FIGURE 5.27 – Exemple de graphe triple alignement

La figure 5.27 montre un exemple d'alignement. Les concepts du domaine source *A, B, C* sont directement alignés sur les concepts du domaine cible *X, Y, Z* par une transformation directe. La structure des graphes sources et cibles sont identiques, seul les noms de concepts divergent. Suite à l'étape d'alignement, les concepts du domaine source sont exprimés selon les concepts du domaine cible. Pendant cette étape il n'y a pas d'ajout ni de suppression d'information, les contraintes liées au domaine cible ayant été traitées lors de l'étape de réorganisation.

Méthode

L'alignement est une transformation de modèle à modèle avec une correspondance directe entre classes du domaine source et du domaine cible. Une grande variété de langages de transformation peuvent être utilisé ici, tel que *QVT, ATL, ETL, Xtend*. Ces langages peuvent être utilisés au travers d'outils de coévolution tel que *ModifRountrip* en utilisant l'opérateur « rename ».

Exemple

Le méta-modèle cible (figure 5.28) permet une simulation basée sur les réseaux de Petri. Nous trouvons dans ce méta-modèle les concepts de « *Place* » et de « *Transi-*

tion » . Le concept de *« Jeton »* est attaché au concept de *« Place »*. Pendant l'exécution les jetons évoluent de place en place par réécriture de graphe tel que présenté dans l'exemple de double pushout figure 2.14 du chapitre 2.2.

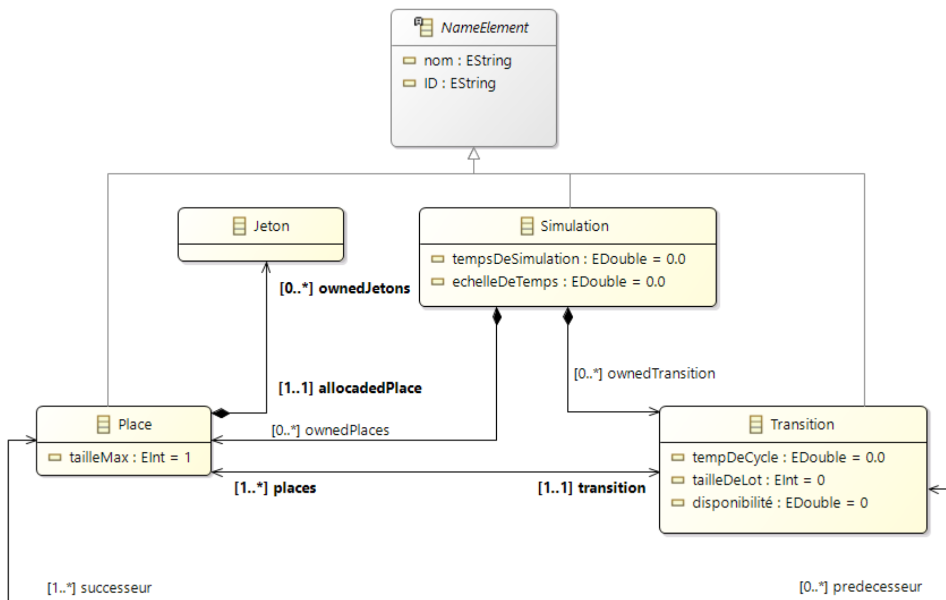


FIGURE 5.28 – Méta-modèle cible

La figure 5.29 présente la relation entre le méta-modèle de traçabilité (au centre), le méta-modèle réorganisé (à gauche) et le méta-modèle cible (à droite). Ainsi, nous obtenons une représentation de l'alignement des concepts du méta-modèle réorganisé vers le méta-modèle de simulation :

- Le concept « AtelierRefactored » est aligné avec le concept « Simulation ».
- Le concept « EquipementProductionRefactored » est aligné avec le concept « Transition ».
- Le concept « ConnexionEquipementAmont » est aligné avec le concept « Place ».

Après l'étape d'alignement nous obtenons le diagramme objets représenté en figure 5.30. Afin de simplifier la représentation l'objet de type « Simulation » n'est pas représenté sur ce diagramme.

Discussion

L'étape d'alignement est l'étape la plus commentée dans la littérature. Elle est considérée comme l'étape la plus importante du point de vue sémantique, dans le

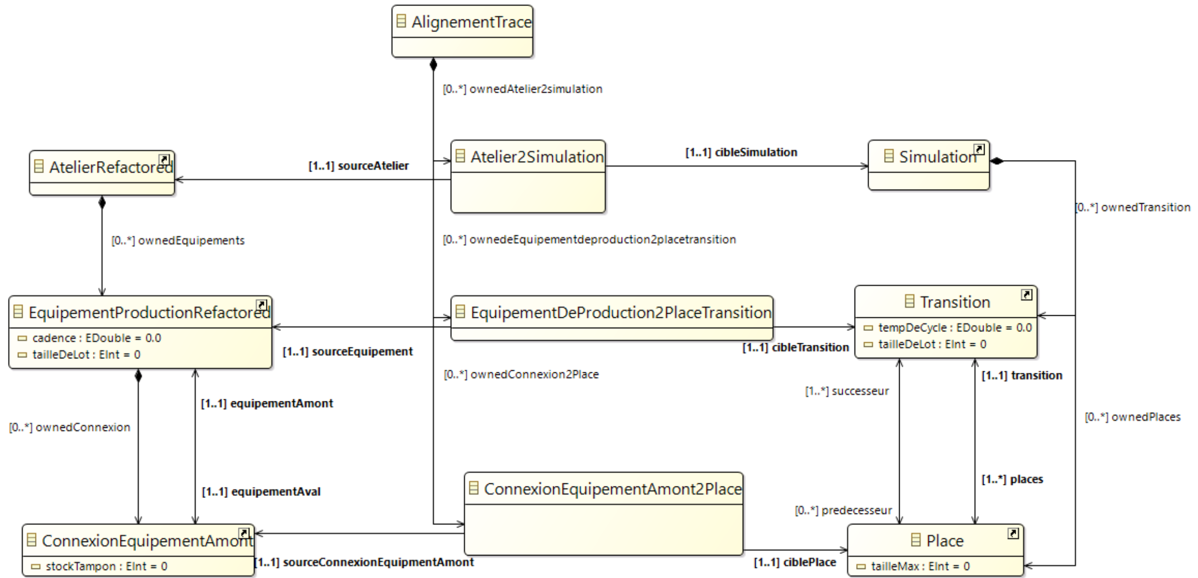


FIGURE 5.29 – Méta-modèle de trace pour l'étape d'alignement

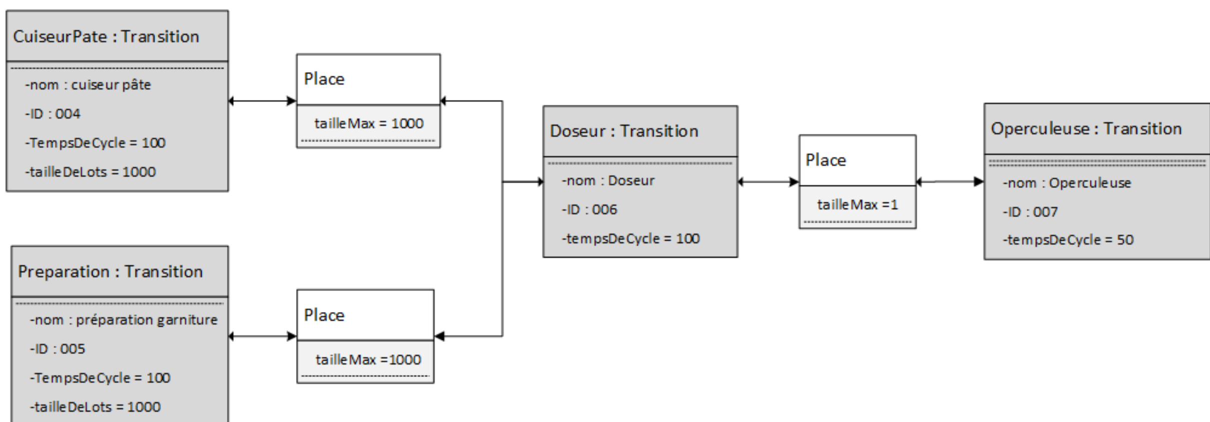


FIGURE 5.30 – Diagramme objets simplifié étape d'alignement

sens où elle établie un changement de domaine ; les concepts du domaine source sont traduis en concepts du domaine cible. Dans le cadre de notre méthode, l'étape d'alignement fournie la sémantique opérationnelle aux éléments sélectionnés et réorganisés du modèle source. A la fin de cette étape, le modèle obtenu n'est pas forcément syntaxiquement et sémantiquement conforme. Il peut manquer des concepts et des informations indispensables à l'exécution du modèle cible. Ces informations n'étant pas présentes dans le modèle source il faut pouvoir les ajouter. Les étapes d'enrichissement et d'adaptation vont permettre d'ajouter et d'adapter les informations

manquantes au modèle cible .

5.2.4 Enrichissement

Objectifs

La sémantique opérationnelle à laquelle nous voulons nous conformer dans cette étape est dépendante du domaine de simulation et des objectifs de vérification. Certaines informations relatives au domaine de simulation peuvent être absentes du domaine source. L'objectif de l'étape d'enrichissement est d'ajouter et d'initialiser les informations manquantes nécessaires à la simulation du modèle.

Formalisation

L'enrichissement est une transformation $G^{aligne} \overset{\rho,m}{\Longrightarrow} G^{enrichi}$ avec $\rho = (\emptyset \leftarrow k \rightarrow R, ca)$ et $n \models ca$ alors $G^{aligne} \subseteq G^{enrichi}$. $\mathcal{L}(TG^{enrichi})$ est un sous-type de $\mathcal{L}(TG^{aligne})$ au sens de [28] [29]. D'un point de vue formel, cette étape est l'étape inverse de l'étape de sélection. En appliquant les principes de l'étape de sélection au modèle enrichi, nous devons pouvoir reproduire le modèle d'entrée.

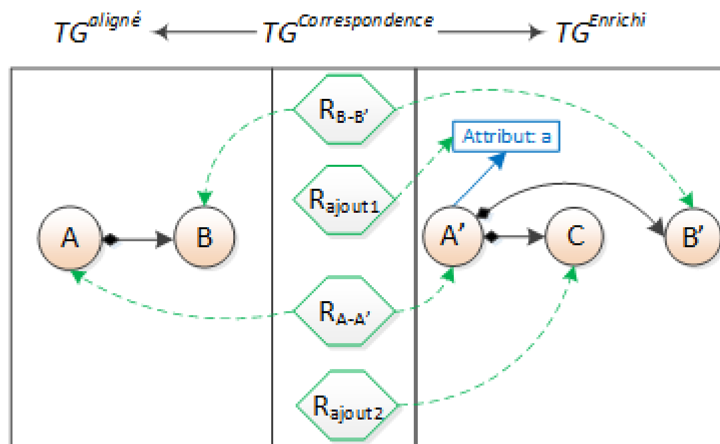


FIGURE 5.31 – Exemple de graphe triple Enrichissement

Dans l'exemple de la figure 5.31, nous voulons ajouter un attribut au concept de type A. Pour chaque concept de type B nous voulons ajouter un concept de type C. La règle triple pour la création de C comporte une condition d'application (figure 5.32) tel que $ca = vrai \iff \exists\, B' \in G^{Enrichi}$.

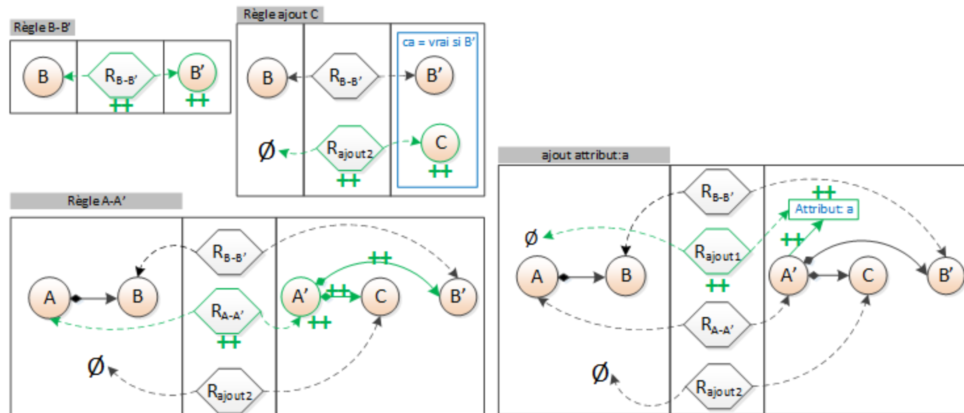


FIGURE 5.32 – Exemple de règle triple Enrichissement

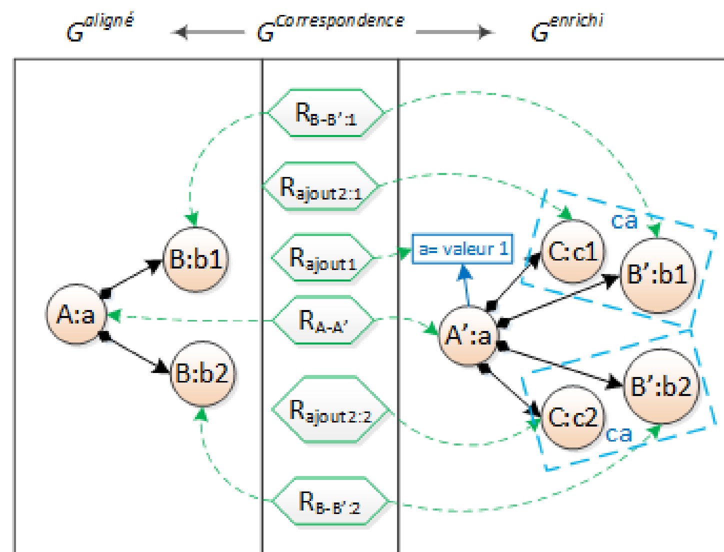


FIGURE 5.33 – Exemple d'instance Enrichissement

Pour chaque instance de B' (b1 et b2) de la figure 5.33, il a été créé une instance de concept C (c1 et c2).

Méthode

Dans cette étape nous ajoutons les classes, les attributs puis les objets et références manquants et appartenant au domaine cible et nécessaires à la simulation. Les objets et les attributs sont initialisés avec leurs valeurs par défaut. Si ces valeurs ne conviennent pas aux objectifs de simulation, elles pourront être modifiées lors de

l'étape suivante d'adaptation.

Exemple

Notre modèle n'est pas complet. Il manque des informations pour pouvoir le simuler. Dans cette étape d'enrichissement nous ajoutons les classes, les attributs ainsi que les instances nécessaires à la simulation :

- **Ajout de classes :** pour que le méta-modèle soit complet nous ajoutons la classe « *Jetons* » que nous relions à la classe « *Place* » par une relation d'agrégation (cf le méta-modèle cible de la figure 5.28).
- **Ajout d'attributs :** nous ajoutons les attributs *tempsDeSimulation* et *echelleDeTemps* à la classe *Simulation*.Nous ajoutons Une information sur la disponibilité de chaque équipement; attribut *disponibilité* dans la classe *Transition* . Cette information dont la valeur est comprise entre 0 et 1 permettra de simuler des arrêts de production liés aux aléas de production.
- **Ajout d'objets :** nous ajoutons des objets de type « *Place* » en amont des premiers équipements de l'unité de production. Ces « *Places* » sont initialisées avec 1000 « *Jetons* » permettant la production du premier lot sur les équipements de production. De même qu'il faut initialiser la simulation, il faut enregistrer les éléments de la simulation. Un objet de type « *Place* » est ajouté en fin de simulation pour stocker les objets de type « *Jeton* » produits pendant la simulation.

Le diagramme d'objets figure 5.34 présente une instance des objets du réseau de Petri après l'étape d'enrichissement. Les éléments issus de l'enrichissement sont représentés en vert sur le diagramme.

Discussion

Cette étape est fortement liée au domaine de simulation choisi. Elle explicite les informations nécessaires à la simulation qui sont absentes de la sémantique descriptives du domaine de modélisation source. Sans cette opération, il faudrait pour définir une sémantique opérationnelle, étendre le domaine de modélisation initial pour y ajouter les informations de simulation. Ce qui impliquerait de créer une nouvelle extension pour chaque domaine de simulation visé.

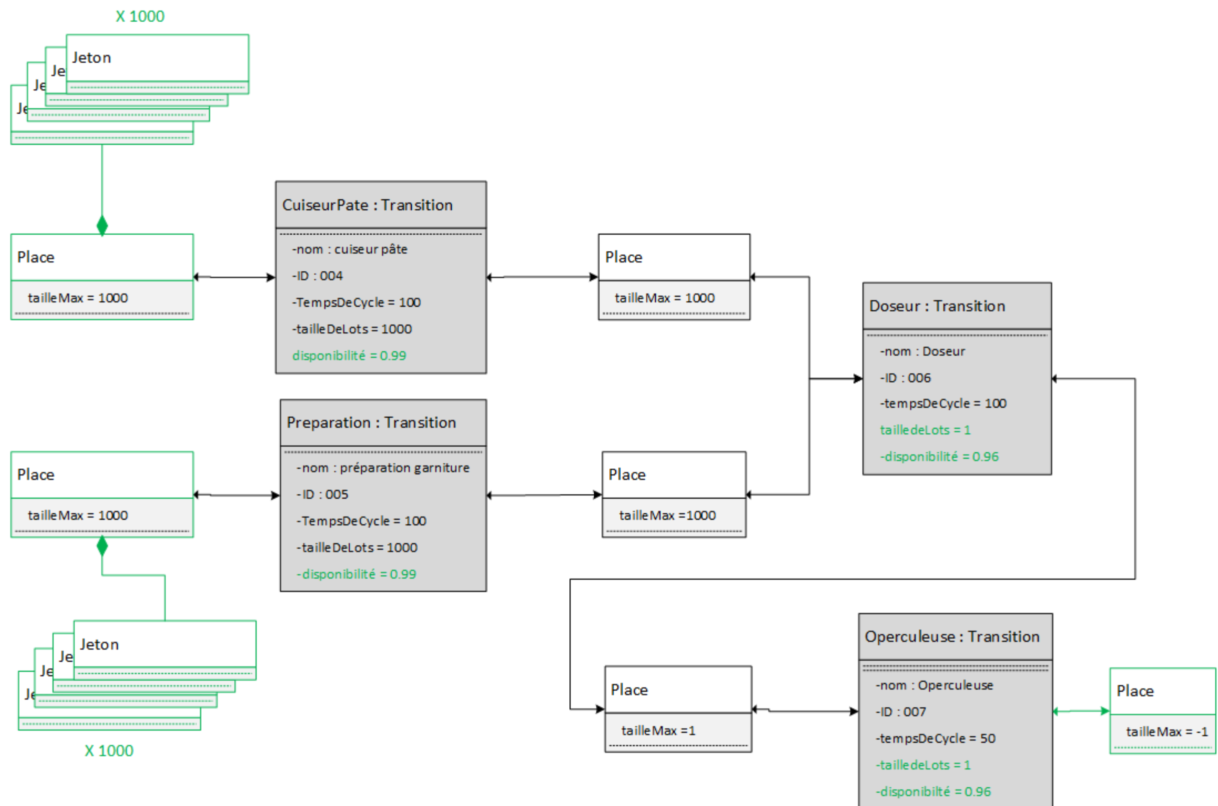


FIGURE 5.34 – Diagramme objet enrichissement simplifié

5.2.5 Adaptation

Objectifs

Dans cette dernière étape, le modèle est adapté pour être exécutable dans l'environnement de simulation défini par le méta-modèle cible. L'objectif de l'étape d'adaptation est de transformer le modèle afin de respecter les contraintes du domaine de simulation.

formalisation

Seuls les concepts sont réorganisés. Il n'y a ni ajout ni suppression d'information. Si $G = (V_G, E_G, s_G, t_G)$ est le graphe source de l'adaptation et $H = (V_H, E_H, s_H, t_H)$ son graphe cible, alors :

- $V_G = V_H$
- $E_G = E_H$

A la fin de cette étape, la syntaxe et la sémantique du modèle sont conformes au méta-modèle du domaine de simulation $\mathcal{M}(H) \models \mathcal{L}(TH)$. Le modèle obtenu est une sélection du modèle source ré-organisé, étendu par enrichissement et adaptation.

L'exemple de la figure 5.35 est une adaptation du graphe triple de la figure 5.31. Le concept de type C ajouté pendant l'étape d'enrichissement est contenue par A' mais doit contenir B'. Ici l'adaptation vise à « déplacer » l'arc de contenance $A' \rightarrow B'$ vers $C' \rightarrow B''$. Une condition d'application permet d'appairer les instances de C' et B" entre

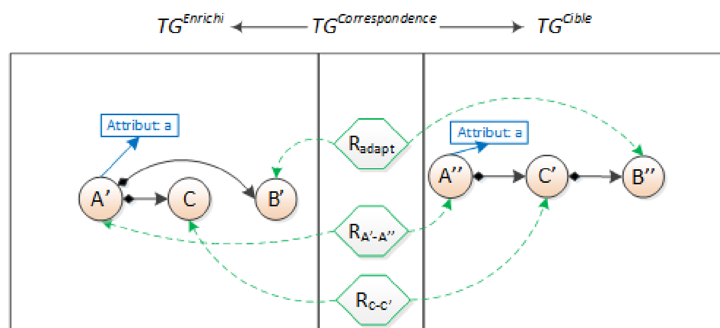


FIGURE 5.35 – triple graphe adaptation

elles (figure 5.36). Cette condition peut porter sur des attributs ou des relations mais elle ne doit pas être ambiguë pour éviter les erreurs d'appairage.

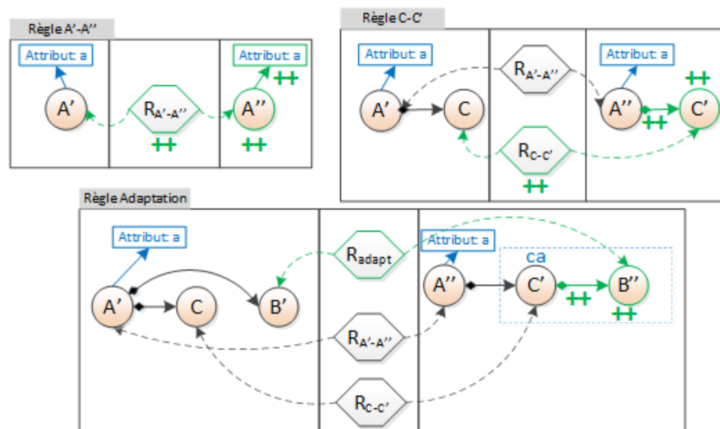


FIGURE 5.36 – Règle triple adaptation

La figure 5.37 montre un exemple d'instances de l'étape d'adaptation, les arcs de contenance entre les objets $a \rightarrow b1$ et $a \rightarrow b2$ sont déplacés respectivement vers $c1 \rightarrow b1$ et $c2 \rightarrow b1$.

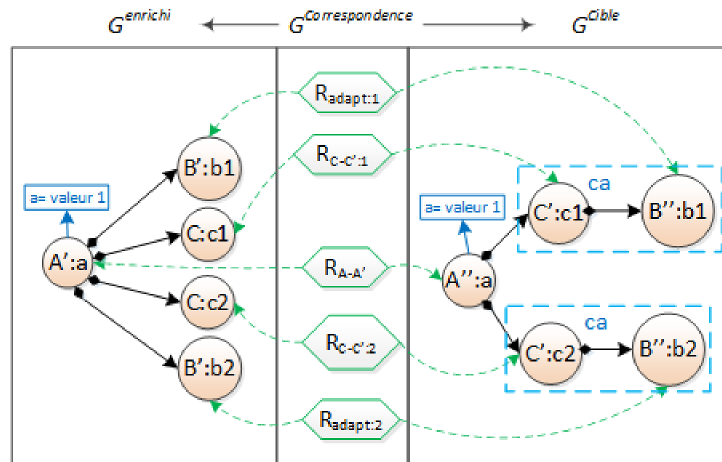


FIGURE 5.37 – instance de triple graphe adaptation

Méthode

Cette étape nécessite une analyse de la correction du modèle obtenu à l'étape précédente. A partir des erreurs rencontrées, le modèle est ajusté pour pouvoir être simulé. Dans cette étape les valeurs par défaut des objets et attributs peuvent par exemple être modifiées en fonction des besoins de la simulation.

Exemple

Sans adaptation, l'exécution du modèle abouti à des erreurs. Premièrement, lorsque le premier lot est produit, les jetons initiaux ne sont pas recréés : la simulation s'arrête. L'ajout d'une relation « prédécesseur - successeur » entre les places et les transitions initiales permet de régénérer ces jetons. Deuxièmement l'unité de productivité est exprimée en nombre de pièces par minute dans le modèle initiale. Cette unité est exprimée en temps de cycle pour l'ensemble du lot. Il faut donc convertir les données de l'attribut « temps de cycle » suivant la formule suivante : $tempsDeCycle = tailleDeLot \div tempsDeCycle$. La figure 5.38 montre le diagramme objets de la simulation après adaptation. Les éléments adaptés sont représentés en vert.

