N° d'ordre : 853

THÈSE

présentée à

l'Institut National des Sciences Appliquées de Toulouse

pour l'obtention du titre de

DOCTEUR

de l'Institut National des Sciences Appliquées de Toulouse

Spécialité : Informatique

par

Hugues MALGOUYRES

Laboratoire d'Étude des Systèmes Informatiques et Automatiques
École Doctorale Systèmes

Titre de la thèse :

Définition et détection automatique des incohérences structurelles et comportementales des modèles UML - Couplage des techniques de métamodélisation et de vérification basée sur la programmation logique

Soutenue le 28 Novembre 2006, devant le jury :

Rapporteurs :	Jeanine Souquières	Professeur à l'Université de Nancy 2
	Bernard Coulette	Professeur à l'Université de Toulouse le Mirail
	Jean-Louis Sourrouille	Professeur à l'INSA de Lyon
Examinateurs :	Gilles Motet	Professeur à l'INSA de Toulouse - Directeur de thèse
Membre invité :	Frédéric Boniol	Professeur à l'ENSEEIHT
	Christian Percebois	Professeur à l'Université Paul Sabatier de Toulouse
	Thierry Le Saux	Industriel, Thales Avionics

Table des matières

1 Introduction **1**

1.1 Domaine et motivation 1

1.2 Le langage UML 2

1.2.1 Pourquoi un langage unifié ? 2

1.2.2 Vue d'ensemble du formalisme UML 2

1.2.3 Métamodélisation 4

1.3 Cohérence des modèles UML 5

1.3.1 Gestion des incohérences dans un processus de développement . . 5

1.3.2 La cohérence dans les modèles UML 6

1.4 Maîtrise de la cohérence des modèles UML par la gestion des risques . . . 9

1.4.1 Risque d'incohérence 9

1.4.2 Vue d'ensemble des travaux 10

2 Identification des règles de cohérence **13**

2.1 Types d'approches 13

2.1.1 Définition par règles 13

2.1.2 Définition par transformation 15

2.1.3 Approche proposée 16

2.2 Vue d'ensemble du résultat 17

2.2.1 Organisation du document 17

2.2.2 Résultats quantitatifs 18

2.3 Exemples de règles de cohérence 19

2.3.1 Règle déduite du métamodèle 20

2.3.2 Nouvelle règle de cohérence 20

2.3.3 Règle tirée de la spécification d'UML 23

2.4 Discussion 24

2.4.1 Limites 24

2.4.2 Évaluation d'outils 25

2.4.3 Conclusion 26

3 Analyse de la cohérence **27**

3.1 Techniques de détection 27

3.1.1 Cadre de la vérification de cohérence 27

3.1.2 Techniques proposées 30

3.1.3 Limites des travaux actuels et besoins 33

3.2 Principe de l'approche 34

3.2.1 Choix du langage d'analyse 34

3.2.2 Introduction aux CLP . . . 35
3.3 Conclusion . . . 41

4 Détection des incohérences structurelles **43**
4.1 Vue d'ensemble de la démarche . . . 43
4.2 Encodage des modèles UML en programmation logique . . . 45
4.2.1 Niveau langage . . . 45
4.2.2 Niveau méta-langage . . . 50
4.2.3 Synthèse . . . 51
4.2.4 Outillage et conclusion . . . 54
4.3 Formalisation des règles de cohérence et diagnostic . . . 55
4.3.1 Exemple . . . 55
4.3.2 Règles supplémentaires facilitant l'expression des incohérences . . . 56
4.3.3 Règles vérifiant le respect du métamodèle . . . 58
4.4 Conclusion . . . 60

5 Détection des incohérences comportementales **63**
5.1 Vue d'ensemble de l'approche . . . 64
5.1.1 Étape A . . . 65
5.1.2 Étape B . . . 65
5.1.3 Étape C . . . 67
5.2 Description de la configuration des modèles UML . . . 68
5.2.1 Exemples de configuration . . . 69
5.2.2 Configuration globale d'un modèle UML . . . 71
5.2.3 Transformation en (C)LP . . . 72
5.2.4 Conclusion . . . 74
5.3 Expression de comportement en (C)LP . . . 75
5.3.1 Éléments agissant sur le comportement des modèles UML . . . 76
5.3.2 Exemple de règles de changement de configuration . . . 77
5.3.3 Concepts de niveau méta-métamodèle . . . 81
5.3.4 Points de variation sémantique . . . 82
5.4 Expression des propriétés . . . 83
5.4.1 Cohérence faisant intervenir des invariants de configuration . . . 84
5.4.2 Expression de cohérence sur l'ensemble des traces . . . 86
5.5 Conclusion . . . 87