La figure 5.39 donne un exemple de syntaxe concrète pour la simulation de l'unité de production à l'aide de réseaux de Petri. Le diagramme représente l'état de la simulation à l'initialisation de celle ci. Chaque cercle représente une place avec le nombre de jetons présents à l'instant t, les rectangles représentent les transitions.

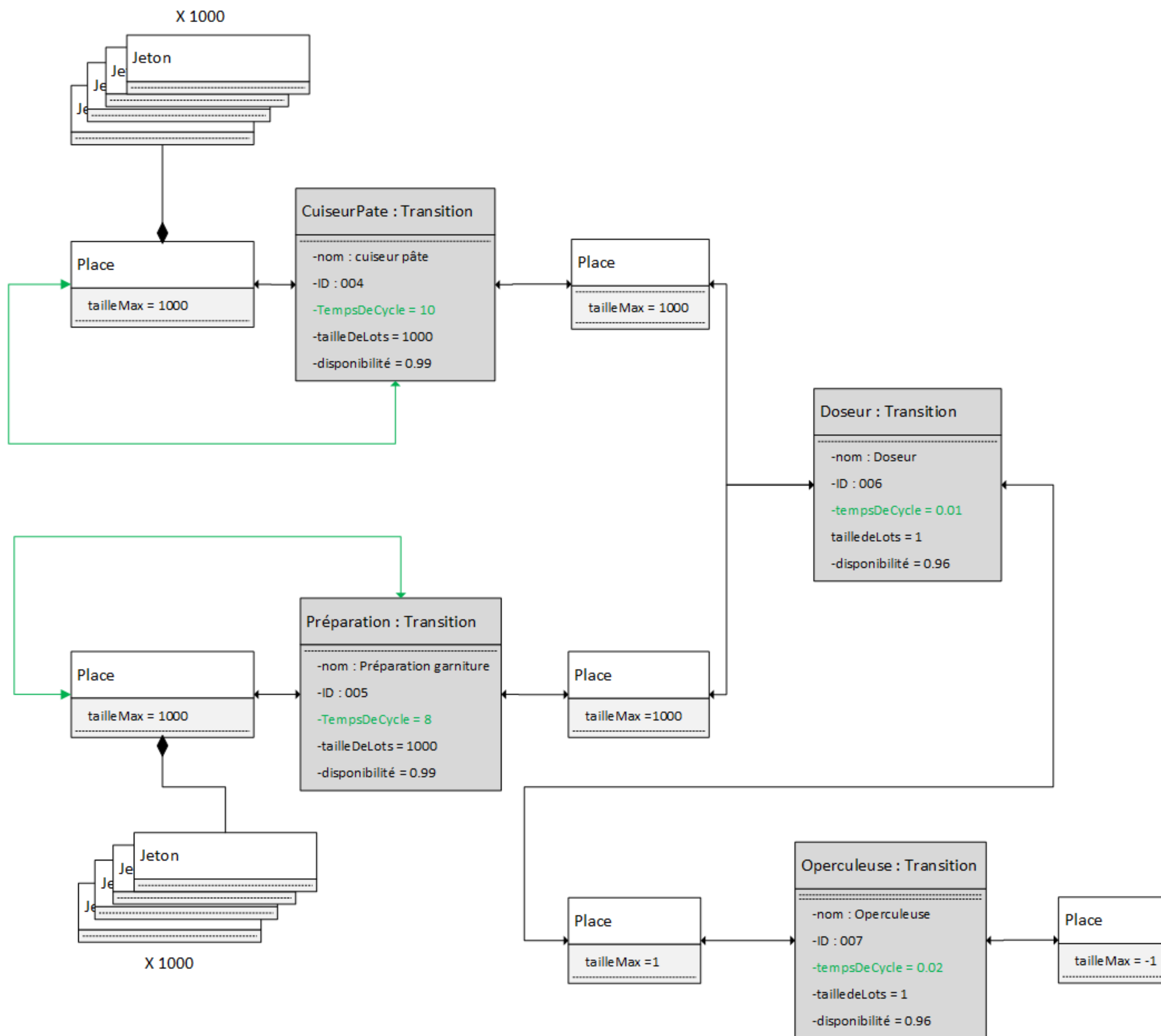


FIGURE 5.38 – Diagramme objet après adaptation

Discussion

L'adaptation permet au modèle et aux informations ajoutées de respecter les contraintes structurelles du domaine de simulation. Après l'étape d'adaptation, le modèle est correct et peut être simulé sans erreur.

5.2.6 Exécution du modèle de simulation

Une fois l'ensemble des étapes de transformation réalisé, nous pouvons effectuer la simulation du modèle pour l'exemple traité. L'exécution du modèle de simulation permet

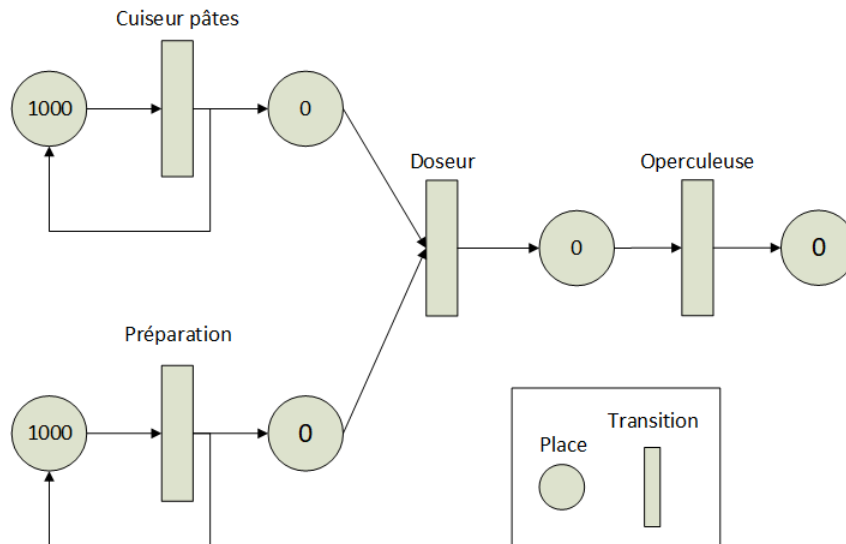


FIGURE 5.39 – Exemple de syntaxe concrète pour représenter la simulation en réseaux de Pétri

de mesurer la productivité et la disponibilité effective de l'unité de production. Dans sa configuration initiale l'unité de production modélisée en figure 5.6 produit en moyenne 46 unités par minute avec une disponibilité globale de 92,16 %. La simulation montre l'existence d'un goulot d'étranglement dans le flux de production. Le dernier équipement de l'unité (« l'operculeuse »), produit à 100% de sa capacité alors que les autres équipements produisent à 50% de leur capacité. La chaîne de traçabilité établie lors de chaque transformation permet de faire un lien entre les objets de type « Transition » de la simulation en réseau de Petri avec les objets de type « EquipementDeProduction » du domaine de modélisation. La figure 5.40 montre le lien de traçabilité entre l'objet « operculeuse » du domaine de simulation et l'objet « operculeuse » du domaine de modélisation.

A partir de cette première simulation un processus itératif peut être initié par les ingénieurs en charge de la conception pour trouver un optimum dans l'organisation de l'unité de production. Par exemple, le modèle est modifié pour ajouter une seconde operculeuse en parallèle de la première pour permettre d'atteindre une cadence nominale de 100 pièces par minute. Un troisième modèle est créé pour installer deux lignes de conditionnement en parallèle (doseuse + operculeuse). Les résultats montrent que le second modèle présente de meilleures résultats en terme de productivité. (cf tableau

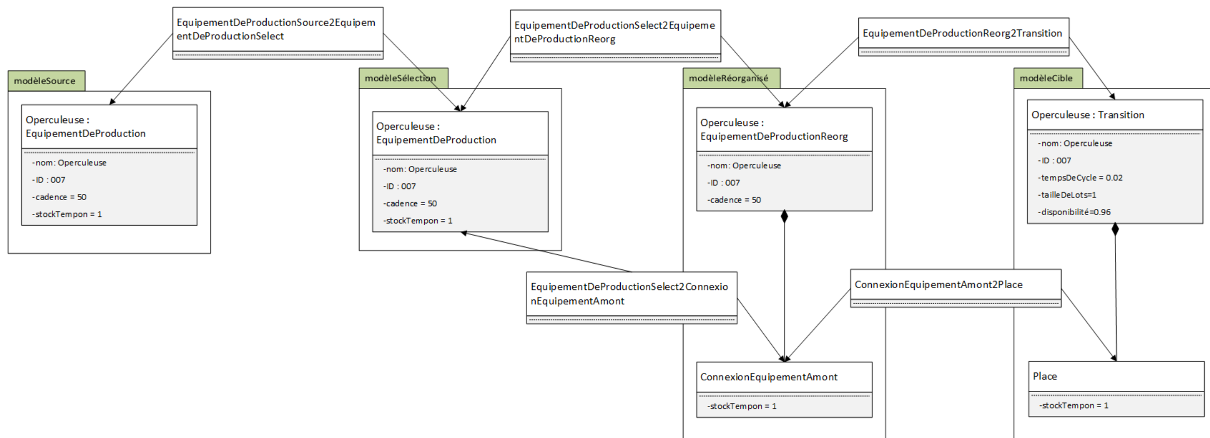


FIGURE 5.40 – Diagramme objet du lien de traçabilité

ci-dessous).

Résultats			
Configuration	Capacité max	Productivité (pcs/min)	Disponibilité globale
En flux	50	46	92,16%
Deux operculeuses	100	91,12	91,12%
Deux lignes parallèles	100	90,93	90,93%

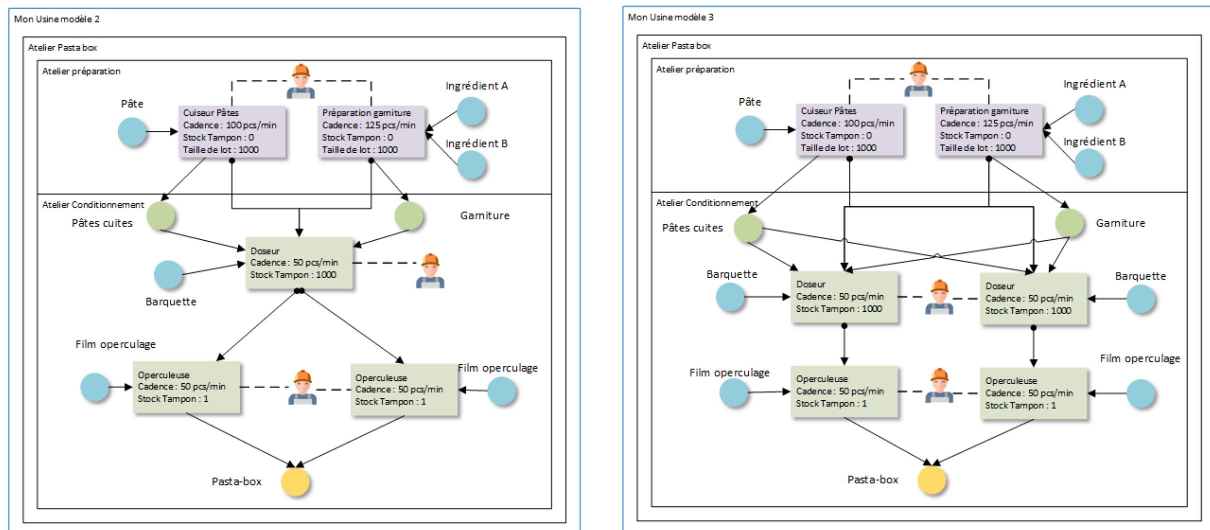


FIGURE 5.41 – Représentation des Modèles 2 et 3 à l'aide de la syntaxe concrète

5.3 Conclusion du troisième chapitre

Dans ce chapitre nous avons proposé une chaîne de transformation basée sur cinq transformations consécutives :

- sélection,
- réorganisation,
- alignement,
- enrichissement,
- adaptation.

L'idée de l'approche est de séparer chaque préoccupation du processus de transformation :

- la première transformation impacte la définition de la sémantique opérationnelle en limitant sa zone de définition : l'hypothèse implicite est que les concepts non considérés par la sélection sont neutres et n'impactent pas les résultats de la simulation,
- la deuxième transformation a un impact sur la définition sémantique en réorganisant les concepts sources pour les adapter aux concepts du domaine cible,
- la troisième transformation traduit les concepts du domaine source en concepts du domaine cible, En alignant les concepts sources sur les concepts du domaine de simulation, nous donnons une interprétation des concepts sources pour pouvoir les simuler,
- la quatrième transformation apporte de la sémantique en ajoutant des informations manquantes nécessaires à la simulation et à l'analyse. Cette étape est une étape complémentaire de modélisation, dans le sens où elle apporte une description complémentaire pour la simulation du comportement du modèle source.
- la cinquième transformation adapte le modèle pour qu'il soit conforme à la sémantique du domaine cible.

En appliquant les cinq étapes de transformation, nous obtenons une sémantique opérationnelle permettant la simulation d'une sélection d'un modèle source réorganisé et adapté auquel nous avons ajouté des informations pour la simulation. Chaque étape joue un rôle déterminant dans la définition de la sémantique opérationnelle.

La traçabilité est un point clé pour interpréter les résultats d'analyse du domaine de simulation dans le domaine de modélisation en créant un lien entre les concepts cibles et les concepts sources. L'utilisation de grammaires de graphes triples pour réaliser les

transformations permet d'exploiter les graphes de correspondance pour réaliser une chaîne de traçabilité entre les instances du modèle de simulation et les instances du modèle d'ingénierie. En remontant cette chaîne de traçabilité, toute incohérence détectée lors de la simulation peut être interprétée au regard du modèle source d'ingénierie.

CHAPITRE 6

APPLICATION

6.1 Application à la vérification d'un modèle de drone.

6.1.1 Introduction

Au cours des trois années de thèse, nous avons expérimenté et amélioré notre méthode de transformation de modèles en construisant différents prototypes d'outils ciblant différents domaines de simulation. Certaines de ces expérimentations ont donnée lieu à des publications et présentations dans des conférences :

- Application de la méthode ModelRun pour la vérification de modèles de dataflow avec Simulink [102].
- Application de la méthode ModelRun pour la vérification de modèles de dataflow et machines à états par simulation des machines à états[103].
- Application de la méthode ModelRun pour la vérification de modèles de dataflow et machines à états par parcours de graphe[104].

Nous présentons dans le chapitre application, la méthode ModelRun appliquée à la transformation d'un modèle de Drone « **PythaDrone** » modèlisé dans Capella. Le modèle est transformé en un modèle permettant la simulation et la vérification des propriétés relatives aux interactions entre le data-flow, les machines d'états et modes et les scénarios. Nous présentons dans ce chapitre : le modèle Capella de « **PythaDrone** », l'outil de vérification par simulation, l'application de ModelRun ainsi que des résultats de simulations.

6.1.2 PythaDrone

PythaDrone est une extension du projet Moïse de l'IRT Saint Exupéry. Ce projet modélise un drone aérien polyvalent conçu pour des missions d'inspection et de contrôle. En fonction de sa configuration, plusieurs types de missions peuvent lui être confiés,

comme le contrôle de l'état extérieur d'un avion ou bien l'inspection de cultures d'exploitations agricoles. Pythadrone a été modélisé avec ARCADIA/Capella. Pour la suite nous considérerons le modèle de drone avec une configuration dédié au contrôle et à l'inspection visuel de l'état extérieur d'avions (« Aircraft Visual Inspercion » A.V.I.) . La modélisation a pour point de départ une analyse opérationnelle et est raffinée à travers une analyse système, une architecture logique pour aboutir à une architecture physique. (cf figure 1.14 du chapitre 1.4.2 dédié à la méthode ARCADIA). PythaDrone est représenté à l'aide de 223 diagrammes différents. Nous ne présentons que quelques diagrammes permettant de présenter et d'illustrer les fonctionnalités de la partie A.V.I. de PythaDrone.

6.1.3 Éléments d'analyse opérationnelle

Pour rappel, l'analyse opérationnelle permet d'identifier les entités, les acteurs, les rôles et les activités que ces derniers doivent effectuer. On y modélise les besoins opérationnels des différents acteurs, ainsi que ce qu'il doivent accomplir dans le cadre de leur travail. L'analyse opérationnelle est indépendante de toute solution. Le diagramme de la figure 6.1 modélise les capacités opérationnelles pour les missions d'A.V.I. Le diagramme montre que les acteurs (service maintenance) ont besoin de réaliser des missions d'inspection visuel dans des espaces extérieurs *« Conduct aircraft outdoor inspections »* ainsi que dans des espaces fermés *« Conduct aircraft indoor inspections »*.

Chaque capacité est précisée à l'aide d'un diagramme *« Operational Activity Interaction Blank (OAIB) »*. La figure 6.2 présente le diagramme OAIB pour la capacité *« Conduct aircraft outdoor inspections »*. Il y est modélisé les différentes taches des inspections quotidiennes réalisées sur un avion : préparation des inspections quotidiennes, inspection des différents éléments de l'avion (fuselage, ailes, train d'atterrissage, réacteurs, éventuelles traces d'impact)

D'autres diagrammes (que nous ne présentons pas ici) précisent les besoins des différents acteurs par exemple : préparation des plans d'inspection, analyse des résultats et établissement des diagnostics pour le service maintenance.
L'ensemble des diagrammes de l'analyse opérationnelle permet d'avoir une vue complète des besoins des utilisateurs dans le cadre de leurs missions d'A.V.I.

FIGURE 6.1 – Modélisation des capacités opérationnelles et des acteurs du système A.V.I.

FIGURE 6.2 – Modélisation de la capacitée opérationnelle « inspection extérieur » réalisée par les acteurs du système A.V.I.

6.1.4 Éléments d’analyse du besoin système

L’analyse opérationnelle a permit de préciser les besoins des utilisateurs. Dans l’analyse des besoins du système, l’ingénieur système définit la contribution du système pour réaliser les missions qui lui sont attribuées. Cette définition est réalisée sous la forme d’analyses fonctionnelles et non fonctionnelles des besoins décrivant les fonctions que doit réaliser le futur système en lien avec l’activité des différents acteurs. A cette étape de la modélisation on ne fait aucune hypothèse sur ce que sera le futur système. Le diagramme de la figure 6.3, précise la mission principale, ses capacités et le périmètre du future système. Le périmètre du système d’A.V.I. est modélisé à l’intérieur du rectangle jaune *« AVI »*. Les capacités à l’extérieur du rectangle sont réalisées par les acteurs en interaction avec le système. Ainsi la mission principale du système est d’améliorer la fiabilité et l’efficacité des inspections visuelles d’avions en les réalisant plus rapidement et de manière plus précise et plus exhaustive. Cette mission se matérialise sous la forme de trois capacités :

- collecter les informations relatives aux inspections visuelles des avions,
- fournir des diagnostiques et des preuves suite aux inspections,
- fournir des moyens d’analyses post-inspection.

Ces capacités permettrons entre autre, aux différentes parties prenantes d’acquérir les données d’inspections (photos, vidéos, scans), de les analyser et de les archiver.

Les capacités sont ensuite développées et précisées avec un ensemble de diagrammes tel-que la description de chaînes fonctionnelles, d’architecture système, de scénarios et de machines d’états et modes.
Avec les analyses opérationnelles et systèmes nous avons une vue complète des exigences client et des besoins du système, l’ingénieur système peut commencer à définir des éléments de solution technique en réponse à ces besoins.

6.1.5 Éléments d’architecture logique

L’architecture logique est la définition d’une architecture de principe qui pose les grands choix de la solution. L’architecture logique est une première vision générale du futur système. La figure 6.4 modélise les principaux composants du systèmes (rectangles bleus) et les fonctionnalités qu’ils leurs sont associées (rectangles verts) . Trois composants principaux sont modélisés :

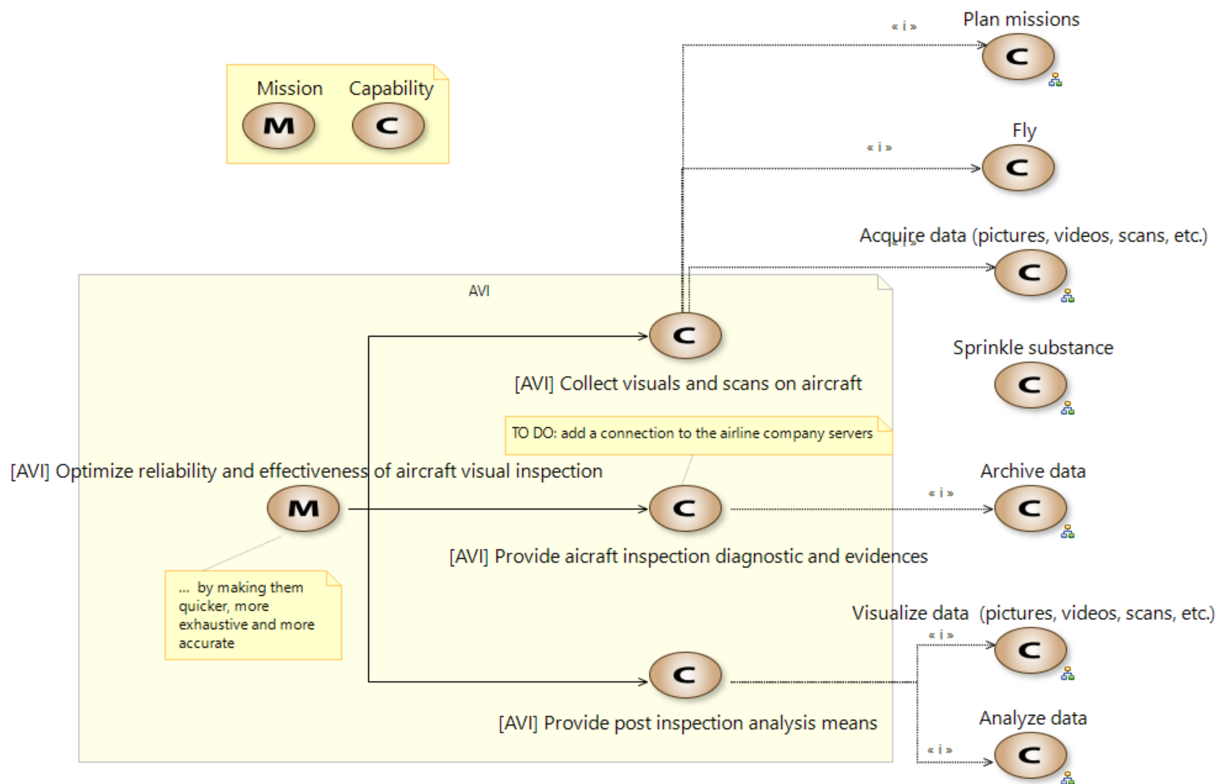


FIGURE 6.3 – Modélisation de la mission et des capacités opérationnelles « d'inspection extérieur » du système A.V.I.

- le « Control desk » contient les fonctions relatives à la gestion et au stockage des données,
- le « Remote control » acquière les ordres du pilote de drone,
- le « Drone » concentre les fonctionnalités d'un drone autonome. Ce dernier est composé de quatre sous-composants :
  - — le « Controler » centralise les fonctions nécessaires au fonctionnement autonome du drone,
  - — le « Monitoring » contrôle les paramètres de vol du drone,
  - — le « Payload » gère la capture des données d'inspection,
  - — les « Actuators-Propellers » contrôlent les hélices du drone.

D'autres diagrammes de l'architecture logique modélisent les échanges fonctionnels entre fonctions, les interactions entre le système et son environnement, les relations entre le système et les différents acteurs. A l'issue de la définition de l'architecture logique, nous avons une architecture de principe construite en réponse aux besoins

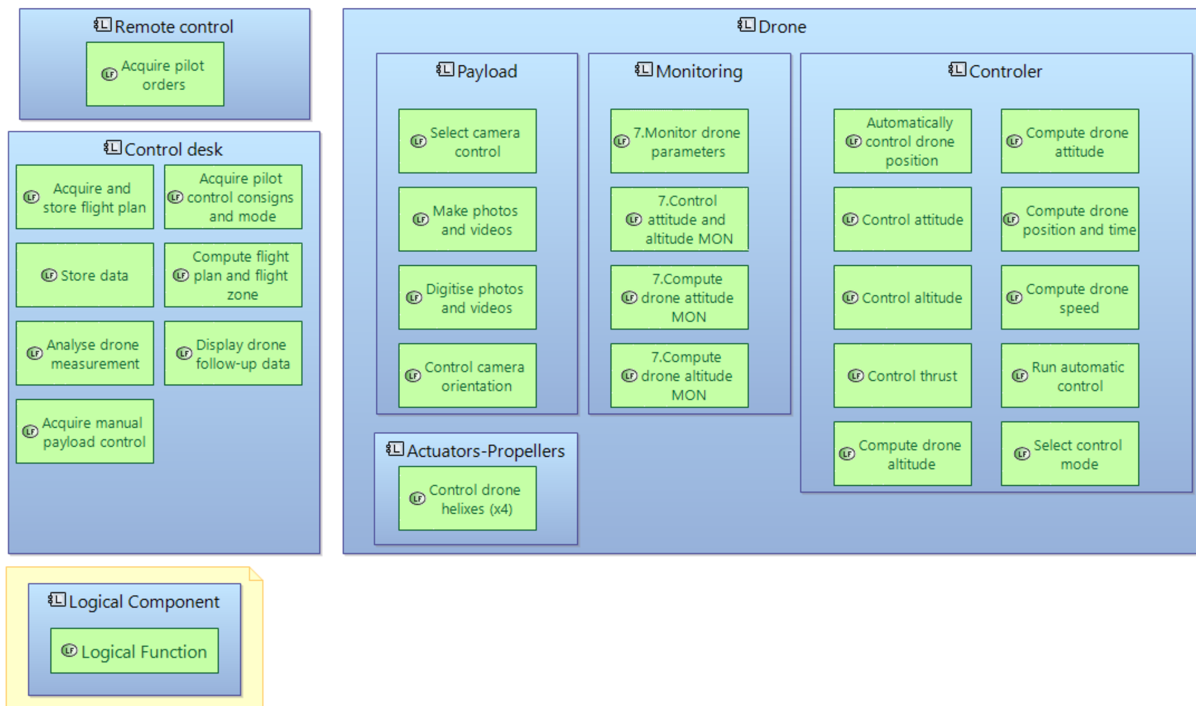


FIGURE 6.4 – Modélisation des composants et fonctions du système A.V.I. « inspection extérieur » du système A.V.I.

spécifiés dans les définitions de l'analyse opérationnelle et de l'analyse du besoin système. Cette architecture de principe va être précisée avec l'architecture physique.

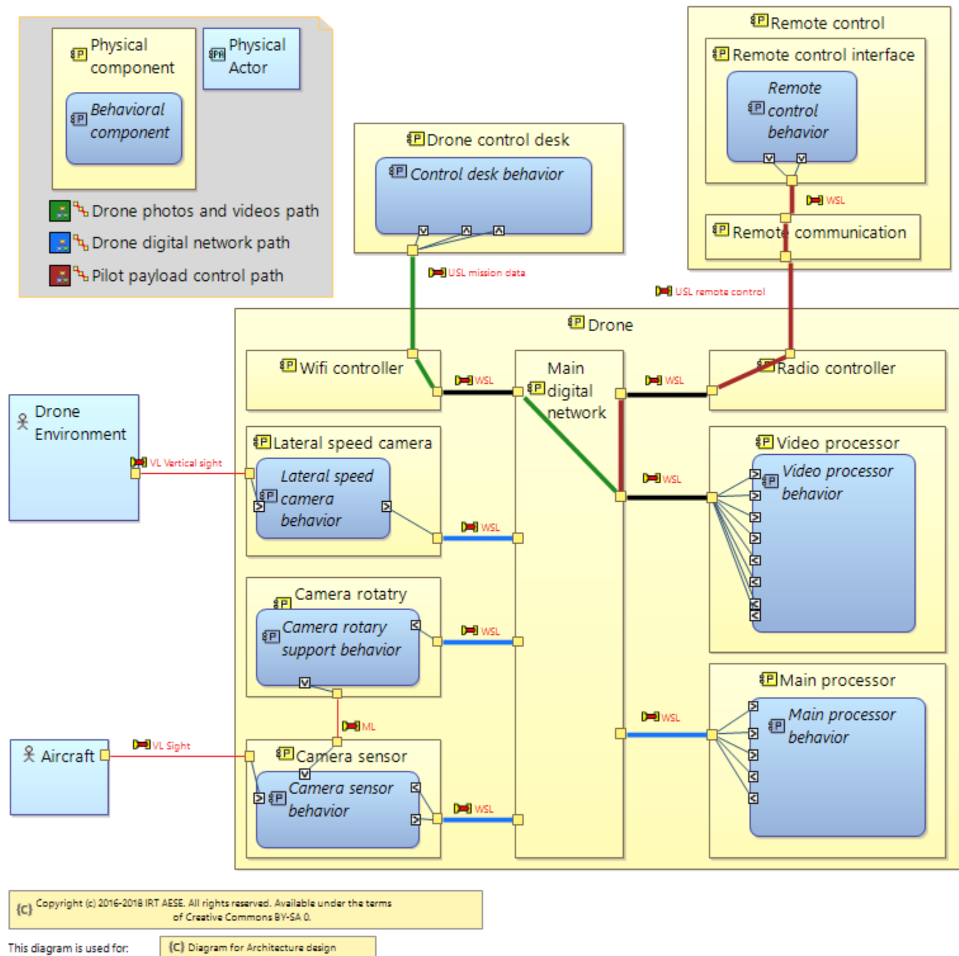
6.1.6 Éléments d'architecture physique

L'architecture physique correspond à une architecture finalisée. Les éléments de l'architecture logique sont raffinés et définis à un niveau de détail suffisant pour spécifier les développements de tous les sous-systèmes à réaliser. L'objectif de l'architecture physique est aussi de spécifier le comportement attendu du futur système.

Modélisation des composants et des fonctions

Dans cette phase de conception, sont définis les composants physiques et les composants comportementaux (composants du système chargés de réaliser une partie des fonctions dévolues au système). La figure 6.5 présente un des nombreux diagrammes d'architecture physique. La vue illustre les relations entre composants phy-

siques et comportementaux impliqués dans le déploiement de la capture vidéo. Les rectangles jaunes représentent les composants physiques, les rectangles bleus foncés représentent les composants comportementaux, les rectangles bleus ciels les acteurs en interaction avec le système. Sur cette vue sont aussi représentés les chemins de flux d'informations.

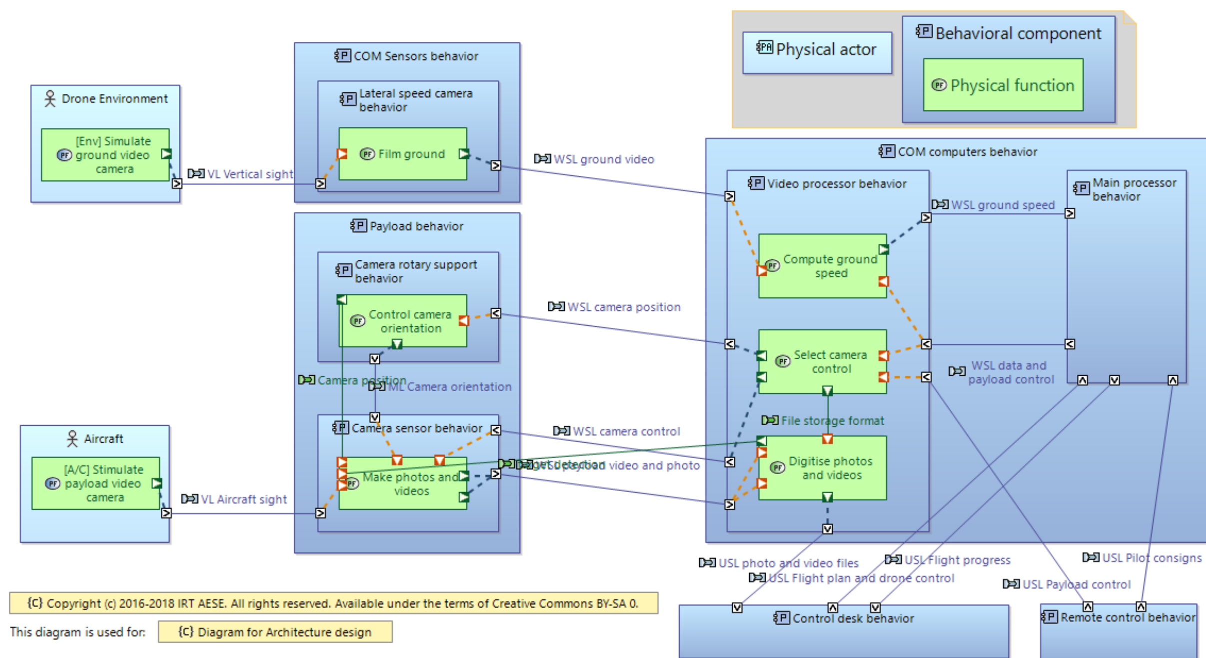


Copyright (c) 2016-2018 IRT AESE. All rights reserved. Available under the terms of Creative Commons BY-SA 0.

FIGURE 6.5 – Relations entre composants de l'architecture physique

Les autres vues décrivent les composants impliqués dans le fonctionnement de la régulation de la position et de la vitesse du drone, le fonctionnement des hélices du drones, la fourniture d'énergie, etc... Ces diagrammes permettent d'avoir une vue haut niveau exhaustive de l'architecture physique du système. Chaque composant comportemental est précisé en modélisant l'ensemble des fonctions qui le composent. La figure 6.6 montre un diagramme d'allocation de fonctions pour les composants com-

portementaux. Nous retrouvons sur ce type de diagramme d'architecture physique les composants comportementaux (bleu foncé) et les acteurs (bleu ciel). Les fonctions du système apparaissent en vert. Les liens entre les composants modélisent les interactions entre composants.



{C} Copyright (c) 2016-2018 IRT AESE. All rights reserved. Available under the terms of Creative Commons BY-SA 0.

FIGURE 6.6 – Allocation Fonctionnelle du « *Video Processor* »

La figue 6.6 est une vue de l'allocation fonctionnelle du système vidéo du drone. Un premier capteur vidéo filme le sol et permet après traitement d'estimer la vitesse du drone par rapport à ce sol. Le second capteur vidéo est en charge de filmer et photographier l'avion à contrôler. Un dernier niveau de raffinement permet de modéliser l'ensemble des fonctions du système indépendamment des composants auxquels elles sont rattachées. La figure 6.7 présente une vue des fonctions impliquées dans la prise d'images vidéos pour l'inspection d'avions. Les fonctions du système sont représentées par des rectangles verts les fonctions réalisées par les acteurs sont représentées par des rectangles bleus. Les fonctions modélisent une action, une opération ou un service réalisé par le système ou un de ses composants. Elles sont reliées entre elle par des « échanges fonctionnels » (traits noirs entre fonctions). Ces derniers modélisent des interactions possibles entre une fonction source et une fonction cible. Les échanges fonctionnels sont orientés : ils relient un port de sortie d'une fonction (carré vert foncé) au port d'entrée d'une fonction (carré rouge).

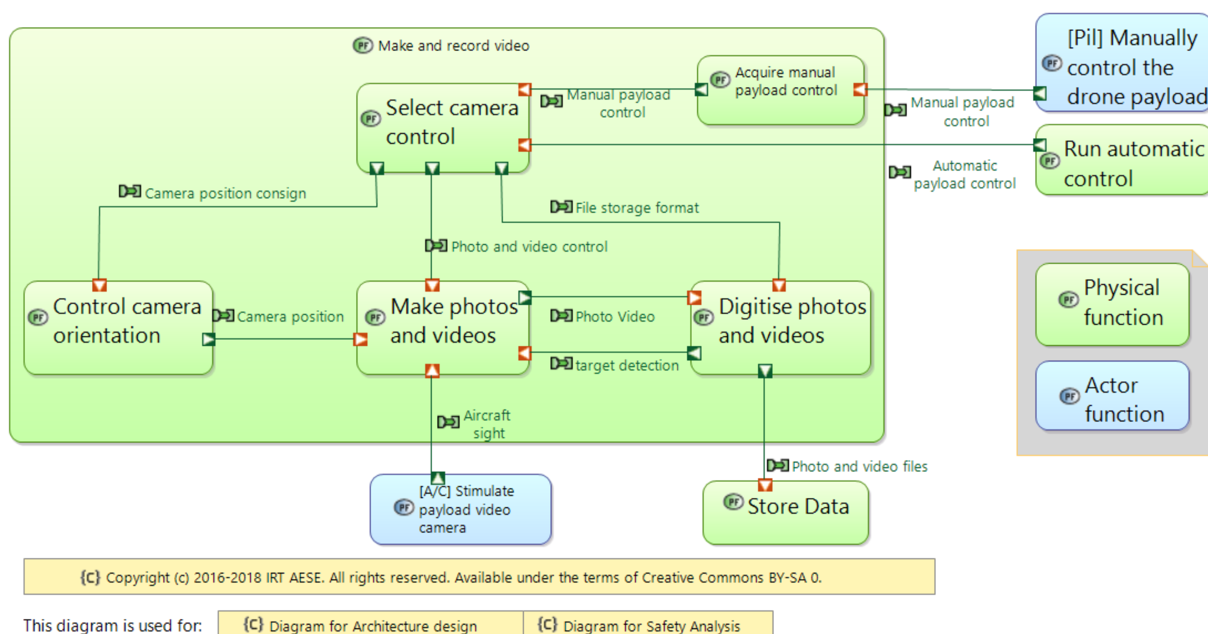


FIGURE 6.7 – Vue d'une partie du data-flow du système du drone centré sur le processus d'enregistrement vidéo

L'ensemble des fonctions et leurs échanges modélisés dans les différents diagrammes de l'architecture physique, constituent le *data-flow* de l'architecture physique du futur système.

Diagramme transversaux

L'architecture physique du système peut être complétée par un ensemble de vues corrélées entre elles : diagrammes de chaînes fonctionnelles, de scénarios, de machines de modes et états. Ces diagrammes permettent de préciser des points de vue et des contextes spécifiques d'exécution de parties du système. La figure 6.8 présente une schématisation des relations entre le *data-flow* d'une architecture physique et différents contextes comportementaux spécifiées par des machines de modes et des scénarios. Les différents concepts de l'architecture physique sont retranscrits dans les scénarios ainsi que dans les machines d'états et modes pour spécifier des comportements dynamiques du système.

La figure 6.9 présente une machine de modes, cette vue modélise l'enchaînement des différents modes de fonctionnement du système PythaDrone. Elle permet de spécifier les concepts (échanges fonctionnels, fonctions) activant les transitions ainsi que

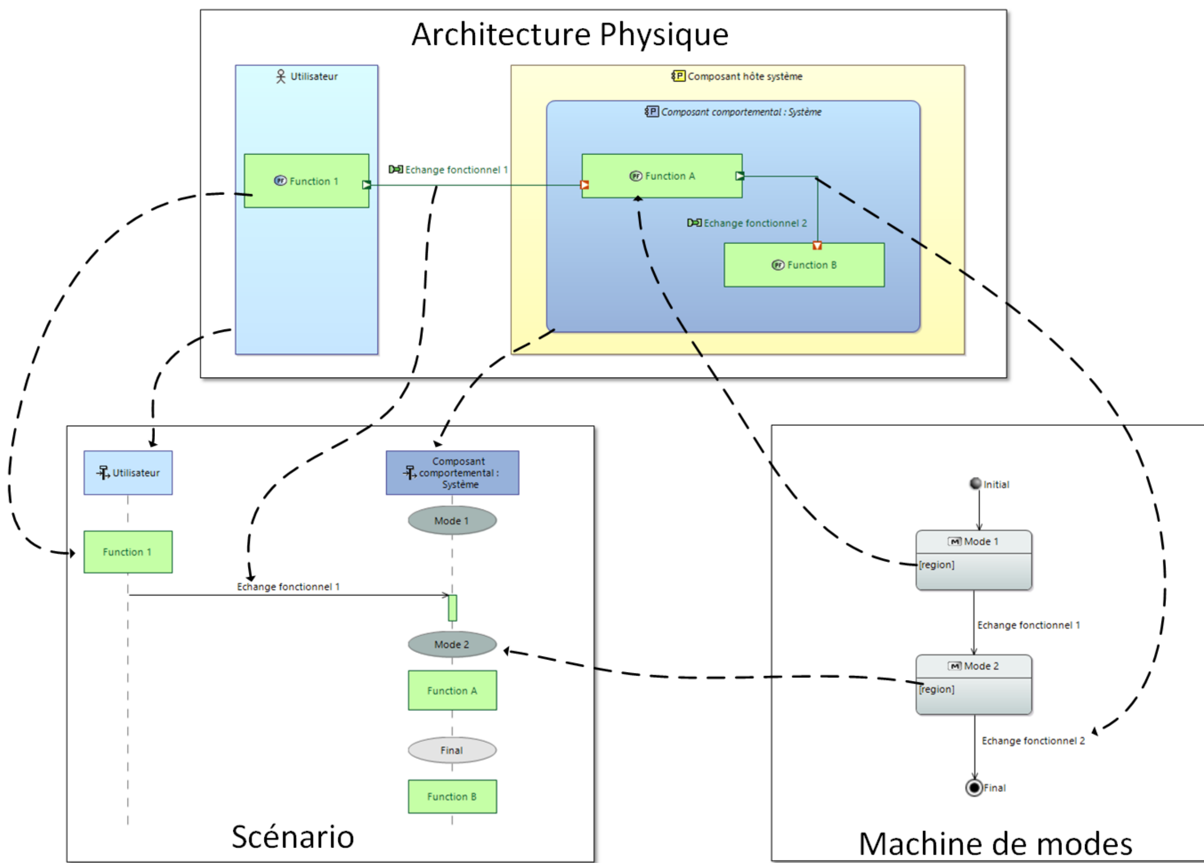


FIGURE 6.8 – Relation entre concepts à travers différents diagrammes de l'architecture physique

les fonctions activées par les différents modes.

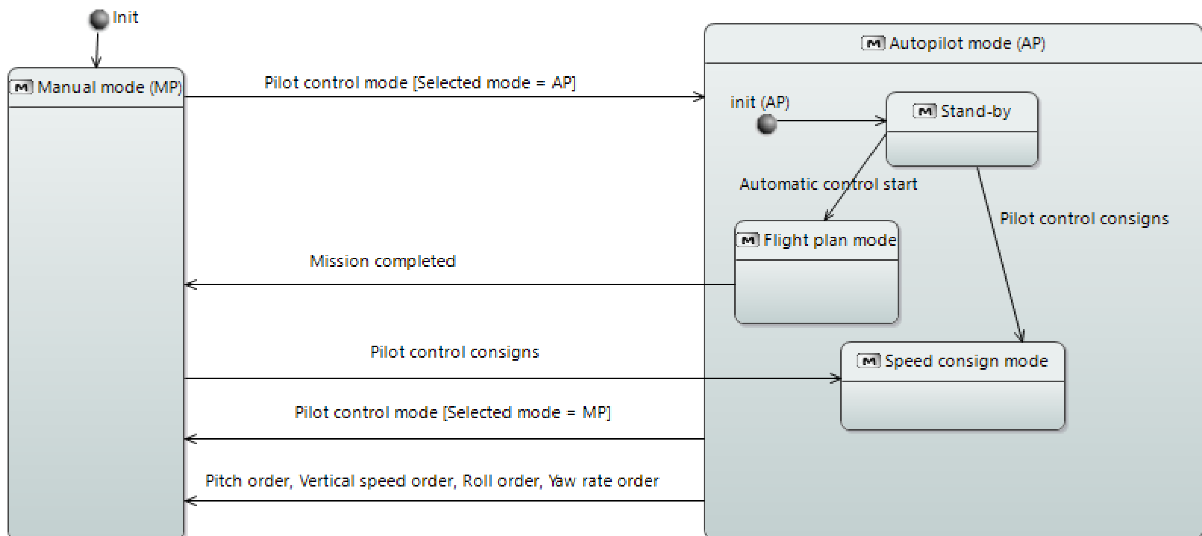


FIGURE 6.9 – Machine de modes décrivant le comportement des différents modes du système PythaDrone

Le comportement du système dans sa globalité peut être spécifié par un ensemble de plusieurs machines d'états et modes, on considère alors une exécution concurrente de ces machines d'états et modes.

Spécification en vue de la réalisation du système

L'ensemble des diagrammes et des vues de l'architecture physique nous permet d'obtenir une spécification exhaustive de l'architecture du système ainsi que de son comportement. Ces spécifications sont partagées avec les différents métiers en charge de la réalisation du système, que se soit en interne à une entreprise, ou en lien avec des sous-traitants.

Au final, le système PytaDrone est composé de 129 fonctions et de 288 échanges fonctionnels, structurés dans 97 composants. Cet ensemble forme un modèle détaillé de toutes les fonctionnalités du système. PythaDrone est ainsi constitué de trois composants principaux :

- Le drone lui-même disposant de toutes les fonctionnalités lui permettant un fonctionnement autonome suivant un plan de vol pré-établie ou un fonctionnement commandé par un pilote au sol.
- Le contrôle à distance permettant au pilote de donner des consignes au drone.

- La partie contrôle au sol en charge de l'acquisition, le stockage et l'analyse des données envoyées par le drone ainsi que la préparation des missions du drone.

6.1.7 Problématique de cohérence des éléments de modèles entre eux.

L'architecture physique est la base de la spécification du système en vue de sa réalisation concrète. Cependant à mesure que les diagrammes et les points de vue se multiplient, il devient difficile pour l'équipe projet d'assurer la cohérence de l'ensemble des éléments de modèles entre eux. Pourtant, le coût de la correction d'un défaut de conception d'un système augmente considérablement au fil des étapes d'ingénierie. Pour limiter les erreurs futurs de réalisation, il est important de pouvoir vérifier la définition du système dés l'étape de conception de l'architecture physique . Quels types d'incohérences pouvons-nous détecter à ce stade précoce de l'ingénierie ? Voici quelques exemples d'incohérences pouvant être détectées dans un contexte décrit par un *data-flow* et un ensemble de machines d'états et modes, de chaînes fonctionnelles et de scénarios :

- Les fonctions ou les échanges fonctionnels requis dans un scénario ou une chaîne fonctionnelle d'un contexte ne sont pas disponibles dans la combinaison de modes et d'états de ce contexte.
- Les fonctions requises dans un contexte ne sont pas disponibles parce que certaines autres fonctions nécessaires pour fournir leurs apports ne sont pas disponibles.
- Les fonctions ou les échanges requis dans un contexte ne sont pas disponibles parce que les composants ou les liens de communication qui les mettent en œuvre ne sont pas disponibles.
- Les fonctions et les échanges fonctionnels requis dans le contexte ne sont pas correctement ordonnés (soit en terme de dépendance fonctionnelle, soit en terme de temps dans un scénario, soit en terme de liens de séquence dans une chaîne fonctionnelle)
- Aucun déclenchement d'une transition à partir d'un mode ou d'un état actuel n'est possible, car la fonction produisant le déclenchement n'est pas disponible.
- Dans un scénario, un mode ou un état mentionné n'est pas réalisable.
- Dans un scénario, les fonctions requises après la mention d'un mode ou d'un

état ne sont pas disponibles.

Toutes ces vérifications impliquent de considérer la globalité du modèle dans un aspect de comportement dynamique. Cependant des vérifications exhaustives ou formelles du modèle nécessiterait une sémantique plus précise et une description plus détaillée du comportement de chaque fonction du modèle, ce qui impliquerait plus de contraintes sur la sémantique de modélisation à un stade de l'ingénierie où une certaine liberté et réactivité est requise. Ce qui aurait pour conséquence de mettre l'accent sur la conception détaillée plutôt que sur les capacités globales du système, comme recommandé dans la méthode ARCADIA. Pour toutes ces raisons nous voulons de manière pragmatique trouver certaines incohérences à faible coût, le plus tôt et le plus rapidement possible, de manière à éliminer les plus évidentes avant d'entrer, dans une vérification plus précise, lors de phases ultérieurs de développement. Pour ce faire nous choisissons de simuler les éléments de modèles dans ce que nous pourrions qualifier de « très haut niveau » de comportement. Dans ces simulations, nous restons agnostique des comportements spécifiques de chaque fonction (qui ne sont pas encore précisées à cette étape d'ingénierie). La méthode de transformation de modèles ModelRun, que nous avons présenté au chapitre 5.1 de cette thèse, permet de cibler des domaines de simulation exogènes au domaine de modélisation. Les étapes successives de notre méthode nous guident pour préciser une sémantique d'exécution à notre modèle source en levant ses différentes ambiguïtés sémantiques. Enfin, les liens de traçabilité permettent de faire un lien non équivoque entre les concepts cibles et sources.

6.2 Modèle cible de simulation

6.2.1 Objectifs de vérification

L'objectif de la vérification est de rechercher des éléments d'incohérence entre les éléments du modèle du système modélisé avec Capella. Pour les vérifications, nous considérons les interactions entre le *data-flow* dans son ensemble, les machines d'états et modes et les scénarios. Nous pouvons alors vérifier les points suivants :

1. Vérification de la cohérence entre les *data-flow* et les machines d'états et modes :
   - Le modèle est considéré cohérent si pour chaque état de la machine à états le data-flow est complet.

- Le data-flow est considéré complet si toutes les fonctions actives dans un état, ont toutes des données disponibles sur leurs entrées.