6 Applications **89**
6.1 Constructions prises en compte . . . 89
6.2 Résultats sur le problème des philosophes . . . 90
6.3 Résultats sur un modèle industriel . . . 92
6.3.1 Description du modèle . . . 92
6.3.2 Résultats . . . 94
6.4 Conclusion . . . 97

7 Conclusion **99**
7.1 Synthèse . . . 99
7.2 Perpectives . . . 101

A Définition de l'encodage des modèles UML en XMI **103**

B Présentation de l'outil de traduction UML → CLP **107**
B.1 Analyse du métamodèle 107
B.1.1 Exemple de métamodèle et de son encodage 107
B.1.2 Vue d'ensemble de l'analyse du métamodèle 109
B.1.3 Extraction des méta-faits sans attributs hérités 110
B.1.4 Extraction des méta-généralisations 112
B.1.5 Calcul des méta-faits complets 113
B.1.6 Calcul des méta-généralisations entre méta-faits complets 114
B.2 Analyse du modèle 114
B.2.1 Exemple de modèle 115
B.2.2 Instanciation des faits 115
B.2.3 Génération des faits avec la syntaxe voulue 117

C Code du checker **119**
C.1 Sémantique des constructions d'une machine à états 119
C.2 Sémantique des actions 122
C.3 Expression des propriétés 126
C.4 Diagnostic d'incohérence comportementale 127
C.5 Prédicats annexes 128

D Représentation en LP du modèle des philosophes **131**

E Exemple de trace **135**

Bibliographie **139**

Index **144**

Chapitre 1

Introduction

1.1 Domaine et motivation

Le développement de systèmes complexes a induit l'utilisation de formalismes de modélisation autorisant l'expression de vues abstraites de ces systèmes. Ces formalismes permettent de décrire le résultat des différentes phases du processus de développement, par exemple de spécifier le système attendu ou d'en concevoir la réalisation. La structuration en vues peut être adoptée pour réduire la complexité du problème en se focalisant sur chaque aspect du système (structure de données, déploiement de l'application, comportement de l'application, etc.) [42]. Le langage UML (*Unified Modeling Language* [59]) est un langage graphique permettant l'expression de multiples vues.

Durant de nombreuses années, les formalismes graphiques n'étaient utilisés que dans un but de documentation et de communication entre les différentes parties intéressées. La correction du système réalisé n'était examinée qu'une fois une réalisation technologique obtenue. Dans notre cas, il s'agit d'un programme exécutable ou d'un programme source. De nombreuses techniques ont été développées dont le test fonctionnel sur les programmes exécutables et le test structurel ou la preuve de propriétés sur les programmes sources. De ce fait, la détection de fautes dues à des erreurs commises lors des premières phases de développement engendre des coûts importants : temps perdu par la conception sur des bases erronées, diagnostic difficile car les choix initiaux transparaissent souvent peu dans l'implantation, etc. Il est donc paru opportun d'évaluer la correction des modèles exprimés dès leurs créations afin de réduire ces dépenses inutiles. L'évaluation de cette correction est cependant plus complexe car les formalismes de modélisation utilisent des notions souvent plus abstraites et plus diverses et les auteurs/lecteurs de ces modèles peuvent avoir de multiples interprétations de ces notions lorsque leur sens n'est pas rigoureusement défini.

Nos travaux se situent dans ce cadre de la vérification de modèles en limitant cependant l'ampleur du sujet par deux restrictions :

- le choix d'un seul formalisme de modélisation (UML) ;
- l'étude d'un sous-ensemble de fautes de modélisation, celles exprimant une incohérence dans les modèles.

Malgré la limitation de ces ambitions, la détection des incohérences des modèles UML est un enjeu scientifique et industriel majeur. En effet, comme nous l'établirons lors de l'application de nos résultats théoriques, la modélisation des systèmes commerciaux est

constituée de centaines de diagrammes développés par des dizaines d'ingénieurs. Garantir la cohérence de l'ensemble est donc une activité importante qui ne peut être effectuée que par un outil manipulant cet ensemble important de données exprimées.