2. Vérification de l'absence de « dead lock » :
   - La machine à états n'est jamais bloquée dans un état sans transition disponible pour en sortir.
3. Vérification de la faisabilité de chaque transition de la machine à états :
   - Disponibilité des échanges fonctionnels du *data-flow* devant déclencher la transition.
4. Vérification de faisabilité du déroulé d'un scénario :
   - Vérification des interactions entre *data-flow* et machine d'états et modes tels que décrits dans les scénarios.

Condition de validité

Nous considérons qu'**une fonction active est valide si tous les échanges fonctionnels entrants sont disponibles**

Le **data-flow est considéré valide dans un état de la machine à états si toutes les fonctions actives sont valides** (cf figure 6.10).

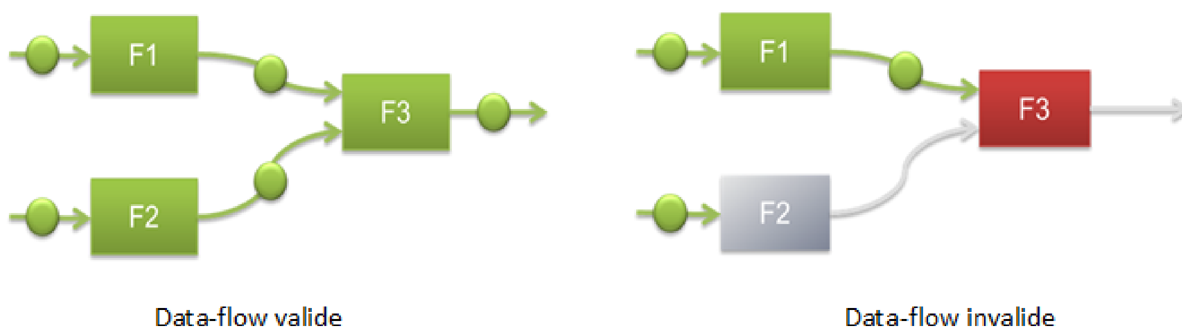


FIGURE 6.10 – Condition de validité d'une fonction du *data-flow*

La figure 6.10 schématise deux exemples de *data-flow*. Pour l'exemple de *data-flow* valide, les fonctions F1, F2 et F3, sont des fonctions actives dans l'état courant du système. Chaque fonction à une donnée disponible sur ses ports d'entrées les fonctions peuvent propager les données vers les fonctions en aval du *data-flow*. Toutes les fonctions actives sont valides. Le *data-flow* est valide. Pour l'exemple de *data-flow* invalide, les fonctions F1 et F3 sont actives dans l'état courant du système, F2 ne l'est

pas. Il manque donc à F3 la donnée en provenance de F2 pour pouvoir propager les données. F3 est considérée invalide car étant active elle ne peut pas propager les données du *data-flow* : le *data-flow* est alors considéré invalide.

Plus formellement, quelque soit une fonction active f du *data-flow* D et quelque soit e_{in} un échange fonctionnel entrant dans une fonction active f. Le *data-flow* est valide ($D = 1$), si et seulement si la propriété "tous les e_{in} sont disponibles" est vérifié ($e_{in} \in e_{disponible}$).

$$\forall f \in D, \quad \forall e_{in} \in f \quad | D = 1 \models e_{in} \in e_{disponible}$$

La figure 6.11 présente les interactions entre les machines d'états et modes et le *data-flow*. Chaque état ou mode active un ensemble de fonctions du *data-flow*. En retour les échanges fonctionnelles disponibles peuvent être déclencheurs de transitions de la machine d'états et modes.

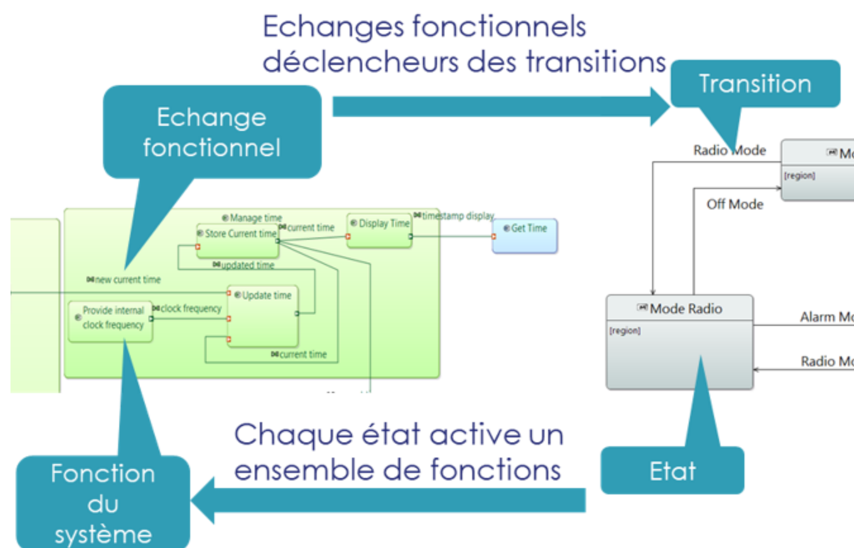


FIGURE 6.11 – interdépendance des machines d'états et modes avec le *data-flow*

Les interactions entre les machines d'états et modes et les *data-flow* sont valides si pour chaque combinaisons d'états et modes des machines d'états et modes concurrentes, le data-flow est valide et chaque combinaison d'états et modes possède au moins une transition réalisable (absence de dead lock).

Chaque scénario décrit le comportement dans un contexte particulier d'exécution du système. **Le scénario est valide si l'enchaînement de l'activation des fonctions**

est réalisable au regard de l'exécution des machines d'états et modes concurrentes.

6.2.2 Sémantique des machines d'états et modes

La sémantique des machines d'états et modes de Capella est proche de la sémantique des machines à états de UML. Les machines d'états et modes sont composées de régions, d'états, de pseudo-états et de transitions. Une région peut contenir des états qui eux mêmes contiennent des régions. Un état peut contenir plusieurs régions orthogonales et ainsi produire une exécution concurrente des sous-états contenues dans ces régions. Chaque mode ou état active un ensemble de fonctions appartenant au *dataflow*. Ces fonctions sont considérées actives lors de l'exécution du *dataflow* (resp inactives). Les pseudo-états sont des états transitoires impliqués dans une transition entre états. Les transitions permettent de passer d'un état à un autre par l'occurrence d'un événement. Les événements sont déclenchés soit par une fonction du *data-flow* soit par un échange fonctionnel. Nous ne considérons dans cette implémentation que les événements déclenchés par un échange fonctionnel.

Pour notre application de simulation nous considérons les machines d'états et modes comme des transducteurs finis [105] qui prennent en entrée les FE (Échanges Fonctionnels) disponibles du *data-flow* et produisent en sortie une information d'activation des fonctions du *data-flow*. Nous définissons la machine à états finis comme suit :
Soit $\mathcal{M}$ une machine à états fini, $\mathcal{M} = (S, s_0, S_F, \Sigma, \Gamma, \delta, \lambda)$ où :

- S est un ensemble fini d'états ;
- $s_0 \in S$ est un état initial ;
- $S_F \in S$ est un ensemble d'états terminaux ;
- Σ est un ensemble fini, alphabet d'entrée (pour nous l'ensemble des échanges fonctionnels du *dataflow*) ;
- Γ est un ensemble fini, alphabet de sortie (pour nous l'ensemble des fonctions du *dataflow*) ;
- $\delta : S \times \Sigma \to S$ est la fonction de transition ;
- $\lambda : S \to \Gamma$ est la fonction de sortie ;

Nous notons $s_i \xrightarrow{\delta} s_j$ la transition entre l'état s_i et l'état s_j
Nous notons $\mathcal{M}_1 \parallel \mathcal{M}_2$ deux machines à états concurrentes $\mathcal{M}_1$ et $\mathcal{M}_2$ Enfin nous

désignons par le terme **situation** une combinaison d'états et modes appartenant à des machines d'états et modes concurrentes.

6.2.3 Sémantique des *data-flows*

Nous avons implémenté dans notre outil deux options de sémantique pour la vérification des *data-flows* :

1. Le *data-flow* a un comportement synchrone au sens de [106]. Cette sémantique est intéressante pour simuler des systèmes sans conservation de mémoire (carte électronique de traitement du signale, système électrique ou mécanique sans stockage d'énergie). A chaque situation du système (combinaison active des états et modes du système), les informations traversent le *data-flow* en fonction des fonctions actives. Par conséquent la validité du *data-flow* ne dépend que de la situation courante .
2. Le *data-flow* a un comportement asynchrone au sens de [107]. Une donnée produite par une fonction source reste disponible tant qu'elle n'a pas été consommée par la fonction cible. Dans ce cas la validité du *data-flow* est le résultat de la situation courante mais aussi des situations précédentes (une donnée peut être produite dans une situation précédente du système et consommée dans la situation courante).

6.2.4 Sémantique des scénarios

Les scénarios dans Capella permettent de spécifier le déroulement dynamique, sur un axe temporel, des échanges entre fonctions ou entre composants comportementaux. Les scénarios peuvent aussi préciser l'enchaînement de l'activation des fonctions ainsi que l'enchaînement de l'activation des modes et des états. Le déroulé des scénarios doit être cohérent avec les contextes induits par le déroulé de l'exécution des machines d'états et modes en relation avec le *data-flow*. La figure 6.12 représente un scénario « Exchange scenario » avec la mise en œuvre des principaux concepts du méta-modèle *« Scénario »* de Capella. La chronologie du scénario se lit de haut en bas, les événements sont ordonnés chronologiquement dans la liste *« sequenceInteraction »* d'un objet de type *« InteractionFragment »*. En fonction du type de

scénario (scénario fonctionnel ou d'échanges entre composants), le concept *« InstanceRole »* représente soit des fonctions, soit des composants. Le concept de « SequenceMessage » représente les échanges entre les *« InstanceRole »*. Sur chaque ligne de vie des *« InstanceRole »*, les événements sont rassemblés dans des objets de type abstrait *« TimeLapse »* implémentés par des classes concrètes *« StateFragment »* et *« Execution »* en fonction du contexte, ces concepts représentent soit des fonctions du *data-flow*, des états et modes ou bien des composants comportementaux.

FIGURE 6.12 – Exemple de scénario de type« Exchange Scenario »

Afin de vérifier la cohérence des scénarios, nous avons réalisé un découpage séquentiel des événements modélisés dans un scénario. Nous reprenons la liste *« sequenceInteraction »* pour réaliser le découpage en séquence, une série d'attributs *« Start »* d'objets de type *« InteractionFragment »* déclenche une nouvelle séquence ; une série d'attributs *« Finish »* d'objets de type *« InteractionFragment »*, ou un changement de contexte (changement d'état ou de mode) la termine. La figure 6.13 montre le découpage en séquence d'un scénario en fonction d'une liste *« sequenceInteraction »*

A partir de ce découpage il est possible de vérifier :

- si l'exécution d'une fonction est réalisable dans le contexte décrit par le scénario,
- si l'enchaînement des contextes décrits par le scénario est réalisable,
- si l'enchaînement de l'exécution des fonctions est cohérent avec l'exécution du couple machines d'états et modes et data-flow.

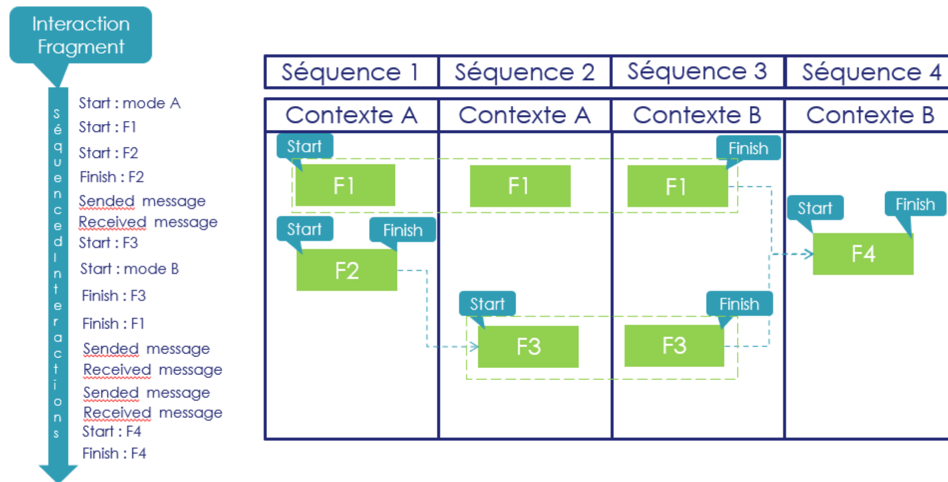


FIGURE 6.13 – Découpage séquenciel d'un sénario suivant la liste « *sequenceInteraction* »

6.2.5 Implémentation de l'outil de simulation

implémentation des *data-flows*

Nous utilisons pour notre application une sémantique de réseaux de Petri pour simuler le *data-flow*. Dans cette sémantique :

- Les transitions représentent les fonctions du *data-flow*. Elles sont activées par les états des machines à états.
- Les places représentent les ports entrants des fonctions du *data-flow*.
- Les arcs entre fonctions et ports entrants représentent les échanges fonctionnels. Ils peuvent être *trigger* d'une transition entre états des machines à états lorsqu'ils propagent une données.
- Les jetons représentent la disponibilité de données sur les ports entrants des fonctions.

Lorsque toutes les places en amonts d'une transition possèdent un jeton alors cette transition propage les jetons sur la suite du réseau de Petri.

Implémentation des machines d'états et modes pour l'option *data-flow* synchrone :

Pour cette option nous considérons une implémentation synchrone du *data-flow*. Le *data-flow* est réinitialisé lors de chaque changement de situation du système, par conséquent la validité du *data-flow* dans une situation ne dépend que de cette situa-

tion. Nous construisons le graphe des situations du système à partir des machines d'états et modes concurrentes. Chaque nœud correspond à une situation du système, chaque arc correspond à une transition entre deux situations. (cf figure 6.14)

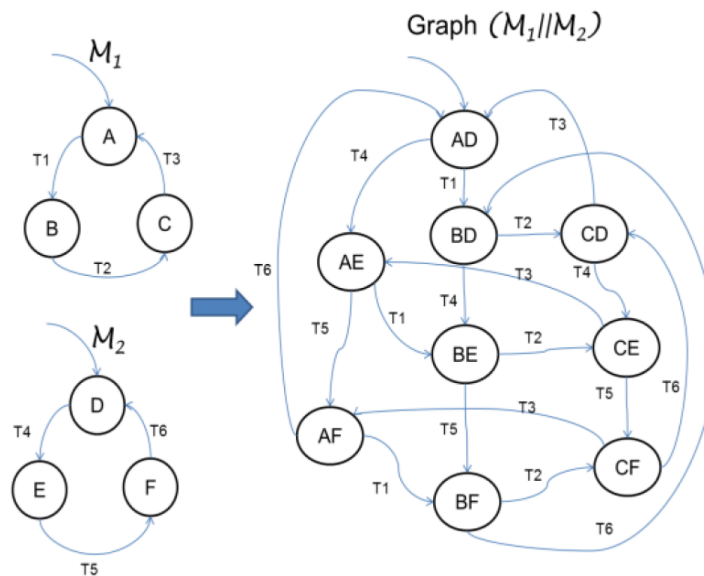


FIGURE 6.14 – Génération du graphe de situation à partir de deux machines à états concurrentes

Pour estimer la validité du modèle, nous parcourons le graphe de situations à l'aide d'un algorithme de parcours en largeur (Breadth-First Search Algoritm). Pour chaque nœud du graphe, l'outil initialise et exécute le data-flow suivant les fonctions activée par la situation. En retour, le data-flow retourne un ensemble de transitions disponibles. La figure 6.15 présente un exemple d'exécution du graphe de la figure 6.14. L'outil est capable dans ce cas de retourner :

- Les situations pour lesquels le data-flow est valide (en vert).
- Les situations pour lesquels le data-flow est invalide (en rouge).
- Les situations pour lesquels le système est bloqué (dead lock). Pour la situation BD le data-flow est valide mais aucune des transitions sortantes ($BD \xrightarrow{T2} CD$ et $BD \xrightarrow{T4} BE$) n'est disponible. Le système est alors bloqué dans la situation BD.
- Les situations qui ne sont pas atteignables (en blanc). Les transitions entrantes de la situation BE ne sont disponibles dans aucune des situations amonts.

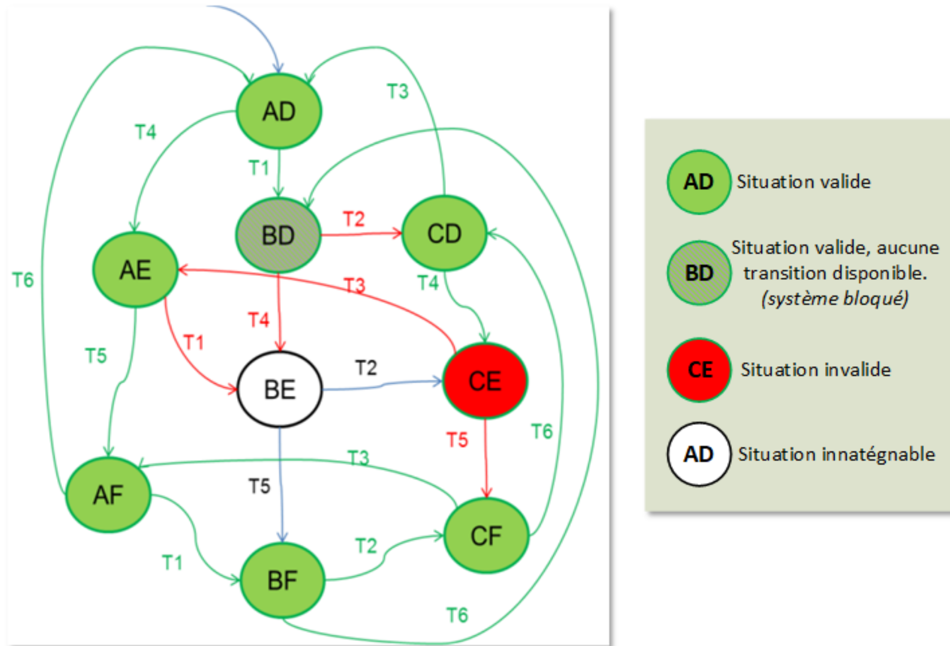


FIGURE 6.15 – État du graphe de situation à la fin de l'exécution

Implémentation des machines d'états et modes pour l'option *data-flow* asynchrone

Dans cette configuration de *data-flow* asynchrone, l'état du data-flow est conservé lors des transitions entre situations. Une donnée produite par une fonction source reste disponible tant qu'elle n'a pas été consommée par la fonction cible. Dans ce cas la validité du *data-flow* résulte de la situation courante du système mais aussi des situations précédentes.

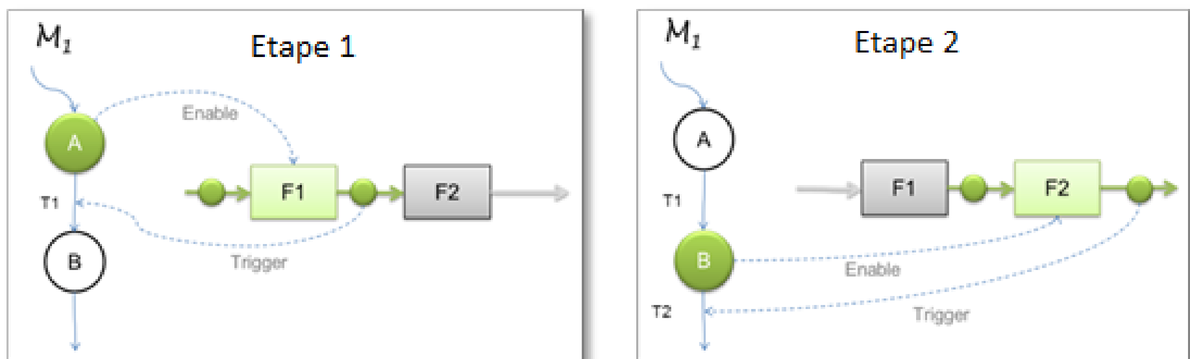


FIGURE 6.16 – Exemple d'exécution asynchrone

La figure 6.16 présente un exemple d'exécution asynchrone. Lors de l'étape 1, la fonction F1 est activée par la situation A. Elle produit une donnée qui n'est pas consommée par F2, car cette dernière n'est pas active. L'échange fonctionnel $F1 \rightarrow F2$ devient disponible et déclenche la transition $A \xrightarrow{T1} B$. Dans l'étape 2, la fonction F1 n'est plus active, mais la donnée produite lors de la première étape reste disponible. Ainsi F2, active dans B, peut produire une donnée. Construire un graphe incluant toutes les combinaisons de configuration entre *data-flow* et situation nous conduit à une explosion combinatoire du nombre de configuration possible, avec ensuite une difficulté pour l'ingénieur à analyser les résultats de la simulation. Ainsi pour cette option, l'outil exécute les machines d'état et modes et pour chaque situation l'outil de simulation évalue la validité du *data-flow*, qui en retour retourne un ensemble de transitions disponibles. plusieurs transitions peuvent être disponibles en même temps. L'outil choisit une transition en fonction d'une stratégie définie par l'utilisateur, trois stratégies sont disponibles :

1. **Roud Robin :** lors de la première visite d'une situation, l'outil de simulation active la première transition dans la liste des transitions disponibles. Si le système revient dans cette même situation, la seconde transition est activée. Lorsque toutes les transitions ont été activées, nous revenons à la première.
2. **Full Random :** une transition est choisie de manière aléatoire parmi les transitions disponibles
3. **Partial Random :** lors de la première visite d'une situation, l'outil choisit une transition de manière aléatoire parmi les transitions disponibles. Lors des visites suivantes les transitions déjà activées sont retirées du tirage. Lorsque toutes les transitions ont été explorées l'outil revient au cas de la première visite.

L'exécution s'arrête après avoir effectué un nombre de transitions prédéfini par l'utilisateur. En fonction de la topologie des machines de modes et états concurrentes certaines situations peuvent être visitées plusieurs fois alors que d'autre ne le seront jamais. L'option data-flow asynchrone permet de retourner :

- les situations visitées conduisant à un data-flow valide,
- les situations visitées conduisant à un data-flow invalide,
- les deadlocks.

Si une situation n'est jamais visitée, l'outil de simulation ne peut pas conclure : soit la situation n'a pas été visitée parce qu'elle n'est structurellement jamais atteignable ou soit parce que les conditions d'exécution font que la situation n'a pas été atteinte.

Comme classiquement avec ces méthodes, on ne peut pas garantir une vérification exhaustive du modèle, plus le système est complexe plus la vérification est partielle. Dans le cas d'un modèle très large, il peut être alors intéressant de réduire la sélection à un nombre limité de composants du système.

Rapport de simulation

A l'issue de la simulation l'outil de simulation produit un rapport au format HTML. Ce rapport décrit :

- les cycles trouvés dans le *data-flow*,
- la liste des incohérences, s'il en existe,
- la liste de chaque étape de la simulation, avec la situation visité, la liste des états actifs dans cette situation, les transitions sortantes de la situation, ainsi que les transitions actives.

La figure 6.17 présente un exemple simple de rapport produit par l'outil de simulation.