1.2 Le langage UML

Cette section propose de donner une vue d'ensemble du langage UML. La section 1.2.1 explique les motivations de la création de ce langage, la section 1.2.2 donne une vue d'ensemble des formalismes utilisés et la section 1.2.3 détaille le moyen de définition du langage UML, à savoir la métamodélisation.

1.2.1 Pourquoi un langage unifié ?

UML est né de la fusion des trois méthodes qui ont le plus influencé la modélisation objet au milieu des années 90 : OMT, Booch et OOSE. Unifier les concepts et la représentation qui leur est associée a trois intérêts majeurs. Premièrement, cela a contribué à sélectionner les formalismes les plus adaptés qui avaient fait leurs preuves dans des développements passés. De plus, fournir un langage commun a permis aux concepteurs de se focaliser sur les problèmes fondamentaux du système en évitant les confusions dues à l'emploi de notations différentes pour des concepts identiques. Enfin, stabiliser le formalisme et les concepts permet aux développeurs d'AGL (Ateliers de Génie Logiciel) d'apporter des fonctionnalités de plus en plus avancées.

D'autre part, l'aspect méthodologique a été exclu de la définition d'UML. Une méthode régit l'enchaînement des activités au sein d'un projet et définit les produits délivrés par chacune d'entre elles. Ce sujet est primordial mais les concepteurs d'UML ont jugé qu'une méthode étant très dépendante du domaine de l'entreprise, celle-ci devait présenter une souplesse importante. Le formalisme d'UML a donc été pensé indépendamment de toute méthode pour qu'il puisse être utilisé quel que soit le processus interne à l'entreprise.

1.2.2 Vue d'ensemble du formalisme UML

UML a été conçu pour pouvoir représenter les résultats des différentes phases du processus de développement ainsi que les différents aspects du système à développer. Pour atteindre ce but, la version 2.0 d'UML contient treize diagrammes dont la classification est exposée par la figure 1.1. Il existe trois catégories de diagrammes :

- six diagrammes qui représentent la structure du système (*Structure Diagram* en anglais) : les diagrammes de classes, d'objets, de composants, de structure composite, de paquetages, et de déploiement ;
- trois représentent le comportement du système (*Behavior Diagram* en anglais) : le diagramme des cas d'utilisation, d'activités et de machines à états ;
- enfin quatre diagrammes se concentrent sur les interactions entre constituants du système (*Interaction Diagram* en anglais) et sont donc des cas particuliers des diagrammes de comportement car les interactions entre constituants apportent certaines informations sur leurs comportements respectifs : le diagramme de séquence, de collaboration, de temps, et de vue d'ensemble des interactions.

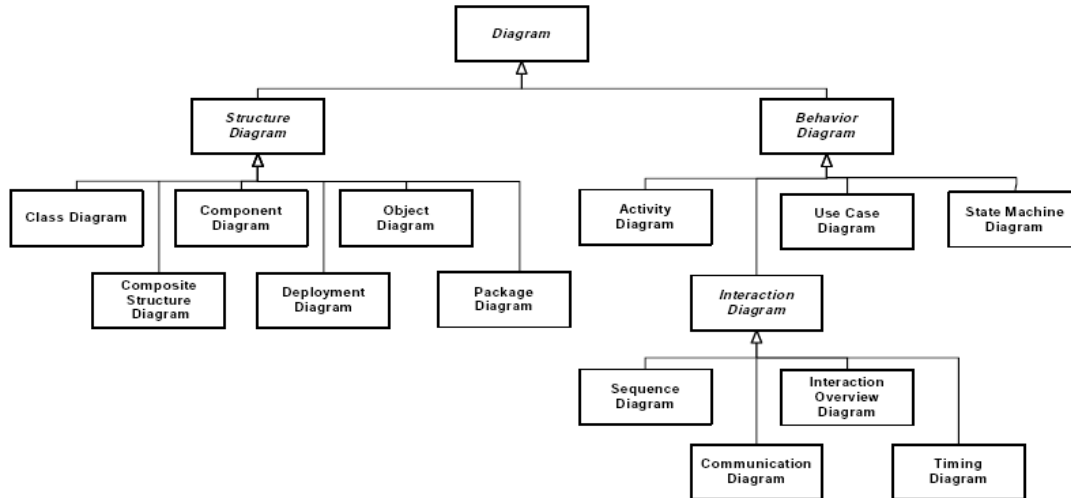


FIG. 1.1 – Classification des diagrammes d'UML 2.0

Les diagrammes de structure permettent principalement de classifier les différents objets de l'application, d'organiser ces objets en paquetages. Les notions d'interfaces, de composants et de collaborations sont également importantes. Le diagramme de déploiement permet quand à lui de décrire les ressources matérielles et logicielles disponibles et la répartition du système sur ces ressources.

Les diagrammes de comportement permettent de représenter les aspects dynamiques de l'application. Les machines à états permettent par exemple de spécifier le comportement des objets de l'application sous forme d'automates à états finis. Les diagrammes d'activités permettent de formaliser la séquence d'actions à exécuter. Ils se concentrent sur les conditions et la séquence de ces actions plutôt que sur les objets qui réalisent ces comportements. Dans la version 2.0 d'UML, les activités reprennent la sémantique à jetons des réseaux de Petri. Enfin, le diagramme des cas d'utilisation est un support pour exprimer les différentes fonctionnalités du système et est principalement utilisé dans les phases d'analyse du système.

Le but de ces treize diagrammes est de pouvoir représenter tous les aspects d'un système informatique durant les différentes phases du processus de développement. Cet objectif est cependant très (trop ?) ambitieux car il est extrêmement difficile de concevoir un langage capable de répondre aux besoins si divers des utilisateurs. C'est pourquoi, ces diagrammes sont parfois associés à des documents qui explicitent les choix ou introduisent de nouvelles informations qui ne sont pas contenues dans les différents digrammes. Par exemple les cas d'utilisation sont souvent associés à une description textuelle plus riche que le modèle UML. Conscient de cette limite, les personnes qui normalisent le langage UML y ont intégré la notion de profil qui permet d'étendre le langage UML avec les concepts voulus. Dans cette thèse cependant, nous considérons UML comme un point d'entrée et ne cherchons pas à le modifier bien que nous serons amenés à en exposer certaines limites.

1.2.3 Métamodélisation

Dans la suite du manuscrit, nous utilisons les concepts liés à la métamodélisation que nous introduisons donc ici.

UML est décrit grâce à une technique de métamodélisation en quatre couches que sont le niveau d'exécution (M0), le niveau modèle (M1), le niveau métamodèle ou langage (M2) et le niveau méta-métamodèle ou métalangage (M3) (cf. figure 1.2 inspirée de [56]).