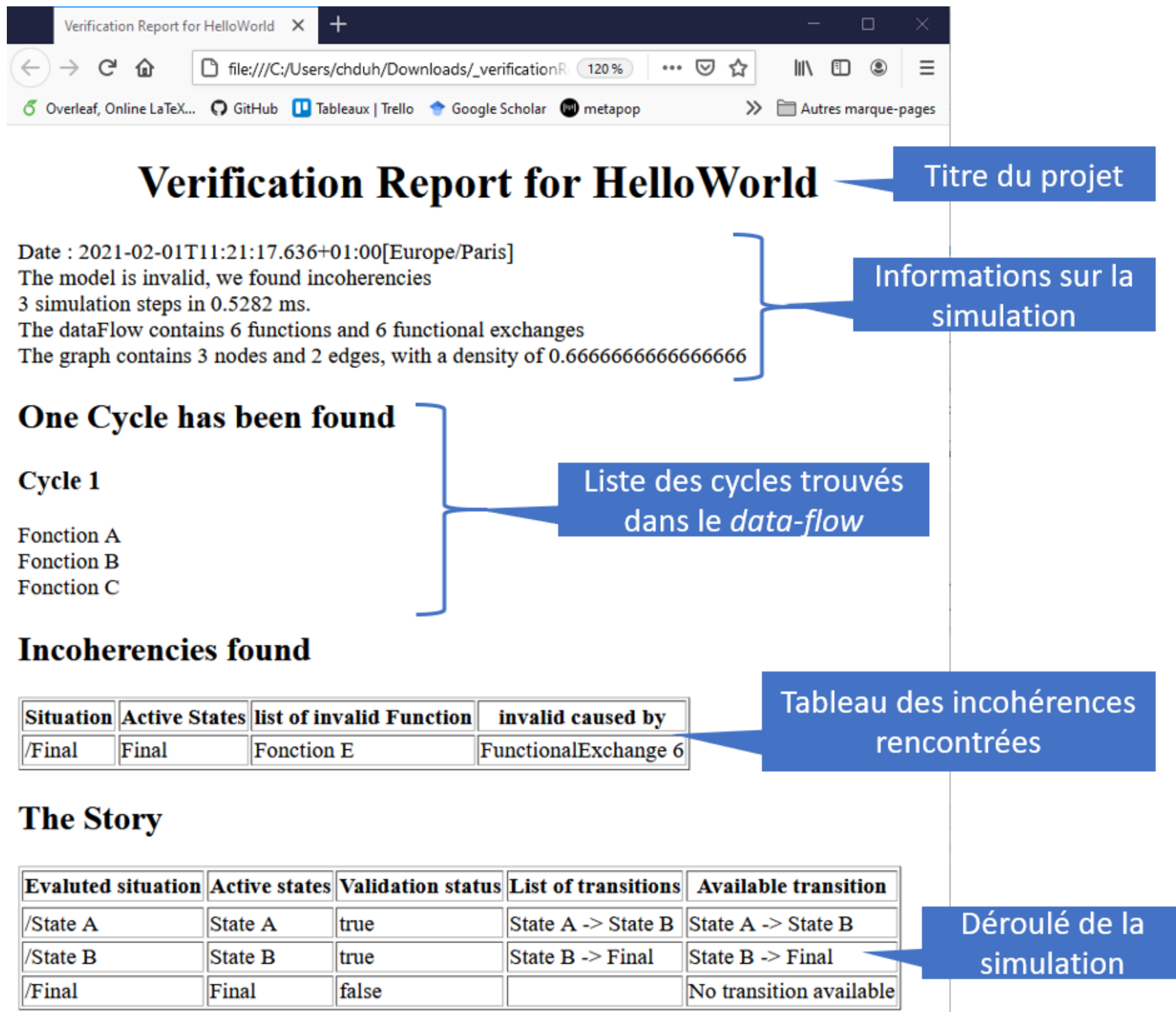


FIGURE 6.17 – Exemple de rapport de simulation

6.3 Transformation de modèles Capella, par application de la méthode ModelRun

Notre objectif est de vérifier la cohérence entre les diagrammes Capella de *data-flows*, de machines d'états et modes et de scénarios pour la partie architecture physique de la définition du système. Nous utilisons la méthode ModelRun pour transformer la sélection du modèle Capella en modèle exécutable par notre plugin de simulation. Dans la pratique une transformation de modèles peut implémenter plusieurs étapes de la méthode ModelRun. Dans notre cas, nous avons implémenté 3 transformations (cf figure 6.18). La première transformation implémente les étapes de sélection et de réorganisation. Elle cible un méta-modèle transitoire afin de stocker les informations du modèle réorganisé. La seconde transformation implémente l'étape d'alignement et cible le méta-modèle cible. La troisième transformation implémente les étapes d'enrichissement et d'adaptation. Cette transformation est endogène au méta-modèle cible. Dans la pratique le méta-modèle cible est prédéfini, il n'y donc pas d'ajout de classe a considérer. Par contre il faut ajouter et initialiser les objets manquants et adapter le modèle pour qu'il soit exécutable.

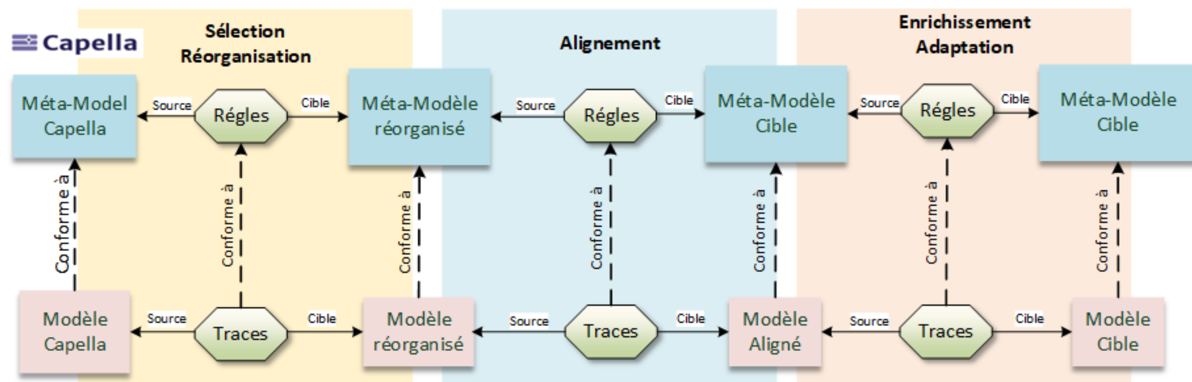


FIGURE 6.18 – étapes de transformation ModelRun

Pour chaque transformation, un modèle de traces fortement typé est généré pour tracer la transformation des concepts en fonction des règles de transformation. Cette chaîne de traçabilité crée un lien non ambiguë entre les objets du modèle Capella et les objets du modèle de simulation.

6.3.1 Étape de sélection

Le méta-modèle Capella est composé de 428 méta-classes réparties dans 20 méta-modèles. La première étape de la méthode consiste à sélectionner les concepts impliqués dans le data-flow, les machines de modes et états et les scénarios de l'architecture physique. Nous conservons aussi la classe « Projet » qui nous permet de garder les informations générales du projet tel que le nom du projet. Cette classe constitue la classe racine de notre sélection.

Sélection du data-flow

Pour pouvoir simuler le data-flow, nous avons besoins des informations contenues dans les classes *« PhysicalFunction »* , *« InputPort »*, *« OutputPort »* et *« FunctionalExchange »*. Conformément à notre méthode nous conservons toutes les classes incluses dans le chemin de contenance jusqu'à la classe racine. Nous conservons tous les attributs et relations entre les concepts sélectionnés ainsi que toutes les classes abstraites impliquées dans la généralisation des attributs et relations. La figure 6.19 présente les éléments du méta-modèle impliqués dans la sélection des concepts du *data-flow*. Pour des questions de lisibilité, nous n'avons pas représenté la classe abstraite *« NameElement »* impliquée dans la généralisation des attributs nom et id de toutes les classes concrètes.

Sélection des machines d'états et modes

Nous utilisons la même approche pour sélectionner les concepts des machines d'états et modes. Chaque machine d'états et modes est inclue dans un composant physique (classe *« PhyscalComposent »*) et interagit avec les fonctions physiques et les échanges fonctionnels de ce composant physique, à travers la relation généralisée par la classe abstraite *« AbstractEvent »*. La figure 6.20 présente la sélection des concepts liés aux machines d'états et modes.

Sélection des scénarios

Les scénarios représentent une séquence d'exécution du système ordonnée dans le temps. Les scénarios dans Capella manipulent des concepts de *« SequenteMessage », « InstanceRole », « d'Execution » « StateFragment »*. Ces concepts sont en re-

FIGURE 6.19 – Sélection des concepts du sous ensemble du méta-modèle Capella *« data-flow »*

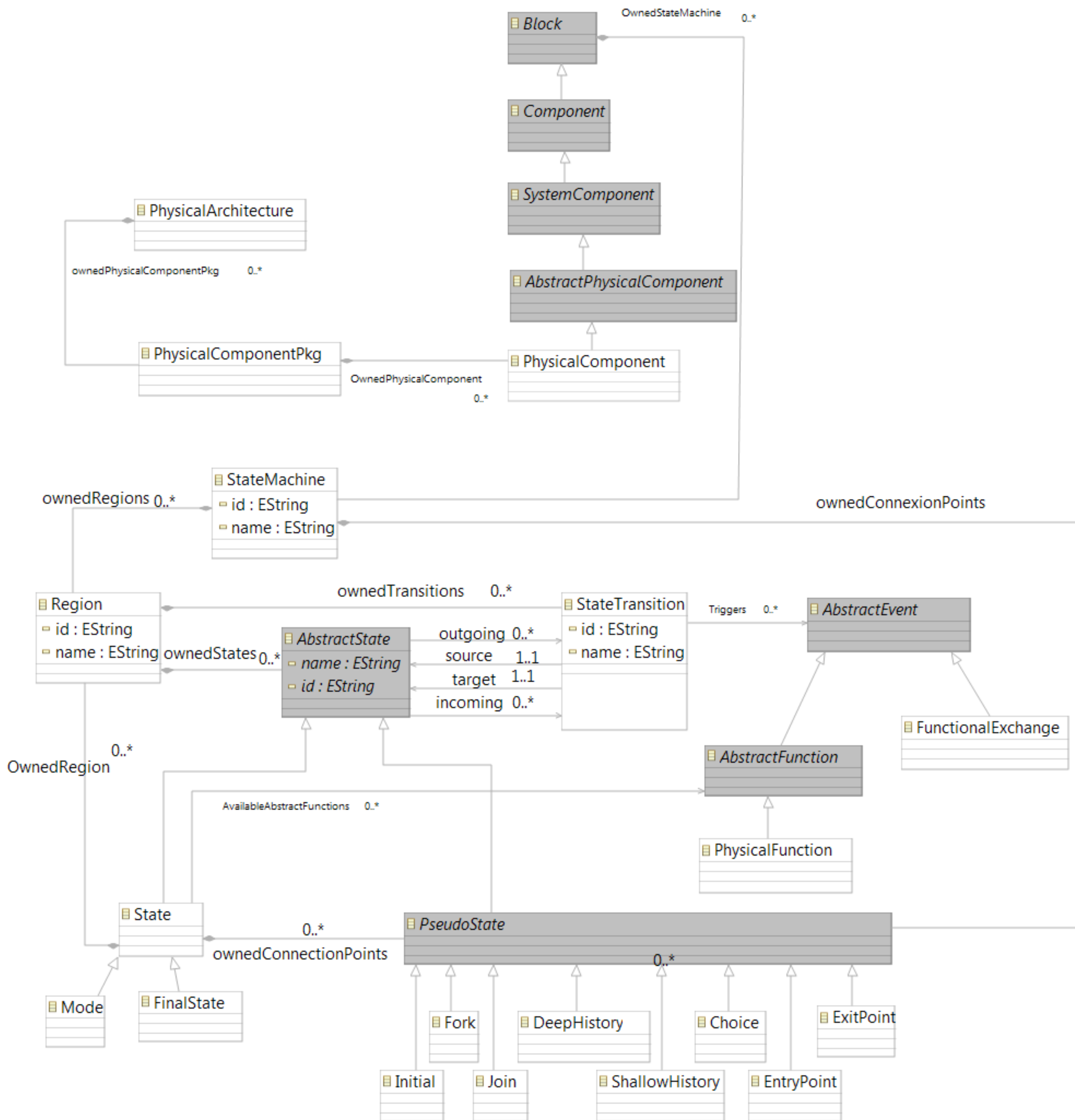


FIGURE 6.20 – Sélection des concepts du sous-ensemble Capella « machine d'états et modes »

lation avec les concepts de fonction, d'échanges fonctionnels et de modes/états qu'ils manipulent. La figue 6.21 présente la sélection des concepts liés aux scénarios ainsi que leurs relations avec les concepts du *data-flow* et des machines d'états et modes.

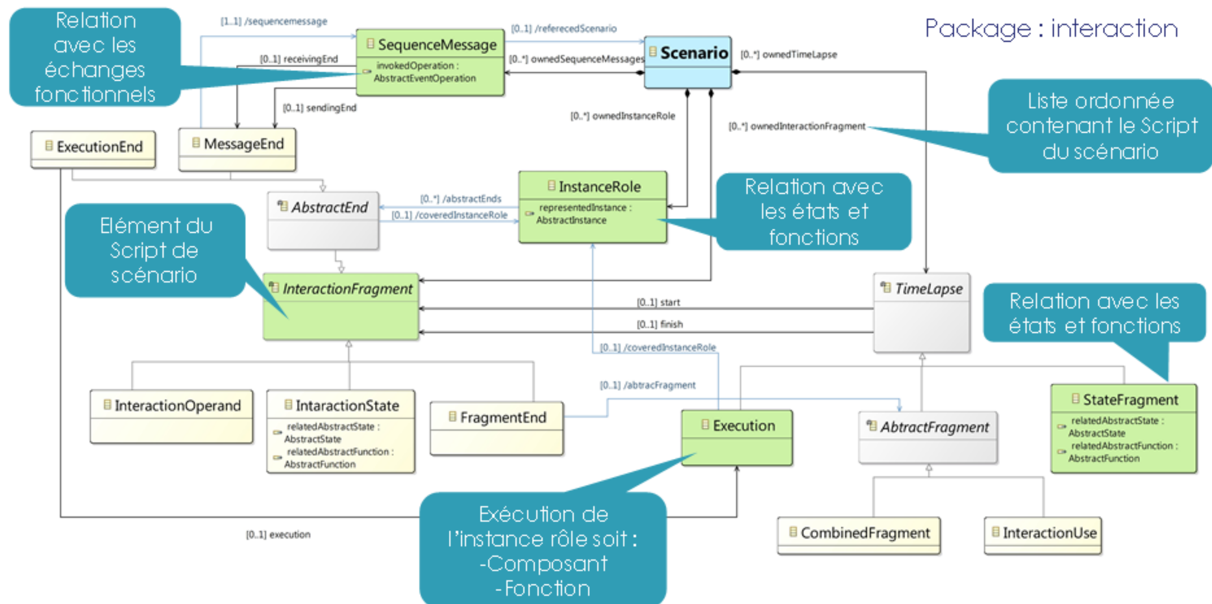


FIGURE 6.21 – Sélection des concepts du sous-ensemble Capella *« scenarios »*

6.3.2 Étape de réorganisation

L'étape de réorganisation prépare les concepts du modèle source avec pour objectif qu'ils soient directement alignés sur les concepts du méta-modèle cible. La réorganisation des concepts se fait donc en ayant à l'esprit le méta-modèle cible.

Réorganisation et alignement du *data-flow*

Le *data-flow* est simulé à l'aide d'un réseau de Petri. Le tableau 6.1 présente la correspondance des concepts impliqués dans la simulation du *data-flow* entre les méta-modèles source, réorganisé et cible.

Pour obtenir le résultat du tableau 6.1, les arbres de classes abstraites sont aplatis et supprimés quand c'est possible. Seul la classe abstraite *« AbstractEvent »* faisant le lien entre le data-flow, les machines d'états et modes ainsi qu'avec les scénarios est conservée. Le concept de *« InputPort »* est transformé in fine en *« Place »* dans

source	réorganisation	cible
PhysicalArchitecture	RefactoredDataFlow	PetriNet
AbstractEvent	RefatoredAbstractEvent	TargetAbstractEvent
FunctionalExchange	RefactoredFunctionalExchange	triggerEdge
PhysicalFunction	RefactoredFunction	Transition
InputPort	RefactoredInputPort	Place

TABLE 6.1 – Correspondance des concepts sources, réorganisés et cibles

le réseau de Petri et permet de stocker les données en attente de traitement en entrée d'une « *Transition* » image d'une « *PhysicalFunction* ». Les échanges fonctionnels(« FunctionalExchange ») peuvent être porteur d'informations permettant la transition entre états des machines d'états du système. L'information de déclenchement est conservée et contenue dans un arc « TriggerEdge » du réseau de Petri. La figure 6.22 présente l'extrait du méta-modèle réorganisé relatif au *data-flow*.

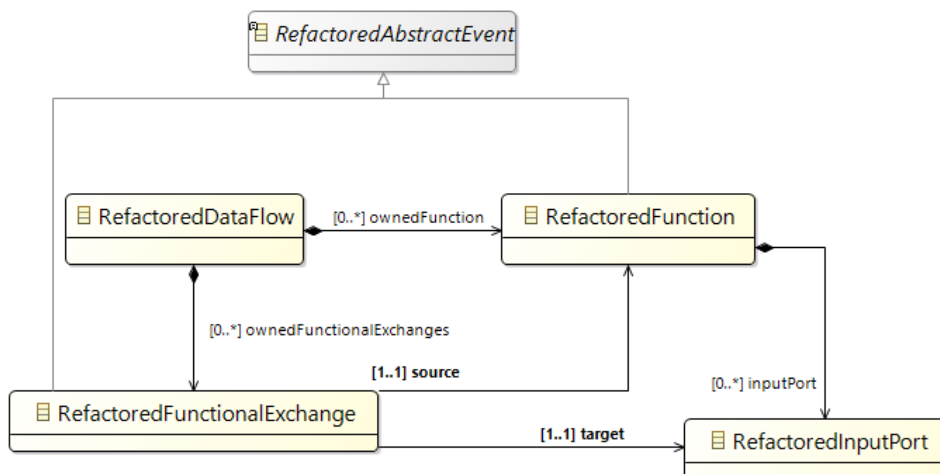


FIGURE 6.22 – Extrait du Méta-modèle réorganisé relatif au *data-flow*

Réorganisation des machines d'états et modes

La partie du méta-modèle consacrée aux machines d'états et modes est semblable aux méta-modèle des machines à états UML, c'est aussi celle qui compose le méta-modèle cible. Les opérations de réorganisation opèrent uniquement pour aplatir certaines abstractions comme par exemple le concept abstrait « *Block* » qui est supprimé et dont la relation « ownedStateMachine » est directement rattachée à la classe

concrète *« PhysicalComponent »*.

Réorganisation des scénarios

Le méta-modèle cible implique de découper les éléments du scénario en séquence d'éléments « exécutés » dans la même unité de temps (cf figure 6.13). Pour cela, nous utilisons les informations « start » et « finish » de chaque *« InteractionFragment »* (cf figure 6.21) pour créer le découpage en séquence. Ici, aucune information n'est crée mais les informations sont réarrangées pour permettre un alignement avec le domaine cible.

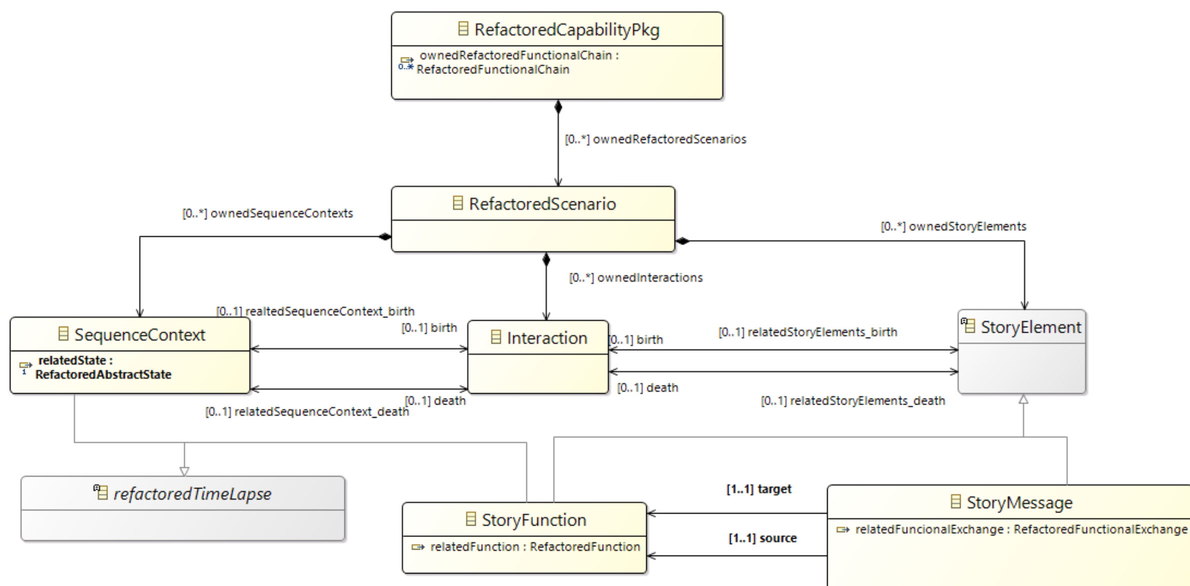


FIGURE 6.23 – Extrait du Méta-modèle réorganisé : scenario

La réorganisation des informations des scénarios dépend du type de scénario. Si le scénario est un scénario « Exchange Scenario » alors nous considérons les objets de type « StateFragment » pour intégrer le lien avec les fonctions physiques du data-flow. Si le scénario est de type *« Functional Scenario »*, nous considérons alors les objets de type *« Execution »*

source	réorganisation	cible
CapabilityPkg	RefactoredCapabilityPkg	TargetCapabylity
Scenario	RefatoredScenario	TargetScenario
InteractionFragment	StoryElement	TargetStoryElement
InteractionFragment	Interaction	TargetInteraction
SequenceMessage	StoryMessage	TargetStoryMessage
si « StateFragment » en relation avec un état ou mode :		
StateFragment	SequenceContext	TargetSequenceContext
source « Exchange Scenario » :		
StateFragment	StoryFunction	TargetStoryTransition
source « Functional Scenario » :		
Execution	StoryFunction	TargetStoryTransition

TABLE 6.2 – Correspondance des concepts sources, réorganisés et cibles pour les scénarios

6.3.3 Étape d'alignement

L'étape d'alignement nous permet de traduire la sélection des concepts sources réorganisés en concepts du domaine de simulation. C'est une transformation directe entre les concepts du modèle réorganisé et ceux du modèle cible. A chaque concept réorganisé correspond un unique concept cible. La figure 6.24 présente l'alignement des concepts du *data-flow* réorganisés vers les concepts du réseau de Petri au travers du graphe de traçabilité. Les autres éléments du méta-modèle réorganisé (concepts des machines d'états et modes, scénarios) suivent le même principe.

6.3.4 Étapes d'enrichissement et d'adaptation

Pour que le modèle cible obtenu puisse être exécuté, il faut l'enrichir et l'adapter. Certains concepts directement liés au domaine de simulation n'ont pas d'antécédent dans le domaine source. Ces concepts apportent des informations nouvelles uniquement liés au domaine de simulation. Il faut donc les ajouter et les adapter au modèle provenant du domaine source.

Enrichissement et adaptation du réseau de Petri

Le méta-modèle du réseau de Petri comporte trois concepts n'ayant pas d'antécédent dans le méta-modèle source du data-flow (en vert sur la figure 6.25) :

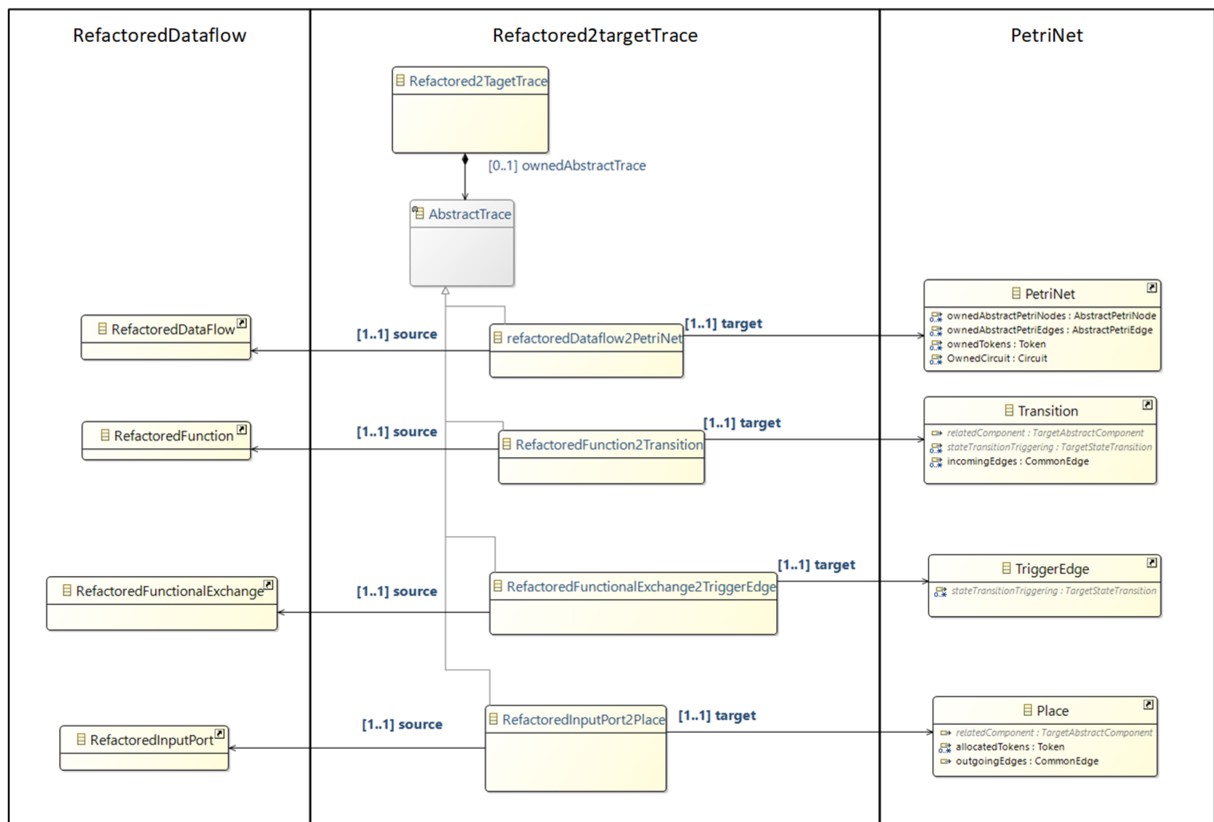


FIGURE 6.24 – Extrait de l'alignement centré sur l'alignement du data-flow réorganisé : data-flow vers PetriNet

- ***« Circuit »*** : Un algorithme de parcours en profondeur permet de répertorier les circuits à l'intérieur du réseau de Petri. Chaque circuit est initialisé avec un jeton.
- ***« Jeton »*** : Les jetons représentent la disponibilité d'une donnée dans le réseau de Petri. Le réseau de pétri est initialisé en ajoutant un jeton sur chaque *« Place »* en entrée du réseau de Petri.
- ***« CommonEdge »*** : les « CommonEdges » représentent la liaison entre les *« Places »* (images des ports d'entrée de fonctions) et les *« Transitions »* (image des fonctions du data-flow).