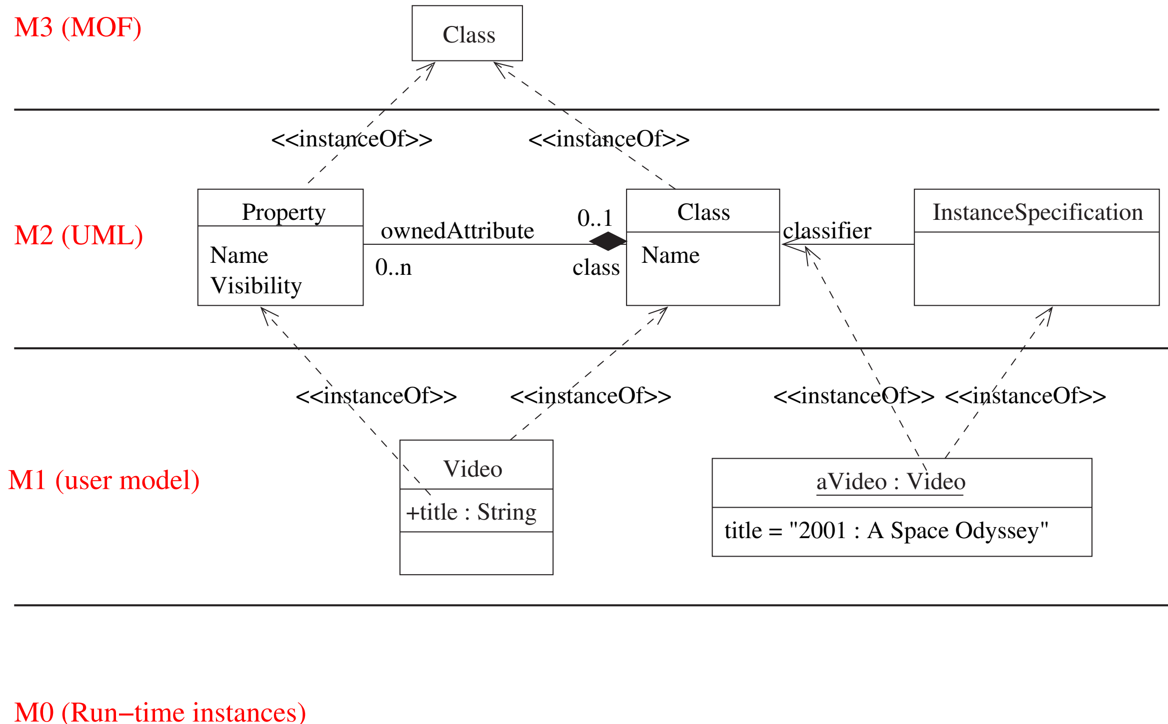


FIG. 1.2 – Exemple de la hiérarchie en 4 couches du métamodèle

Le niveau M0 correspond au système concret, c'est à cette couche que sont créés et détruits les objets exécutables.

La couche M1 est la couche où est modélisé le système.

La couche M2 définit le langage de modélisation utilisé à la couche M1, UML dans notre cas. Le but est de spécifier tous les concepts fournis par UML en considérant UML comme sujet de modélisation. Le métamodèle d'UML est décrit par le formalisme appelé MOF (*Meta-Object Facility* [56]) combiné à du texte en langage naturel. Le formalisme du MOF se rapproche du diagramme de classes UML. Il permet de définir et de hiérarchiser l'ensemble des éléments qui seront mis à la disposition des utilisateurs d'UML, les caractéristiques de ces éléments ainsi que la manière de les assembler. Par exemple, la figure 1.2 introduit les concepts de classes (`Class`) et d'attributs (`Property`) et leurs caractéristiques (`Name` pour les classes et `Name` et `Visibility` pour les attributs). Enfin ce métamodèle spécifie la relation qui peut exister entre ces deux éléments, à savoir qu'une classe peut contenir de 0 à n attributs et qu'un attribut peut être contenu par au plus une classe. Notons que les diagrammes dynamiques d'UML sont décrits de la même manière.

Enfin la couche M3 décrit le formalisme du MOF en MOF lui-même. Cette couche fournit un cadre commun pour définir les différents langages de modélisation. L'intérêt

est grand puisque la définition dans un même formalisme de plusieurs langages de modélisation permet par exemple de comparer ces langages ou de réaliser des passerelles entre ces langages.

D'autre part, le principe d'instanciation est primordial dans la métamodélisation. Celui-ci veut que chaque construction d'une couche est une instance d'une construction de la couche supérieure. Instancier un élément de niveau n consiste à donner des valeurs précises aux caractéristiques de cet élément pour obtenir un élément de niveau $n-1$. Sur l'exemple de la figure 1.2, instancier la classe `Video` en objet a consisté à donner une valeur définie à son attribut `title`. De même, l'instanciation de la métaclasse `property` est réalisée en donnant le nom particulier `title` au méta-attribut `Name` et la valeur `public` (représentée par le signe +) au méta-attribut `Visibility`. La figure 1.3 donne une vue graphique de cette instanciation. Le principe d'instanciation entre les couches M3 et M2 est similaire.

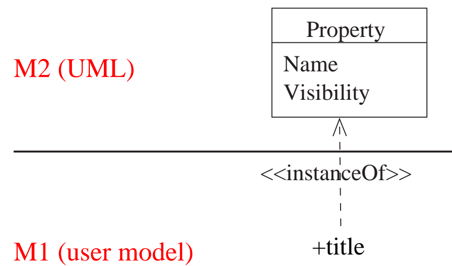


FIG. 1.3 – Représentation d'un attribut comme une instance de la métaclasse Property

Notons que le MOF permet de décrire formellement la structure que doit respecter un modèle UML. Par exemple, la figure 1.2 spécifie qu'une classe peut contenir plusieurs attributs, qu'une classe est nommée, etc. Par contre, les aspects comportementaux sont définis en langage naturel dans la spécification. C'est pourquoi UML est souvent qualifié de langage semi-formel.

1.3 Cohérence des modèles UML

L'objectif de cette section est d'introduire les notions d'incohérences et de gestion de ces incohérences. Nous considérons d'abord ces notions dans un cadre général (cf. partie 1.3.1) puis dans le cas d'UML (cf. partie 1.3.2).

1.3.1 Gestion des incohérences dans un processus de développement

Chacune des phases d'un processus de développement logiciel conduit à la productions de documents qui peuvent prendre des formes multiples. Ces documents sont corrélés car ils contiennent des informations redondantes ou complémentaires. Étant donné que de nombreux développeurs produisent et mettent à jour ces documents, ces informations peuvent être ou devenir incohérentes, car elles ne respectent pas les relations sémantiques qui lient ces documents.

Incohérence	*Informations non indépendantes qui transgressent une relation sémantique qui les lie*

Les relations peuvent être exprimées au moyen de règles de cohérence à partir desquelles la cohérence peut être vérifiée. En pratique, ces règles sont listées dans des documents, implantées dans un outil, ou non référencées du tout.

La plupart des approches considèrent ces incohérences comme indésirables, comme quelque chose à bannir autant que possible. D'une manière générale, nous pensons que lors du développement d'un système, il est important de pouvoir identifier les incohérences, d'y apporter un traitement adapté de manière à ce que celles-ci aient le moins d'impact sur la suite du développement.

[54] a un point de vue relativement similaire et propose un cadre de gestion de ces incohérences. L'auteur pense en effet qu'au cours du développement d'une application, l'apparition d'incohérences est inévitable lorsque, par exemple, de nombreuses personnes font évoluer les documents ou lors de l'évolution du système pour prendre en compte de nouveaux besoins. Il semble donc plus pertinent d'adopter des environnements qui autorisent la présence des incohérences et aident ensuite à les détecter plutôt que des environnements qui les interdisent. Dans le cas de la détection d'une incohérence, l'auteur propose trois alternatives, la résoudre, la tolérer mais la documenter, l'ignorer.

Afin d'étayer cette opinion, l'auteur considère l'exemple suivant, fruit d'une observation industrielle. La méthode de l'entreprise définissait que la spécification d'une fonction devait contenir une machine à états ainsi que des besoins exprimés en langage naturel. La machine à états était alors considérée comme une représentation graphique du texte. L'évolution des spécifications a fait diverger le texte et le diagramme. Le texte a été mis à jour mais pas le diagramme de machine à états pour des raisons de temps. Le concepteur a annoté cette divergence et signalé que le texte devait être considéré comme définitif. Cette incohérence n'a ainsi pas posé de problème lors de la phase de codage car le développeur était conscient qu'en cas de conflit d'informations entre le texte et la machine à états, il fallait donner la priorité au texte. Cet exemple montre que toute incohérence ne doit pas forcément être corrigée mais que le fait d'avoir identifié l'incohérence et de l'avoir traitée à réduit l'impact de cette incohérence sur le reste du processus.

1.3.2 La cohérence dans les modèles UML

Considérons maintenant le concept de cohérence pour le langage UML. L'incohérence d'un modèle UML est un cas particulier des incohérences introduites précédemment qui se limite aux constructions d'UML, d'où la définition :

Incohérence d'un modèle UML	*Un modèle UML est incohérent lorsque un ou plusieurs éléments d'UML y sont employés en transgressant les relations sémantiques qui lient ces éléments.*

La définition fait apparaître le terme « un ou plusieurs » car il est possible de produire une incohérence avec un seul élément d'UML. Par exemple, une transition d'un diagramme d'état doit être associée à exactement un état source et un état cible, l'emploi dans un diagramme d'état d'une transition isolée est donc incohérent.

Notons que l'incohérence d'un modèle UML que nous considérons est basée sur une violation de règles sur le langage UML et non pas sur l'application modélisée. Par exemple, on ne cherchera pas à détecter comme incohérent un modèle représentant une machine en fonctionnement alors que l'alimentation électrique est coupée. En effet la règle sémantique « seules les machines électriques alimentées peuvent fonctionner » est propre à l'application.