En complément de ces ajouts et adaptations des objets de type « Place » sont crées pour chaque entrée et chaque sortie du réseau de Petri.

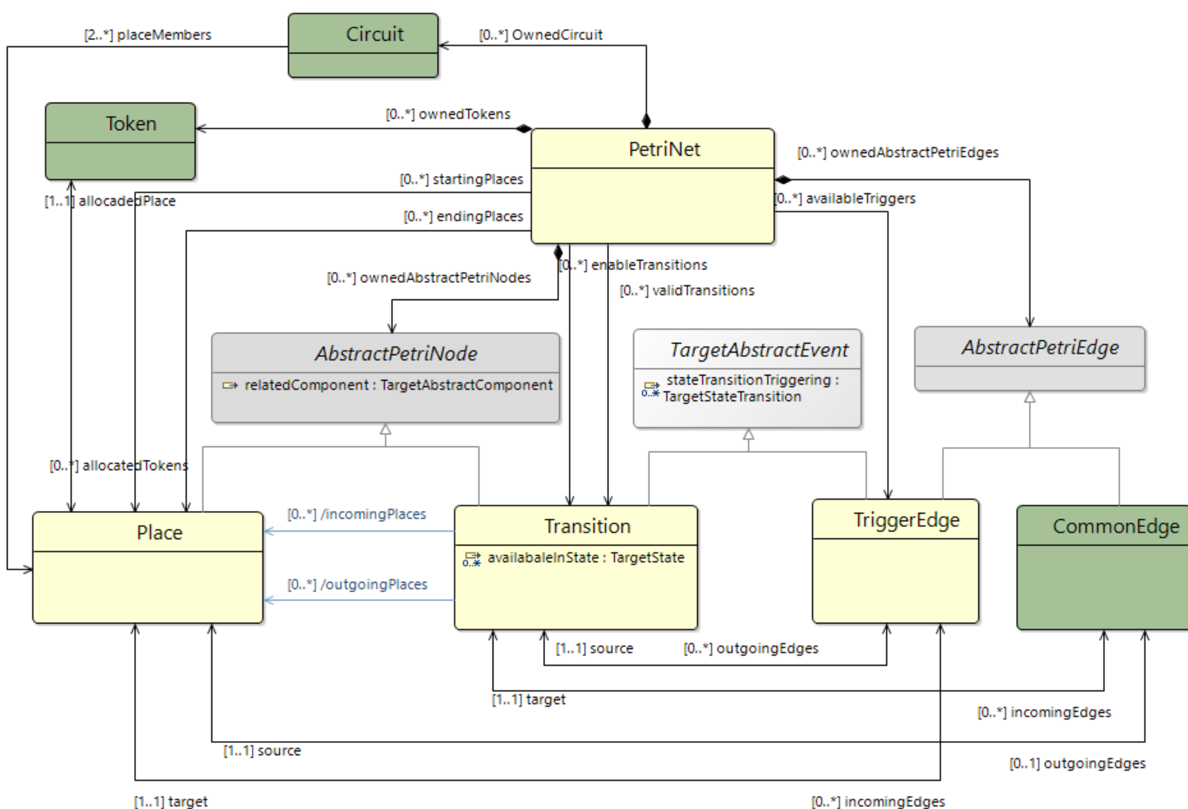


FIGURE 6.25 – Méta-modèle du réseau de Petri pour la simulation du data-flow

Enrichissement et adaptation des machines d'états et modes

Les machines d'états et modes du domaine de simulation sont l'image du méta-modèle de Capella (avec une simplification des classes abstraites). En complément

sont ajoutés un ensemble de concepts permettant de générer un graphe de situation, tel que présenté au chapitre 6.2.5 :

- ***« SituationSet »*** est le concept racine du graphe de situation.
- ***« initialSituationNode »*** regroupe les états et modes initiaux des différentes machines concurrentes.
- ***« SituationNode »*** sont les nœuds du graphe, ils sont constitués d'un ensemble d'états et modes appartenant à des régions orthogonales et/ou des machines d'états et modes concurrentes.
- ***« SituationEdge »*** sont les arcs du graphe ils sont constitués des transitions possibles entre les différentes situations.
- La relation *« theActiveSituation »* est unique. Lors de l'exécution, elle pointe vers la situation active.

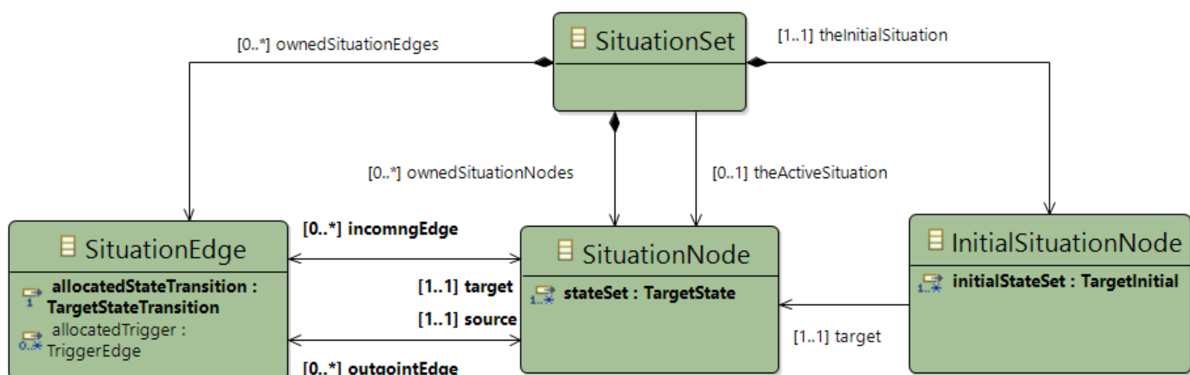


FIGURE 6.26 – Méta-modèle des situations

Enrichissement et adaptation des scénarios.

Les enrichissements et les adaptations de la partie scénario du méta-modèle cible permettent de découper le déroulé des scénarios en séquences et en contextes (cf paragraphe 6.2.4 sur la sémantique des scénarios). La figure 6.27 présente le méta-modèle pour la simulation des scénarios. Les concepts ajoutés pour l'enrichissement sont représentés en vert :

- ***« TargetSequence »*** regroupe les fonctions du data-flow actives dans une même unité de temps. (sans notion de cette unité de temps)
- ***« TargetContext »*** indique le contexte d'exécution d'une séquence. quels sont les états et modes actifs pour la séquence considérée.

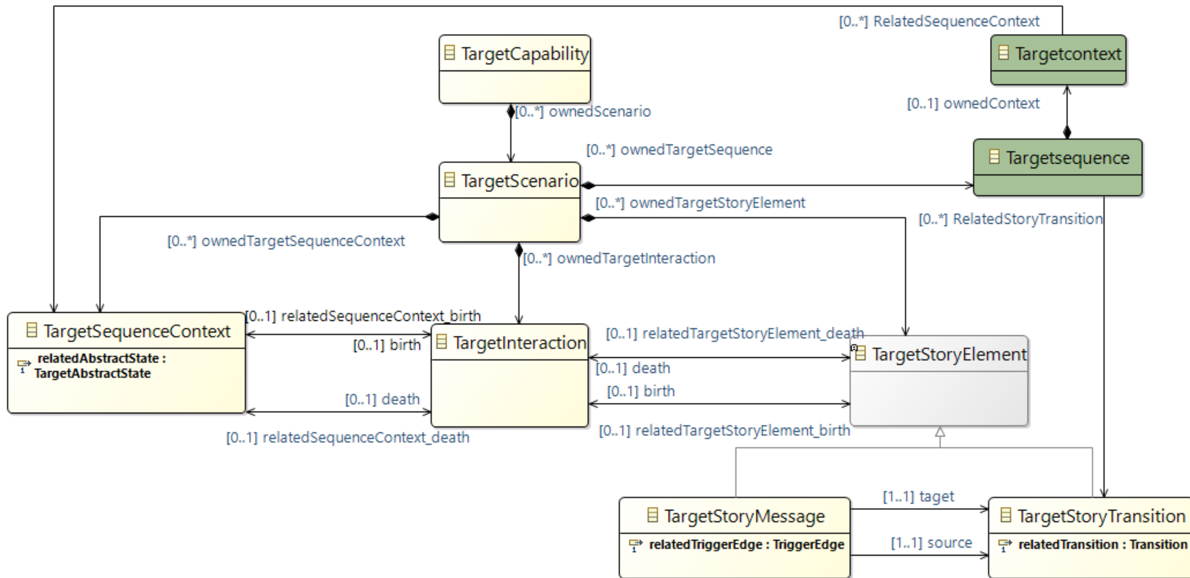


FIGURE 6.27 – Méta-modèle de simulation des scénarios

6.4 Résultats de simulation

6.4.1 Vérification de la cohérence entre le *data-flow* et les machines d'états et modes.

La machine de modes que nous avons présenté en figure 6.9 du chapitre 6.1.6 définit les modes de fonctionnement du drone. Ainsi le drone fonctionne suivant deux modes principaux ; le mode manuel et le mode automatique. La figure 6.28 présente une vue du *data-flow* relative au processus de prise et d'enregistrement de photos et de vidéos. La fonction « *Select camera control* » synchronise ce processus, elle reçoit ses instructions en fonctionnement manuel par la fonction « *Acquire manual payload control* » et en fonctionnement automatique par la fonction « *Run automatic control* ». Ces deux fonctions sont actives, respectivement dans les modes « *Manual mode* » et « *autopilot mode* » de la machine de modes. Lors de la simulation, nous sommes confrontés à une incohérence entre le *data-flow* et la machine de modes. En effet, la sémantique de *data-flow* synchrone, que nous avons choisi pour simuler le *data-flow* implique qu'une fonction active doit avoir toutes les données disponibles sur ses entrées pour être valide et propager la disponibilité des données (cf chapitre 6.2.3). Or

les deux fonctions « *Aquire manual payload control* » et « *Run automatic control* » ne sont jamais actives dans le même temps (l'une l'est en mode manuel, l'autre en mode automatique). La fonction « *Select camera control* » est donc toujours invalide.

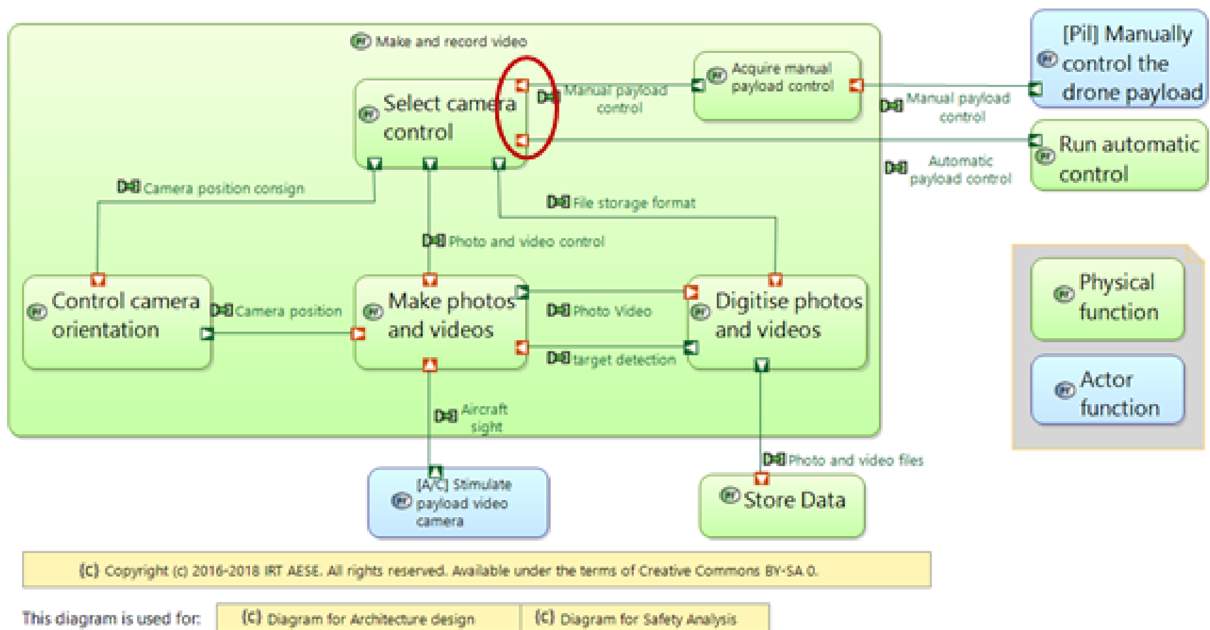


FIGURE 6.28 – Exemple de data-flow incohérent avec la machine de modes de fonctionnement du drone

La figure 6.29 montre un extrait du rapport produit lors de cette simulation. Le rapport présente la liste des fonctions invalides pour chaque situation évaluée par la simulation. En mode manuel les fonctions sont invalides à cause de l'indisponibilité de l'échange fonctionnel « *Automatic payload control* ». En mode automatique les fonctions sont invalides à cause de l'indisponibilité de l'échange fonctionnel « *Manual payload control* » (le rapport complet est présenté en annexe 1).

Pour corriger cette incohérence, nous proposons de relier les échanges fonctionnels « *Manual payload control* » et « *Automatic payload control* » au même port d'entrée de « *Select camera control* » (cercle vert figure 6.30). Ainsi que ce soit dans un mode manuel ou automatique, la fonction « *Select camera control* » a toujours une donnée disponible sur son entrée, elle est donc valide et peut propager des données en aval du *data-flow*.

Active States	list of invalid Function	invalid caused by
Manual mode (MP) /Stand by Camera/Stand by Record (in : Video and photo process)	Select camera control Control camera orientation Make photos and videos Digitise photos and videos Store Data	Automatic payload control
Stand-by (in : Autopilot mode (AP)) /Stand by Camera/Stand by Record (in : Video and photo process)	Select camera control Control camera orientation Make photos and videos Digitise photos and videos Store Data	Manual payload control

FIGURE 6.29 – Extrait du rapport produit lors de la simulation

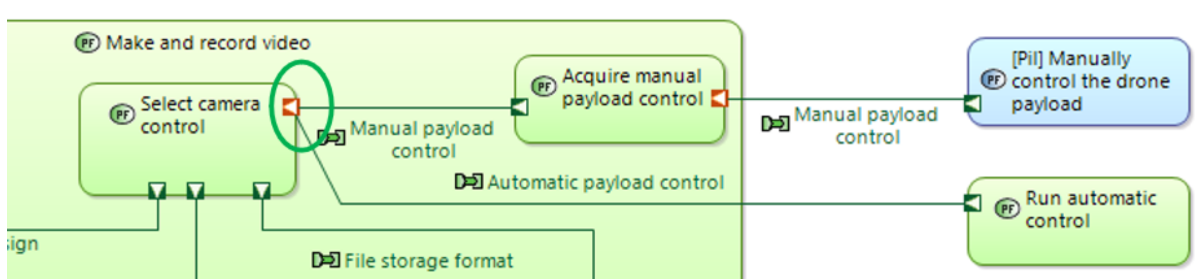


FIGURE 6.30 – correction de l'incohérence *data-flow* avec la machine de modes de fonctionnement du drone

6.4.2 Vérification de propriétés dynamiques dans l'enchaînement des situations

Le fonctionnement du processus de prise de vidéos et de photos est précisé par deux machines à états concurrentes (figure 6.31). Ces machines à états fonctionnent de manière concurrentes à la machine de modes (automatique/manuel), cette dernière spécifiant le fonctionnement des modes manuel et automatique de l'ensemble du système. La première machine à états permet d'orienter la caméra vers une cible puis de la suivre. La seconde machine à états déclenche l'enregistrement et la numérisation des photos et vidéo après détection de la cible, puis déclenche la sauvegarde des fichiers de ces photos et vidéos en fin de mission. Pour une situation donnée, L'ensemble des fonctions actives du *data-flow* correspond à l'union des fonctions activées par chaque état. Ainsi, si nous considérons la situation composée des états : [*« Oriente the camera »* ; *« Record Photo and vidéo »*]. L'ensemble des fonctions actives est composé des fonctions : *« Select camera control », « Control camera orientation », « Make photos and videos »* et *« Digitise photos and videos »*.

Dans la configuration de la figure (figure 6.31), l'outil de simulation ne rencontre pas de *dead lock*. La machine de mode « automatique, manuelle » peut évoluer sans blocage. Cependant, l'outil de simulation retourne un certain nombre de situations ne pouvant pas être atteintes parmi lesquelles :

- *[« Points the camera »* ; *« Record Photo and video »]*,
- *[« Points the camera » ; « Store photo and video »]*,
- *[« Position ready » ; « Record Photo and video »]*,
- *[« Position ready » ; « Store photo and video »]*,

Les machines à états pour la prise de vues vidéos et photographiques restent bloquées dans la situation [*« Poins the camera » ; « Stand by Record »*] pour que le système évolue il faut que le *trigger* porté par l'échange fonctionnel *« target detection »* soit disponible. Comme le montre la figure 6.32 l'échange fonctionnel *« target detection »* relie les fonctions *« Digitise photos and video »* et *« Make photos and videos »* pour que l'échange fonctionnel *« target detection »* soit disponible, il faut que la fonction *« Digitise photos and video »* soit active et valide. Cependant, pour que cette fonction soit active la machine à états « Record video » doit être dans l'état « Record photo and video » ce qui est impossible puisque pour passer dans cet état *« target detection »* doit être disponible. Nous nous trouvons dans une situation ne pouvant plus évoluer

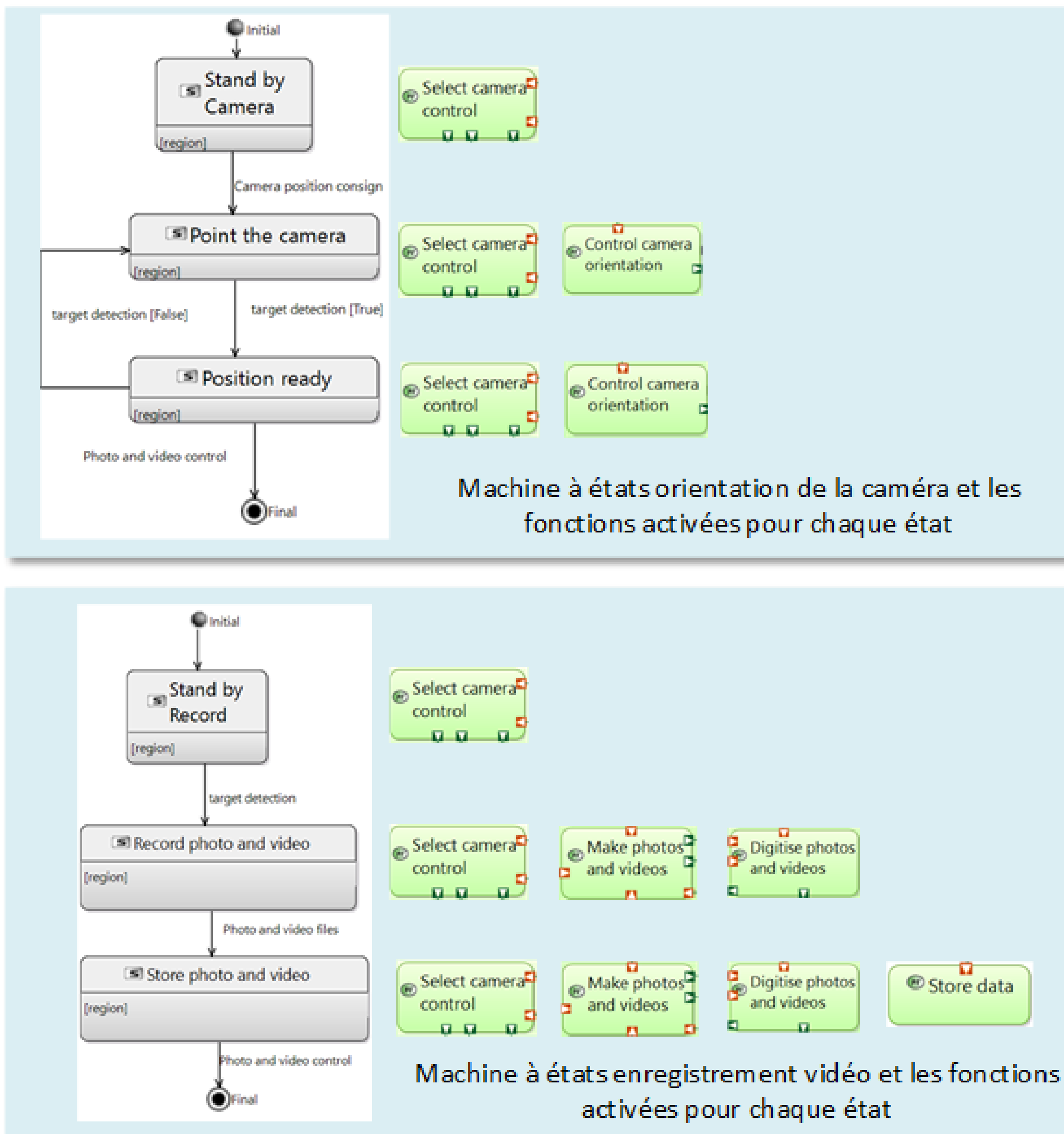


FIGURE 6.31 – Machines à états concurrentes pour la prise de vue vidéos et photographiques

vers une situation permettant l'enregistrement et le stockage des photos et vidéos.

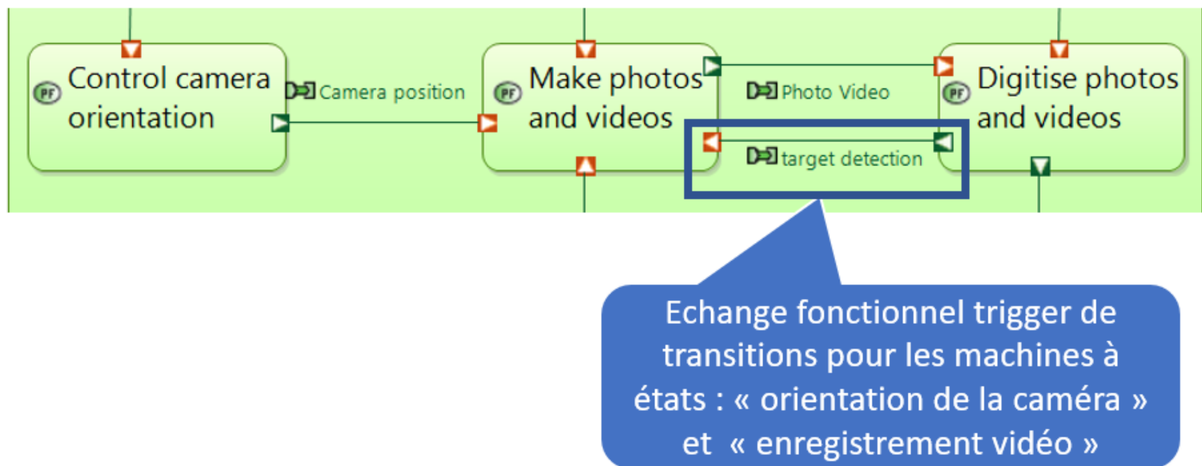


FIGURE 6.32 – Extrait du *data-flow* montrant l'échange fonctionnel *trigger* de transitions

La figure 6.33 montre un extrait du rapport de simulation. Les transitions permettant d'accéder aux états « *Position ready* » et « *Record Photo and video* » apparaissent dans la liste des transitions, mais sont absentes de la liste des transitions disponibles. Les états « *Position ready* » et « *Record Photo and video* » ne sont jamais actifs, ils n'apparaissent jamais dans le rapport (le rapport complet est présenté en annexe 2).

Active states	Validation status	List of transitions	Available transition
Manual mode (MP) /Point the camera/Stand by Record (in : Video and photo process)	true	Manual mode (MP) -> Autopilot mode (AP) Manual mode (MP) -> Speed consign mode Point the camera -> Position ready Stand by Record -> Record photo and video	Manual mode (MP) -> Autopilot mode (AP) Manual mode (MP) -> Speed consign mode

FIGURE 6.33 – Extrait du rapport de simulation

Pour résoudre ce problème, nous ajoutons les fonctions « *Make photo and video* » et « *Digitise photo and video* » à la liste des fonctions activées par l'état « *Points the camera* ».

Suite à la correction des deux incohérences précédentes l'outil de simulation produit un rapport confirmant que le modèle est valide (cf : figure 6.34)

6.4.3 Vérification de la faisabilité d'un scénario

Dans cette version de l'outil de simulation il est possible de vérifier la cohérence entre les fonctions du *data-flow* et les états et modes des machines d'états et modes

Verification Report for PythagoreUAS

Date : 2021-01-27T14:10:00.805+01:00[Europe/Paris]
The model is valid
64 simulation steps in 395.1522 ms.
The dataFlow contains 129 functions and 288 functional exchanges
The graph contains 64 nodes and 288 edges, with a density of 4.5

FIGURE 6.34 – Extrait du rapport de simulation valide

du système. L'outil de simulation retourne une erreur lorsqu'il est décrit l'activation d'une fonction dans une même unité de temps qu'une situation qui ne comporte pas l'activation de cette fonction. L'exemple de la figure 6.35 présente un scénario pour la prise de de vidéo. Le scénario indique que le processus se déroule en mode manuel et indique quand l'état *« Record photo and video »* devient actif. Cependant conformément au scénario et la sémantique que nous avons choisi pour l'interpréter, lorsque la fonction « Store Data » devient active, l'état *« Record photo and video »* est toujours actif. Or *« Store Data »* est inactif dans l'état « Record photo and video ». L'outil de simulation retourne alors une erreur. Dans cette version de l'outil, les résultats de vérification des scénarios ne sont pas intégrés au rapport HTML. La figure 6.36 montre une copie d'écran du résultat de la vérification.

Pour corriger l'erreur, il faut spécifier le changement d'état *« Store photo and video »* avant que la fonction *« Store Data »* ne soit active. La figure 6.37 montre la modification du scénario Capella pour que ce scénario soit conforme à la sémantique de l'outil de simulation.

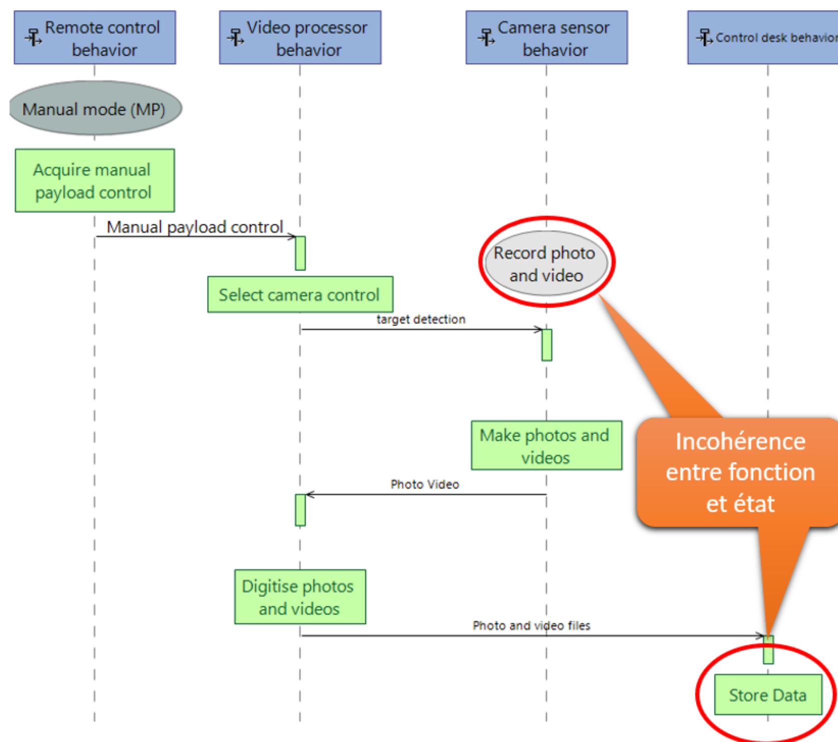


FIGURE 6.35 – Exemple de scénario incohérent

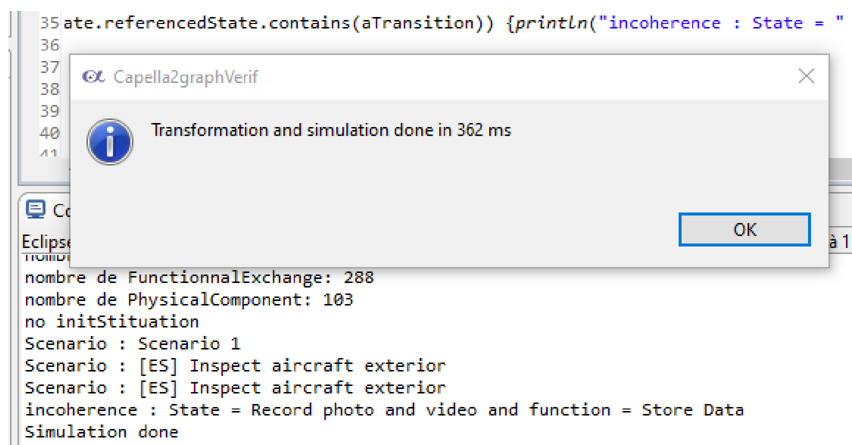


FIGURE 6.36 – Copie d'écran montrant l'incohérence dans le scénario

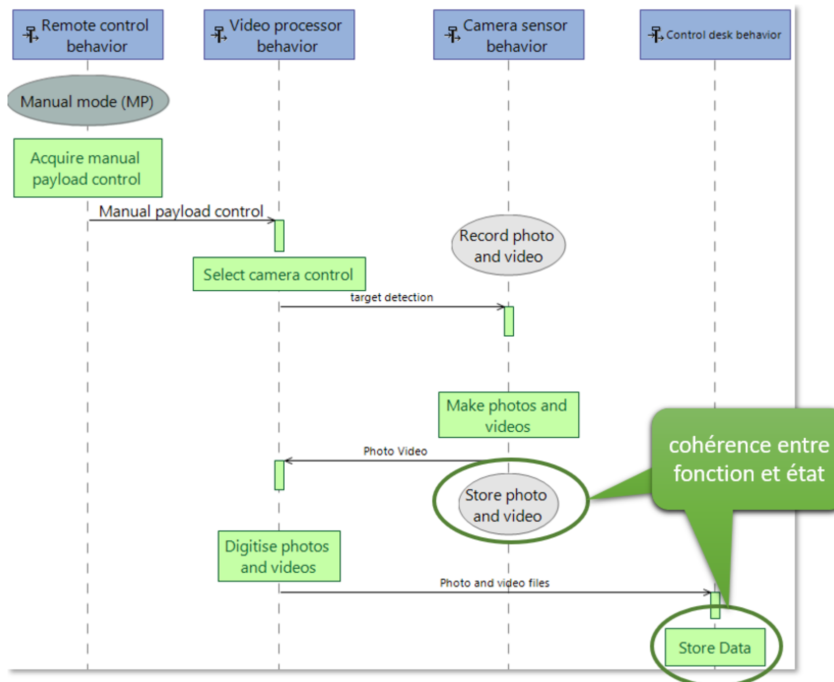


FIGURE 6.37 – Correction de l'incohérence entre état et fonction

6.4.4 Bilan des résultats de vérification

Notre outil de simulation permet de réaliser un premier niveau de vérification de l'aspect comportemental des modèles d'ingénierie Capella. Cette vérification est réalisée sans avoir d'indication précise sur le comportement de chaque fonction. En créant un lien direct entre les instances des concepts de l'outil de simulation et les instances des concepts Capella, la chaîne de traçabilité permet d'interpréter les résultats de simulation au regard des modèles Capella. Ces résultats de simulation sont fortement dépendants de la sémantique choisie pour interpréter le comportement des modèles Capella. Il revient donc en dernier lieu à l'ingénieur système d'interpréter ces résultats de simulation et de décider si les incohérences détectées par l'outil de simulation doivent faire l'objet d'une modification du modèle Capella.

Conclusion

Nous traitons dans ce mémoire, de la transformation de modèles d'ingénierie de systèmes complexes à des fins de simulation. Le besoin de simulation répond à la nécessitée de vérifier le plus tôt possible dans le processus d'ingénierie la cohérence de modèles composées de vues corrélées entre elle mais dont la sémantiques n'est pas totalement décrite. Ces modèles décrivent le fonctionnement du système, ils ne possèdent pas de sémantique opérationnelle permettant d'exécuter des comportements.

Bilan

Dans cette thèse nous proposons une méthode de transformation de modèles, appelé « ModelRun » permettant de transformer des modèles d'ingénierie système en un modèle conforme à un domaine de simulation ciblé. Nous formalisons notre méthode à l'aide de grammaires de graphes triples. Pour cela, nous définissons la notion de graphe typé attribué avec lien de contenances et arbres d'héritages *(AIC-graphe)*, ainsi que les morphisme de *AIC-graphe*. Nous établissons un lien entre les transformations de graphes et les transformations de modèles.

Notre méthode comporte cinq étapes, chaque étape joue un rôle déterminant dans la définition de la sémantique :

1. **La sélection** définit le sous-ensemble du modèle d'ingénierie pour lequel nous voulons vérifier des propriétés. Cette première transformation impacte la sémantique opérationnelle en limitant sa zone de définition.
2. **La réorganisation** prépare les concepts sélectionnés pour qu'ils puissent être alignés avec les concepts du domaine cible. Nous considérons, trois opérations de base :
   - **La fusion de concepts** : deux ou plusieurs concepts du modèle source sont réorganisés pour ne former qu'un seul concept.
   - **La fission de concepts** : les informations d'un concept du modèle source sont réparties entre plusieurs concepts du modèle cible.

- **La suppression de concept** : un concept du modèle source est supprimé dans le modèle cible. Les liens qui unissent ce concept aux autres sont alors réorganisés.

3. **L'alignement** traduit les concepts du DSML source sélectionné en concepts du DSML cible. Nous donnons ici une interprétation aux concepts sources pour pouvoir les simuler.
4. **L'enrichissement** ajoute les informations et concepts inexistants dans le domaine source, mais nécessaires au domaine cible. Cette transformation apporte de la sémantique en ajoutant des informations manquantes nécessaires à la simulation et à l'analyse.
5. **L'adaptation** corrige les concepts obtenus pour que le modèle soit conforme au langage défini par le méta-modèle du domaine cible.

A l'issue des cinq étapes de transformation, nous obtenons un modèle, conforme à un langage doté d'une sémantique opérationnelle, permettant la simulation d'une sélection d'un modèle source réorganisé et adapté auquel nous avons ajouté des informations nécessaires à la simulation.

La traçabilité est un élément central de ModelRun. Nous utilisons le graphe de correspondance défini dans la grammaire de graphes triples pour établir un lien de traçabilité fortement typé entre les concepts du modèle source et les concepts du modèle cible. Ce lien de traçabilité nous permet d'interpréter les résultats de simulation établis dans le domaine cible de simulation dans le domaine source de modélisation.

Nous appliquons notre démarche à un modèle d'ingénierie système de drone aérien. Nous présentons un outil de simulation permettant de vérifier des propriété de cohérence entre les diagrammes de data-flow, de scénarios et de machine d'états et modes, de modèles d'ingénierie système modélisé avec ARCADIA/Capella . L'application de ModelRun pour transformer le modèle Capella en modèle de simulation exécutable et l'exécution de ce modèle par notre outil de simulation, a permis de détecter des incohérences de modélisation à même d'éclairer l'ingénieur système, lui permettant de corriger son modèle.

Au delà de l'aspect technique de la transformation de modèles, « ModelRun » propose un cadre méthologique, permettant aux ingénieurs en charge de la simulation des modèles d'ingénierie de concevoir des ponts sémantiques entre les outils de modélisation et les outils de simulation afin de limiter les biais d'interprétation dans l'analyse des résultats de simulation.

Perspectives

Dans cette deuxième partie, nous proposons des perspectives à la thèse.

Considérer d'autres méta-modèles sources.

Dans cette thèse, nous avons appliqué ModelRun à des modèles d'ingénierie système modélisés avec Capella. ModelRun peut être appliqué à d'autre méta-modèles sources comme SysML. Nous pourrions aussi considérer d'autre champs d'application que la simulation de modèles l'ingénierie système. Par exemple il pourrait être intéressant de transformer des modèles temps réel vers des simulation temps réel, ou transformer des modèles UML vers des environnements d'exécution de modèle.

Cibler d'autres sémantiques d'exécution.

Nous avons appliqué jusqu'à présent ModelRun pour cibler un seul outil de simulation. Nous pourrions standardiser certaines étapes de ModelRun pour cibler plusieurs outils d'analyse. Par exemple on pourrait réutiliser les étapes de sélection et de réorganisation d'un modèle système pour l'analyser par un outil de simulation et par un outil d'analyse formel.

Créer des bibliothèques de transformation

Si nous considérons plusieurs domaines sources et plusieurs domaines cible, il devient intéressant d'envisager la création de bibliothèques d'étapes de transformations. Les différentes bibliothèques inter-connectées permettraient une intégration de plusieurs domaines sources et cibles.

Outiller l'approche ModelRun

ModelRun est une méthode de transformation de modèles. Elle n'est pas liée à un langage ni à un outil particulier. Nous pourrions envisager la création d'un outil de transformation spécifique à ModelRun. Cet outil pourrais s'appuyer sur des opérateurs dédiés et sur les grammaires de graphes triples et intégrer entre autre, une génération automatique des modèles de traçabilité.

Adapter ModelRun aux problématiques de co-évolution

Le modèle de traçabilité est au centre de ModelRun, il permet de créer un lien entre les concepts du modèle source et les concepts du modèle cible. Nous pourrions envisager d'étendre cette propriété en l'adaptant à la co-évolution entre modèle de système et modèle de simulation. Par exemple il serait intéressant de co-développer un modèle d'architecture dans Capella et un modèle de simulation dans Simulink. Les modifications sur le modèle simulation pouvant être répercutées sur le modèle d'ingénierie et vice et versa.

Annexes

Rapport d'expérience N°1

Verification Report for PythagoreUAS

Date : 2021-01-27T14:08:10.314+01:00[Europe/Paris]
The model is invalid, we found incoherencies
4 simulation steps in 87.1942 ms.
The dataFlow contains 129 functions and 288 functional exchanges
The graph contains 64 nodes and 288 edges, with a density of 4.5

16 Cycles have been found

Cycle 1

Create motion 1
Measure motor 1 position
Compute motor 1 rate
7.Compare drone motor rate
7.Disable drone motor
Depower motor 1

Cycle 2

7.Compare drone motor rate
7.Disable drone motor
Depower motor 2
Create motion 2
Measure motor 2 position
Compute motor 2 rate

Cycle 3

Create motion 2
Measure motor 2 position
Compute motor 2 rate
Control motor 2

Cycle 4

7.Compare drone motor rate
7.Disable drone motor
Depower motor 3
Create motion 3
Measure motor 3 position
Compute motor 3 rate

Cycle 5

Create motion 3
Measure motor 3 position
Compute motor 3 rate
Control motor 3

Cycle 6

7.Compare drone motor rate
7.Disable drone motor
Depower motor 4
Create motion 4
Measure motor 4 position
Compute motor 4 rate

Cycle 7

Create motion 4
Measure motor 4 position
Compute motor 4 rate
Control motor 4

Cycle 8

Control motor 1
Create motion 1
Measure motor 1 position
Compute motor 1 rate

Cycle 9

Control speed
Compute thrust and attitude consign
Select attitude consign
Estimate X and Y rates
Compute pitch, roll, yaw rates and angles
Compute ground altitude
Compute altitude and vertical speed

Cycle 10

Control speed
Compute thrust and attitude consign
Select attitude consign
Estimate X and Y rates
Compute pitch, roll, yaw rates and angles
Compute ground altitude
Compute ground speed

Cycle 11

Store data general
Compute the flight plan

Cycle 12

Run flight plan
Display drone follow-up data
Store data general
Compute the flight plan
Acquire and store flight plan

Cycle 13

Compute the flight plan
Detect aircraft position

Cycle 14

Store data general
Analyse mission automatically
Generate mission report

Cycle 15

Make photos and videos
Digitise photos and videos

Cycle 16

Run flight plan
Select drone control mode
Passivate engagement oscillation

Incoherencies found

Situation	Active States	list of invalid Function	invalid caused by
/Manual mode (MP)//Stand by Camera/Stand by Record (in : Video and photo process)	Manual mode (MP) /Stand by Camera/Stand by Record (in : Video and photo process)	Select camera control Control camera orientation Make photos and videos Digitise photos and videos Store Data	Automatic payload control
/Stand-by (in : Autopilot mode (AP))//Stand by Camera/Stand by Record (in : Video and photo process)	Stand-by (in : Autopilot mode (AP)) /Stand by Camera/Stand by Record (in : Video and photo process)	Select camera control Control camera orientation Make photos and videos Digitise photos and videos Store Data	Manual payload control
/Speed consign mode (in : Autopilot mode (AP))//Stand by Camera/Stand by Record (in : Video and photo process)	Speed consign mode (in : Autopilot mode (AP)) /Stand by Camera/Stand by Record (in : Video and photo process)	Select camera control Control camera orientation Make photos and videos Digitise photos and videos Store Data	Manual payload control
/Flight plan mode (in : Autopilot mode (AP))//Stand by Camera/Stand by Record (in : Video and photo process)	Flight plan mode (in : Autopilot mode (AP)) /Stand by Camera/Stand by Record (in : Video and photo process)	Select camera control Control camera orientation Make photos and videos Digitise photos and videos Store Data	Manual payload control

The Story

Evaluted situation	Active states	Validation status	List of transitions	Available transition
/Manual mode (MP)//Stand by Camera/Stand by Record (in : Video and photo process)	Manual mode (MP) /Stand by Camera/Stand by Record (in : Video and photo process)	false	Manual mode (MP) -> Autopilot mode (AP) Manual mode (MP) -> Speed consign mode Stand by Camera -> Point the camera Stand by Record -> Record photo and video	Manual mode (MP) -> Autopilot mode (AP) Manual mode (MP) -> Speed consign mode
/Stand-by (in : Autopilot mode (AP))//Stand by Camera/Stand by Record (in : Video and photo process)	Stand-by (in : Autopilot mode (AP)) /Stand by Camera/Stand by Record (in : Video and photo process)	false	Autopilot mode (AP) -> Manual mode (MP) Autopilot mode (AP) -> Manual mode (MP) Stand-by -> Speed consign mode Stand-by -> Flight plan mode Stand by Camera -> Point the camera Stand by Record -> Record photo and video	Autopilot mode (AP) -> Manual mode (MP) Autopilot mode (AP) -> Manual mode (MP) Stand-by -> Speed consign mode Stand-by -> Flight plan mode
/Speed consign mode (in : Autopilot mode (AP))//Stand by Camera/Stand by Record (in : Video and photo process)	Speed consign mode (in : Autopilot mode (AP)) /Stand by Camera/Stand by Record (in : Video and photo process)	false	Autopilot mode (AP) -> Manual mode (MP) Autopilot mode (AP) -> Manual mode (MP) Stand by Camera -> Point the camera Stand by Record -> Record photo and video	Autopilot mode (AP) -> Manual mode (MP) Autopilot mode (AP) -> Manual mode (MP)
/Flight plan mode (in : Autopilot mode (AP))//Stand by Camera/Stand by Record (in : Video and photo process)	Flight plan mode (in : Autopilot mode (AP)) /Stand by Camera/Stand by Record (in : Video and photo process)	false	Autopilot mode (AP) -> Manual mode (MP) Autopilot mode (AP) -> Manual mode (MP) Flight plan mode -> Manual mode (MP) Stand by Camera -> Point the camera Stand by Record -> Record photo and video	Autopilot mode (AP) -> Manual mode (MP) Autopilot mode (AP) -> Manual mode (MP) Flight plan mode -> Manual mode (MP)

Rapport d’expérience N°2

Verification Report for PythagoreUAS

Date : 2021-01-27T14:18:30.815+01:00[Europe/Paris]
The model is valid
8 simulation steps in 19.7007 ms.
The dataFlow contains 129 functions and 288 functional exchanges
The graph contains 64 nodes and 288 edges, with a density of 4.5

16 Cycles have been found

Cycle 1

Create motion 1
Measure motor 1 position
Compute motor 1 rate
7.Compare drone motor rate
7.Disable drone motor
Depower motor 1

Cycle 2

7.Compare drone motor rate
7.Disable drone motor
Depower motor 2
Create motion 2
Measure motor 2 position
Compute motor 2 rate

Cycle 3

Create motion 2
Measure motor 2 position
Compute motor 2 rate
Control motor 2

Cycle 4

7.Compare drone motor rate
7.Disable drone motor
Depower motor 3
Create motion 3
Measure motor 3 position
Compute motor 3 rate

Cycle 5

Create motion 3
Measure motor 3 position
Compute motor 3 rate
Control motor 3

Cycle 6

7.Compare drone motor rate
7.Disable drone motor
Depower motor 4
Create motion 4
Measure motor 4 position
Compute motor 4 rate

Cycle 7

Create motion 4
Measure motor 4 position
Compute motor 4 rate
Control motor 4

Cycle 8

Control motor 1
Create motion 1
Measure motor 1 position
Compute motor 1 rate

Cycle 9

Control speed
Compute thrust and attitude consign
Select attitude consign
Estimate X and Y rates
Compute pitch, roll, yaw rates and angles
Compute ground altitude
Compute altitude and vertical speed

Cycle 10

Control speed
Compute thrust and attitude consign
Select attitude consign
Estimate X and Y rates
Compute pitch, roll, yaw rates and angles
Compute ground altitude
Compute ground speed

Cycle 11

Store data general
Compute the flight plan

Cycle 12

Run flight plan
Display drone follow-up data
Store data general
Compute the flight plan
Acquire and store flight plan

Cycle 13

Compute the flight plan
Detect aircraft position

Cycle 14

Store data general
Analyse mission automatically
Generate mission report

Cycle 15

Make photos and videos
Digitise photos and videos

Cycle 16

Run flight plan
Select drone control mode
Passivate engagement oscillation

The Story

Evaluted situation	Active states	Validation status	List of transitions	Available transition
/Manual mode (MP)//Stand by Camera/Stand by Record (in : Video and photo process)	Manual mode (MP) /Stand by Camera/Stand by Record (in : Video and photo process)	true	Manual mode (MP) -> Autopilot mode (AP) Manual mode (MP) -> Speed consign mode Stand by Camera -> Point the camera Stand by Record -> Record photo and video	Manual mode (MP) -> Autopilot mode (AP) Manual mode (MP) -> Speed consign mode Stand by Camera -> Point the camera
/Stand-by (in : Autopilot mode (AP))//Stand by Camera/Stand by Record (in : Video and photo process)	Stand-by (in : Autopilot mode (AP)) /Stand by Camera/Stand by Record (in : Video and photo process)	true	Autopilot mode (AP) -> Manual mode (MP) Autopilot mode (AP) -> Manual mode (MP) Stand-by -> Speed consign mode Stand-by -> Flight plan mode Stand by Camera -> Point the camera Stand by Record -> Record photo and video	Autopilot mode (AP) -> Manual mode (MP) Autopilot mode (AP) -> Manual mode (MP) Stand-by -> Speed consign mode Stand-by -> Flight plan mode Stand by Camera -> Point the camera
/Speed consign mode (in : Autopilot mode (AP))//Stand by Camera/Stand by Record (in : Video and photo process)	Speed consign mode (in : Autopilot mode (AP)) /Stand by Camera/Stand by Record (in : Video and photo process)	true	Autopilot mode (AP) -> Manual mode (MP) Autopilot mode (AP) -> Manual mode (MP) Stand by Camera -> Point the camera Stand by Record -> Record photo and video	Autopilot mode (AP) -> Manual mode (MP) Autopilot mode (AP) -> Manual mode (MP) Stand by Camera -> Point the camera
/Manual mode (MP)//Point the camera/Stand by Record (in : Video and photo process)	Manual mode (MP) /Point the camera/Stand by Record (in : Video and photo process)	true	Manual mode (MP) -> Autopilot mode (AP) Manual mode (MP) -> Speed consign mode Point the camera -> Position ready Stand by Record -> Record photo and video	Manual mode (MP) -> Autopilot mode (AP) Manual mode (MP) -> Speed consign mode
/Flight plan mode (in : Autopilot mode (AP))//Stand by Camera/Stand by Record (in : Video and photo process)	Flight plan mode (in : Autopilot mode (AP)) /Stand by Camera/Stand by Record (in : Video and photo process)	true	Autopilot mode (AP) -> Manual mode (MP) Autopilot mode (AP) -> Manual mode (MP) Flight plan mode -> Manual mode (MP) Stand by Camera -> Point the camera Stand by Record -> Record photo and video	Autopilot mode (AP) -> Manual mode (MP) Autopilot mode (AP) -> Manual mode (MP) Flight plan mode -> Manual mode (MP) Stand by Camera -> Point the camera
/Stand-by (in : Autopilot mode (AP))//Point the camera/Stand by Record (in : Video and photo process)	Stand-by (in : Autopilot mode (AP)) /Point the camera/Stand by Record (in : Video and photo process)	true	Autopilot mode (AP) -> Manual mode (MP) Autopilot mode (AP) -> Manual mode (MP) Stand-by -> Speed consign mode Stand-by -> Flight plan mode Point the camera -> Position ready Stand by Record -> Record photo and video	Autopilot mode (AP) -> Manual mode (MP) Autopilot mode (AP) -> Manual mode (MP) Stand-by -> Speed consign mode Stand-by -> Flight plan mode
/Speed consign mode (in : Autopilot mode (AP))//Point the camera/Stand by Record (in : Video and photo process)	Speed consign mode (in : Autopilot mode (AP)) /Point the camera/Stand by Record (in : Video and photo process)	true	Autopilot mode (AP) -> Manual mode (MP) Autopilot mode (AP) -> Manual mode (MP) Point the camera -> Position ready Stand by Record -> Record photo and video	Autopilot mode (AP) -> Manual mode (MP) Autopilot mode (AP) -> Manual mode (MP)
/Flight plan mode (in : Autopilot mode (AP))//Point the camera/Stand by Record (in : Video and photo process)	Flight plan mode (in : Autopilot mode (AP)) /Point the camera/Stand by Record (in : Video and photo process)	true	Autopilot mode (AP) -> Manual mode (MP) Autopilot mode (AP) -> Manual mode (MP) Flight plan mode -> Manual mode (MP) Point the camera -> Position ready Stand by Record -> Record photo and video	Autopilot mode (AP) -> Manual mode (MP) Autopilot mode (AP) -> Manual mode (MP) Flight plan mode -> Manual mode (MP)

BIBLIOGRAPHIE

[1] J.-L. VOIRIN, *Conception architecturale des systèmes basée sur les modèles avec la méthode Arcadia*. ISTE Group, 2018, t. 3.