Comme dans le cas général, les relations sémantiques peuvent être exprimées au moyen de règles de cohérence. Le métamodèle est un moyen extrêmement pratique pour formaliser un grand nombre de ces règles de cohérence. Prenons l'exemple de la règle qui dit que « l'arbre des relations de généralisations (ou héritage) doit être direct et acyclique ». La figure 1.4 montre un exemple de modèle incohérent vis-à-vis de cette règle. Nous allons maintenant montrer comment le métamodèle permet la formalisation

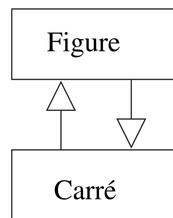


FIG. 1.4 – Un modèle incohérent

de règles de cohérence. Le modèle considéré contient deux types d'éléments, des classes et des généralisations. La figure 1.5 présente le sous-ensemble du métamodèle qui définit les relations existantes entre ces deux éléments. Notons que le concept de classe est une spécialisation du concept de classificateur (*classifier* en anglais) et donc que toute contrainte qui s'applique sur un classificateur s'applique également sur une classe. Dans

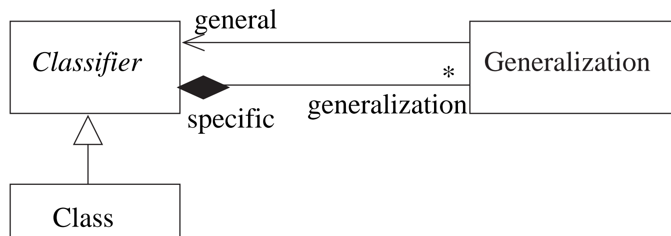


FIG. 1.5 – Sous-ensemble du métamodèle concernant les généralisation

la spécification d'UML [59] la règle est exprimée en OCL par :
`not(self.allParent->includes(self))` où `self` est un classificateur quelconque du modèle (une classe par exemple) et `self.allParent` contient l'ensemble des éléments dont hérite ce classificateur. Le calcul de cet ensemble est réalisé par navigation au travers des fins de méta-associations navigables `generalization` et `general`. Cette règle exprime donc qu'un classificateur ne peut pas hériter de lui-même. Cet exemple illustre qu'une règle de cohérence contraint la manière d'assembler un ensemble d'éléments (ici `Classifier` et `Generalization`). Cette contrainte est écrite au niveau langage et doit

être respectée par tous les modèles UML. Une incohérence est une instance particulière au niveau modèle de non respect de cette règle (comme le montre la figure 1.4).

En Génie Logiciel en général, la pluralité des formalismes rend la détection d'incohérences extrêmement difficile. Comme introduit auparavant, UML est un langage graphique multi-vues intégrant des formalismes différents qui permettent de décrire les différents aspects du système. Il est alors légitime de se demander comment les relations entre les différentes vues peuvent être établies. UML offre par exemple la possibilité de décrire le comportement des différents objets d'une classe par une machine à états. Cette relation est formalisée dans le métamodèle par la relation entre les métaclasses `Class` et `StateMachine` comme présenté par la figure 1.6. Dans sa thèse, [65] qualifie très justement le métamodèle de référentiel commun à tous les diagrammes. Il permet d'unifier les différents concepts et ce de manière indépendante des diagrammes dans lesquels ils peuvent être instanciés. Il n'est en effet pas rare que la description d'un diagramme particulier fasse intervenir des concepts introduits dans d'autres diagrammes.

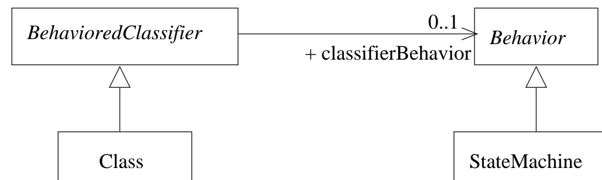


Fig. 1.6 – Exemple de relations entre concepts se trouvant dans des diagrammes différents

Au sein de la communauté UML, il est assez communément admis qu'une règle sémantique est exprimée en langage naturel et ne peut pas être vérifiée automatiquement. Au contraire, une règle syntaxique est définie formellement et peut être vérifiée automatiquement [35, 74]. C'est par exemple le cas pour la règle d'absence de cycle précédemment exprimée en OCL. Ainsi lorsqu'une règle exprimée en langage naturel est exprimée formellement elle passe du domaine sémantique au domaine syntaxique.

Notre point de vue consiste à considérer les règles de cohérence comme une expression de la sémantique de vérification, c'est-à-dire de la bonne manière d'utiliser les constructions d'UML. Pour illustrer cette sémantique, prenons l'exemple de l'affectation en langage ADA. La sémantique opérationnelle associée à l'instruction « `A :=B ;` » pourrait être : « on copie la valeur de l'expression B dans la variable A », alors que la sémantique de vérification serait : « le type de la variable A et le type de l'expression B doivent être identiques ». Respecter la sémantique de vérification des différentes constructions d'un langage est un prérequis à la bonne exécution de la sémantique opérationnelle de ces constructions. Dans la plupart des langages cependant, la sémantique de vérification est implicitement noyée dans la sémantique opérationnelle, voire absente du langage.