[2] INCOSE, *about INCOSE, web-page.* Accessed August 18, 2020. https://www.incose.org/about-incose.

[3] ——, *about Systems Engineering, web-page.* Accessed August 18, 2020. https://www.incose.org/about-incose.

[4] F. VARENNE, « Epistémologie des modèles et des simulations », 2008.

[5] B. HENDERSON-SELLERS, *On the mathematics of modelling, metamodelling, ontologies and modelling languages*. Springer Science & Business Media, 2012.

[6] J. BÉZIVIN et O. GERBÉ, « Towards a precise definition of the OMG/MDA framework », in *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, IEEE, 2001, p. 273–280.

[7] T. KÜHNE, « What is a Model ? », in *Dagstuhl Seminar Proceedings*, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2005.

[8] G. GUIZZARDI, « On ontology, ontologies, conceptualizations, modeling languages, and (meta) models », *Frontiers in artificial intelligence and applications*, t. 155, p. 18, 2007.

[9] S. ULLMANN, *Semantics : an introduction to the science of meaning*. Barnes & Noble, 1962.

[10] C. K. OGDEN, I. A. RICHARDS, B. MALINOWSKI et F. G. CROOKSHANK, *The meaning of meaning*. Kegan Paul London, 1923.

[11] F. DE SAUSSURE, *Course in general linguistics*. Columbia University Press, 2011, (original from 1916).

[12] D. VARRÓ et A. BALOGH, « The model transformation language of the VIATRA2 framework », *Science of Computer Programming*, t. 68, n° 3, p. 214–234, 2007.

[13] T. CLARK, P. SAMMUT et J. WILLANS, *Applied metamodelling : a foundation for language driven development.* 2008.

[14] J. BÉZIVIN, « In search of a basic principle for model driven engineering », *Novatica Journal, Special Issue*, t. 5, n° 2, p. 21–24, 2004.

[15] K. EHRIG, J. M. KÜSTER et G. TAENTZER, « Generating instance models from meta models », *Software & Systems Modeling*, t. 8, n° 4, p. 479–500, 2009.

[16] J. BÉZIVIN, « On the unification power of models », *Software & Systems Modeling*, t. 4, n° 2, p. 171–188, 2005.

[17] R. FRANCE et B. RUMPE, « Model-driven development of complex software : A research roadmap », in *Future of Software Engineering (FOSE'07)*, IEEE, 2007, p. 37–54.

[18] C. ATKINSON et T. KUHNE, « Model-driven development : a metamodeling foundation », *IEEE software*, t. 20, n° 5, p. 36–41, 2003.

[19] C. ATKINSON, M. GUTHEIL et B. KENNEL, « A flexible infrastructure for multilevel language engineering », *IEEE Transactions on Software Engineering*, t. 35, n° 6, p. 742–755, 2009.

[20] C. GONZALEZ-PEREZ et B. HENDERSON-SELLERS, « A powertype-based metamodelling framework », *Software & Systems Modeling*, t. 5, n° 1, p. 72–90, 2006.

[21] C. ATKINSON et T. KÜHNE, « The essence of multilevel metamodeling », in *International Conference on the Unified Modeling Language*, Springer, 2001, p. 19–33.

[22] S. SENDALL et W. KOZACZYNSKI, « Model transformation : The heart and soul of model-driven software development », *IEEE software*, t. 20, n° 5, p. 42–45, 2003.

[23] A. G. KLEPPE, J. WARMER, J. B. WARMER et W. BAST, *MDA explained : the model driven architecture : practice and promise*. Addison-Wesley Professional, 2003.

[24] T. MENS et P. VAN GORP, « A taxonomy of model transformation », *Electronic notes in theoretical computer science*, t. 152, p. 125–142, 2006.

[25] K. CZARNECKI et S. HELSEN, « Feature-based survey of model transformation approaches », *IBM systems journal*, t. 45, nº 3, p. 621–645, 2006.

[26] ——, « Classification of model transformation approaches », in *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, USA, t. 45, 2003, p. 1–17.

[27] L. LÚCIO, M. AMRANI, J. DINGEL, L. LAMBERS, R. SALAY, G. M. SELIM, E. SYRIANI et M. WIMMER, « Model transformation intents and their properties », *Software & systems modeling*, t. 15, nº 3, p. 647–684, 2016.

[28] W. SUN, B. COMBEMALE, S. DERRIEN et R. B. FRANCE, « Using model types to support contract-aware model substitutability », in *European Conference on Modelling Foundations and Applications*, Springer, 2013, p. 118–133.

[29] J. STEEL et J.-M. JÉZÉQUEL, « On model typing », *Software & Systems Modeling*, t. 6, nº 4, p. 401–413, 2007.

[30] T. INCOSE, « Systems engineering vision 2020 », *INCOSE, San Diego, CA, accessed Jan*, t. 26, p. 2019, 2007.

[31] P. ROQUES, *Modélisation de systèmes complexes avec SysML*. Editions Eyrolles, 2013.

[32] E. HERZOG et A. TÖRNE, « 5.6. 1 AP-233 Architecture », in *INCOSE International Symposium*, Wiley Online Library, t. 10, 2000, p. 765–772.

[33] S. FRIEDENTHAL, A. MOORE et R. STEINER, « Omg systems modeling language tutorial », *Proceedings of Incose 2007, San Diego*, 2007.

[34] M. HAUSE et al., « The SysML modelling language », in *Fifteenth European Systems Engineering Conference*, t. 9, 2006, p. 1–12.

[35] M. RASHID, M. W. ANWAR et A. M. KHAN, « Toward the tools selection in model based system engineering for embedded systems—A systematic literature review », *Journal of Systems and Software*, t. 106, p. 150–163, 2015.

[36] NOMAGIC, *MagicDraw SysML Plugin, web-page.* Accessed August 27, 2020. https://www.nomagic.com/product-addons/magicdraw-addons/sysml-plugin.

[37] MODELIO, *SysML Achitect, web-page.* Accessed August 27, 2020. `https://store.modelio.org/resource/modules/sysml-architect-open-source.html`.

[38] IBM, *Rhapsody - Architect for Systems Engineers, web-page.* Accessed August 27, 2020. `https://www.ibm.com/products/architect-for-systems-engineers`.

[39] PAPYRUS, *Eclipse Papyrus SysML project, web-page.* Accessed August 27, 2020. `https://www.eclipse.org/papyrus/components/sysml/0.9.0/user/tuto1-createsysmlproject.html`.

[40] S. BONNET, J.-L. VOIRIN, V. NORMAND et D. EXERTIER, « Implementing the mbse cultural change : Organization, coaching and lessons learned », in *INCOSE International Symposium*, Wiley Online Library, t. 25, 2015, p. 508–523.

[41] P. ROQUES, *Systems Architecture Modeling with the Arcadia Method : A Practical Guide to Capella*. Elsevier, 2017.

[42] ——, « MBSE with the ARCADIA Method and the Capella Tool », 2016.

[43] E. HERZOG, J. HALLONQUIST et J. NAESER, « 4.5. 1 Systems Modeling with SysML–an experience report », in *INCOSE International Symposium*, Wiley Online Library, t. 22, 2012, p. 600–611.

[44] P. PIHLANKO, S. SIERLA, K. THRAMBOULIDIS et M. VIITASALO, « An industrial evaluation of SysML : The case of a nuclear automation modernization project », in *2013 IEEE 18th Conference on Emerging Technologies & Factory Automation (ETFA)*, IEEE, 2013, p. 1–8.

[45] G. KULKARNI et S. PRICE, « MBSE Model on Gas Turbine Tip Clearance Control », in *Gas Turbine India Conference*, American Society of Mechanical Engineers, t. 83532, 2019, V002T11A001.

[46] H. FECHER, J. SCHÖNBORN, M. KYAS et W.-P. de ROEVER, « 29 new unclarities in the semantics of uml 2.0 state machines », in *International Conference on Formal Engineering Methods*, Springer, 2005, p. 52–65.

[47] B. COMBEMALE, X. CRÉGUT, P.-L. GAROCHE et X. THIRIOUX, « Essay on semantics definition in MDE. An instrumented approach for model verification », *Journal of Software*, t. 4, nº 9, p. 943–958, 2009.

[48] G. ROZENBERG, *Handbook of graph grammars and computing by graph transformation*. World scientific, 1997, t. 1.

[49] H. EHRIG, G. ENGELS, H. J. KREOWSKI et G. ROZENBERG, *Handbook of graph grammars and computing by graph transformation : vol. 2 : applications, languages, and tools*. 1999.

[50] H. EHRIG, U. PRANGE et G. TAENTZER, « Fundamental theory for typed attributed graph transformation », in *International conference on graph transformation*, Springer, 2004, p. 161–177.

[51] HARTMUT EHRIG, CLAUDIA ERMEL, ULRIKE GOLAS et FRANK HERMANN, *Graph and Model Transformation General Framework and Applications*, sér. Monographs in Theoretical Computer Science. An EATCS Series. Springer, Berlin, Heidelberg, 2015. adresse : `https://doi.org/10.1007/978-3-662-47980-3`.

[52] E. BIERMANN, C. ERMEL et G. TAENTZER, « Precise semantics of EMF model transformations by graph transformation », in *International Conference on Model Driven Engineering Languages and Systems*, Toulouse, France : Springer, 2008, p. 53–67.

[53] J. de LARA, R. BARDOHL, H. EHRIG, K. EHRIG, U. PRANGE et G. TAENTZER, « Attributed graph transformation with node type inheritance », *Theoretical Computer Science*, t. 376, n° 3, p. 139–163, 2007.

[54] A. CORRADINI, U. MONTANARI et F. ROSSI, « Graph processes », *Fundamenta Informaticae*, t. 26, n° 3, 4, p. 241–265, 1996.

[55] S. JURACK et G. TAENTZER, « A component concept for typed graphs with inheritance and containment structures », in *International Conference on Graph Transformation*, Springer, 2010, p. 187–202.

[56] R. J. MOKKEN et al., « Cliques, clubs and clans », *Quality & Quantity*, t. 13, n° 2, p. 161–173, 1979.

[57] H. EHRIG et B. MAHR, *Fundamentals of algebraic specification 1 : Equations and initial semantics*. Springer Science & Business Media, 2012, t. 6.

[58] M. POUDRET, « Transformations de graphes pour les opérations topologiques en modélisation géométrique : application à l'étude de la dynamique de l'appareil de Golgi », thèse de doct., Evry-Val d'Essonne, 2009.

[59] G. TAENTZER, U. PRANGE, K. EHRIG et H. EHRIG, *Fundamentals of algebraic graph transformation.* Springer-Verlag Berlin Heidelberg, sér. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2006.

[60] A. SCHÜRR, « Specification of graph translators with triple graph grammars », in *International Workshop on Graph-Theoretic Concepts in Computer Science*, Springer, 1994, p. 151–163.

[61] A. KÖNIGS et A. SCHÜRR, « Tool integration with triple graph grammars-a survey », *Electronic Notes in Theoretical Computer Science*, t. 148, n° 1, p. 113–150, 2006.

[62] H. EHRIG, K. EHRIG, C. ERMEL, F. HERMANN et G. TAENTZER, « Information preserving bidirectional model transformations », in *International Conference on Fundamental Approaches to Software Engineering*, Springer, 2007, p. 72–86.

[63] A. HABEL, R. HECKEL et G. TAENTZER, « Graph grammars with negative application conditions », *Fundamenta Informaticae*, t. 26, n° 3, 4, p. 287–313, 1996.

[64] F. HERMANN, M. HÜLSBUSCH et B. KÖNIG, « Specification and verification of model transformations », *Electronic Communications of the EASST*, t. 30, 2010.

[65] G. TAENTZER, « AGG : A graph transformation environment for modeling and validation of software », in *International Workshop on Applications of Graph Transformations with Industrial Relevance*, Springer, 2003, p. 446–453.

[66] D. BALASUBRAMANIAN, A. NARAYANAN, C. van BUSKIRK et G. KARSAI, « The graph rewriting and transformation language : GReAT », *Electronic Communications of the EASST*, t. 1, 2007.

[67] S. WINKLER et J. von PILGRIM, « A survey of traceability in requirements engineering and model-driven development », *Software & Systems Modeling*, t. 9, n° 4, p. 529–565, 2010.

[68] O. C. Z. GOTEL, « Contribution structures for requirements traceability », thèse de doct., University of London, 1995.

[69] B. RAMESH et M. JARKE, « Toward reference models for requirements traceability », *IEEE transactions on software engineering*, t. 27, n° 1, p. 58–93, 2001.

[70] O. C. GOTEL et C. FINKELSTEIN, « An analysis of the requirements traceability problem », in *Proceedings of IEEE International Conference on Requirements Engineering*, IEEE, 1994, p. 94–101.

[71] « IEEE Standard Glossary of Software Engineering Terminology », *IEEE Std 610.12-1990*, p. 1–84, 1990.

[72] N. AIZENBUD-RESHEF, B. T. NOLAN, J. RUBIN et Y. SHAHAM-GAFNI, « Model traceability », *IBM Systems Journal*, t. 45, n° 3, p. 515–526, 2006.

[73] G. SPANOUDAKIS et A. ZISMAN, « Software traceability : a roadmap », in *Handbook Of Software Engineering And Knowledge Engineering : Vol 3 : Recent Advances*, World Scientific, 2005, p. 395–428.

[74] D. S. KOLOVOS, R. F. PAIGE et F. A. POLACK, « On-demand merging of traceability links with models », in *3rd ecmda traceability workshop*, 2006, p. 47–55.

[75] N. DRIVALOS, R. F. PAIGE, K. J. FERNANDES et D. S. KOLOVOS, « Towards rigorously defined model-to-model traceability », in *ECMDA Traceability Workshop (ECMDA-TW)*, 2008, p. 17–26.

[76] W. HEAVEN et A. FINKELSTEIN, « UML profile to support requirements engineering with KAOS », *IEE Proceedings-Software*, t. 151, n° 1, p. 10–27, 2004.

[77] N. MUSTAFA et Y. LABICHE, « Modeling traceability for heterogeneous systems », in *2015 10th International Joint Conference on Software Technologies (ICSOFT)*, IEEE, t. 1, 2015, p. 1–9.

[78] Z. DISKIN, A. GÓMEZ et J. CABOT, « Traceability mappings as a fundamental instrument in model transformations », in *International Conference on Fundamental Approaches to Software Engineering*, Springer, 2017, p. 247–263.

[79] J.-M. JÉZÉQUEL, O. BARAIS et F. FLEUREY, « Model driven language engineering with kermeta », in *International Summer School on Generative and Transformational Techniques in Software Engineering*, Springer, 2009, p. 201–221.

[80] R. F. PAIGE, D. S. KOLOVOS et F. A. POLACK, « An action semantics for MOF 2.0 », in *Proceedings of the 2006 ACM symposium on Applied computing*, 2006, p. 1304–1305.

[81] J. E. RIVERA et A. VALLECILLO, « Adding Behavioral Semantics to models », in *11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007)*, IEEE, 2007, p. 169–180.

[82] M. Clavel, F. Duran, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer et C. Talcott, *All About Maude-A High-Performance Logical Framework : How to Specify, Program, and Verify Systems in Rewriting Logic*. Springer, 2007, t. 4350.

[83] B. Combemale, C. Brun, J. Champeau, X. Crégut, J. Deantoni et J. Le Noir, « A Tool-Supported Approach for Concurrent Execution of Heterogeneous Models », 2016.

[84] E. Bousse, T. Degueule, D. Vojtisek, T. Mayerhofer, J. Deantoni et B. Combemale, « Execution framework of the gemoc studio (tool demo) », in *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, 2016, p. 84–89.

[85] A. Sindico, M. Di Natale et G. Panci, « Integrating SysML with Simulink using Open-source Model Transformations. », in *SIMULTECH*, 2011, p. 45–56.

[86] MathWorks, *Simulink, web-page.* Accessed september 04, 2020. https://fr.mathworks.com/products/simulink.html.

[87] Obeo, *Acceleo, web-page.* Accessed september 04, 2020. https://www.eclipse.org/acceleo/.

[88] A. A. Shah, A. A. Kerzhner, D. Schaefer et C. J. Paredis, « Multi-view modeling to support embedded systems engineering in SysML », in *Graph transformations and model-driven engineering*, Springer, 2010, p. 580–601.

[89] M. Association et al., « Modelica-A unified objectoriented language for physical systems modeling-Language specification version 3.2 », *Online, Sep*, t. 5, 2010, Accessed september 1, 2020. https://www.modelica.org/documents/ModelicaSpec32.pdf.

[90] J.-M. Gauthier, F. Bouquet, A. Hammad et F. Peureux, « Tooled process for early validation of SysML models using Modelica simulation », in *International Conference on Fundamentals of Software Engineering*, Springer, 2015, p. 230–237.

[91] C. J. Paredis, Y. Bernard, R. M. Burkhart, H.-P. de Koning, S. Friedenthal, P. Fritzson, N. F. Rouquette et W. Schamai, « 5.5. 1 An overview of the SysML-modelica transformation specification », in *INCOSE International Symposium*, Wiley Online Library, t. 20, 2010, p. 709–722.

[92] T. JOHNSON, A. KERZHNER, C. J. PAREDIS et R. BURKHART, « Integrating models and simulations of continuous dynamics into SysML », *Journal of Computing and Information Science in Engineering*, t. 12, nº 1, 2012.

[93] B. CHABIBI, A. ANWAR et M. NASSAR, « Towards a Model Integration from SysML to MATLAB/Simulink. », *JSW*, t. 13, nº 12, p. 630–645, 2018.

[94] M. DI NATALE, F. CHIRICO, A. SINDICO et A. SANGIOVANNI-VINCENTELLI, « An MDA approach for the generation of communication adapters integrating SW and FW components from Simulink », in *International Conference on Model Driven Engineering Languages and Systems*, Springer, 2014, p. 353–369.

[95] C. J. PAREDIS et T. JOHNSON, « Using OMG's SysML to support simulation », in *2008 Winter Simulation Conference*, IEEE, 2008, p. 2350–2352.

[96] D. VARRÓ, G. BERGMANN, Á. HEGEDÜS, Á. HORVÁTH, I. RÁTH et Z. UJHELYI, « Road to a reactive and incremental model transformation platform : three generations of the VIATRA framework », *Software & Systems Modeling*, t. 15, nº 3, p. 609–629, 2016.

[97] D. C. CAFE, F. V. DOS SANTOS, C. HARDEBOLLE, C. JACQUET et F. BOULANGER, « Multi-paradigm semantics for simulating SysML models using SystemC-AMS », in *Proceedings of the 2013 Forum on specification and Design Languages (FDL)*, IEEE, 2013, p. 1–8.

[98] C. GRIMM, M. BARNASCONI, A. VACHOUX et K. EINWICH, « An introduction to modeling embedded analog/mixed-signal systems using SystemC AMS extensions », in *DAC2008 International Conference*, sn, t. 23, 2008.

[99] E. J. MACIAS et M. P. de la PARTE, « Simulation and optimization of logistic and production systems using discrete and continuous Petri nets », *Simulation*, t. 80, nº 3, p. 143–152, 2004.

[100] P. VALLEJO, J.-P. BABAU et M. KERBOEUF, « ModifRoundtrip : A Model-Based tool to reuse legacy transformations. », in *D&P@ MoDELS*, 2016, p. 72–79.

[101] L. M. ROSE, D. S. KOLOVOS, R. F. PAIGE et F. A. POLACK, « Model migration with epsilon flock », in *International Conference on Theory and Practice of Model Transformations*, Springer, 2010, p. 184–198.

[102] C. DUHIL, J.-P. BABAU, E. LÉPICIER, J.-L. VOIRIN et J. NAVAS, « Chaining model transformations for system model verification : application to verify capella model with simulink », in *8th International Conference on Model-Driven Engineering and Software Development*, SCITEPRESS-Science et Technology Publications, 2020, p. 279–286.

[103] ——, « Chaining model transformations to develop a system model verification tool : application to capella state machines and data flows models », in *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, 2020, p. 1654–1657.

[104] C. DUHIL, J.-L. VOIRIN, E. LÉPICIER et J.-P. BABAU, « Early Detection of Flaws in System Architecture Model by means of Model Simulation », in *INCOSE International Symposium*, Wiley Online Library, t. 30, 2020, p. 1746–1757.

[105] E. ROCHE et Y. SCHABES, *Finite-state language processing*. MIT press, 1997.

[106] E. A. LEE et D. G. MESSERSCHMITT, « Synchronous data flow », *Proceedings of the IEEE*, t. 75, nº 9, p. 1235–1245, 1987.

[107] J. A. BERGSTRA, C. A. MIDDELBURG et G. ŞTEF [ACARON] NESCU, « Network algebra for asynchronous dataflow », *International Journal of Computer Mathematics*, t. 65, nº 1-2, p. 57–88, 1997.

DOCTORAT BRETAGNE LOIRE / MATHSTIC

UBO
Université de Bretagne Occidentale

Titre : ModelRun, une méthode de transformations de modèles pour la vérification de propriétés de modèles de systèmes complexes par simulation.

Mot clés : Ingénierie Dirigée par les Modèles, transformation de modèles, ingénierie système, simulation de modèles, grammaires de graphes.

Résumé : Pour maîtriser la conception de systèmes complexes, tels que les véhicules, les drones, les satellites, les ingénieurs utilisent des méthodes de conception basées sur les modèles. Les systèmes complexes sont modélisés sous la forme d'un ensemble de diagrammes généralement exprimés dans des sémantiques hétérogènes mais non exécutables. Le besoin émerge alors de vérifier la cohérence des modèles entre eux, notamment dans leur aspect comportemental. Dans cette thèse nous proposons une méthode appelée « ModelRun » permettant de transformer un ensemble de vues d'un modèle d'ingénierie en un modèle exécutable. Notre méthode comprend cinq étapes ; la sélection des concepts à vérifier, la réorganisation de ces concepts en vue de l'alignement avec le domaine de simulation, l'alignement avec les concepts du domaine de simulation, l'ajout d'informations nécessaires à la simulation et l'adaptation de ces informations. Notre méthode s'appuie sur un formalisme de grammaire de graphes triples. Chaque transformation fait l'objet d'une traçabilité permettant d'interpréter les résultats des simulations au regard des concepts sources du modèle d'ingénierie. Pour valider notre approche, nous appliquons « ModelRun » à un modèle de système de drone.

Title: ModelRun, a model transformation method for complex system models properties checking by simulation.

Keywords: Model driven engineering, model transformation, system engineering, model simulation, graph grammar

Abstract: To master the design of complex systems, such as vehicles, UAVs, satellites, engineers use model-based design methods. Complex systems are modelled as a set of diagrams usually based on heterogeneous semantics. The need then emerges to verify the coherence of the models between them, particularly in their behavioural aspect. In this thesis, we propose a method called« ModelRun » to transform a set of views of an engineering model into an executable model. Our method consists of five steps; the selection of the concepts to be verified, the refactoring of these concepts in order to map them with the simulation domain, the mapping with the concepts of the simulation domain, the enrichment with information necessary for the simulation and the adaptation of this information. Our method is based on a triple graph grammar formalism. Each transformation is subject to a traceability allowing the interpretation of the simulation results with respect to the source concepts of the engineering model. To validate our approach, we apply « ModelRun » to a UAV system model.