Le métamodèle est un moyen très pratique pour exprimer la plupart des règles de cohérence. Cependant, la spécification d'UML introduit les aspects comportementaux

d'UML en langage naturel exclusivement. La détection d'une incohérence faisant intervenir les aspects comportementaux d'UML ne peut donc pas s'appuyer exclusivement sur le métamodèle.

Pour conclure, soulignons que la potentialité d'incohérences dans les modèles UML n'est pas un défaut du langage ; il s'agit au contraire d'un révélateur de fautes de modélisations. Il convient cependant de maîtriser ces incohérences potentielles afin d'améliorer la correction des modèles.

1.4 Maîtrise de la cohérence des modèles UML par la gestion des risques

L'approche choisie au LESIA pour la maîtrise de la cohérence des modèles UML consiste à appliquer les différentes étapes de la gestion du risque aux *événements dommageables* que sont les incohérences pouvant affecter les modèles UML. Les travaux présentés dans cette thèse se situent dans ce cadre.

Dans un premier temps, nous justifierons l'approche de la gestion des incohérences par le processus de gestion des risques (cf. section 1.4.1) puis nous définirons les étapes considérées dans cette thèse (cf. section 1.4.2).

1.4.1 Risque d'incohérence

Les incohérences liées au langage UML sont associées aux constructions du langage UML. En effet, ces constructions peuvent être utilisées et assemblées de manière valide ou non. La présence d'incohérences est donc potentielle. Il apparaît donc légitime de penser qu'une incohérence peut être associée à un degré de vraisemblance. Ce degré pourra par exemple être corrélé à la complexité des constructions mises en jeu, à la complexité du modèle ou à la connaissance du langage UML par l'utilisateur.

Comme introduit dans la section 1.3.1, les incohérences ont un effet plus ou moins grave sur la suite du processus de développement. Cet effet peut par exemple être mesuré sur les conséquences engendrées sur la production de code ou sur la maintenance.

La notion de risque consiste à coupler le degré de vraisemblance d'un événement dommageable à la gravité du dommage engendré. Il nous est donc apparu adapté de gérer les incohérences d'un modèle UML par une approche de gestion du risque.

Les principales étapes de la gestion du risque sont [1] :
- l'**identification** des risques dont le but est de recenser et décrire l'ensemble des événements dommageables, un événement étant un ensemble particulier de circonstances, ainsi que ses conséquences ;
- l'**estimation** des risques qui attribut une valeur à la vraisemblance de l'événement dommageable et à la gravité du dommage ;
- l'**évaluation** des risques qui détermine l'importance relative de chacun des risques (hiérarchisation) ;
- le **traitement** des risques dont le but est généralement de mettre en œuvre des mesures conduisant à modifier la vraisemblance (prévention) ou la gravité d'un

risque (protection) de manière à le rendre acceptable ;
- l'**acceptation** des risques qui porte un jugement sur les risques en fixant souvent un seuil d'acceptabilité.

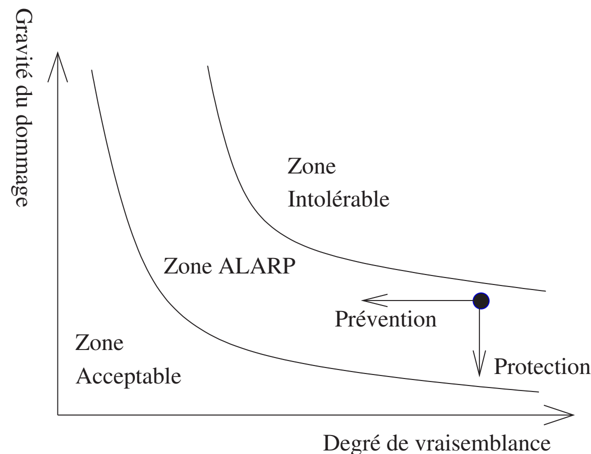


Fig. 1.7 – Couplage vraisemblance/dommage

Le but des trois premières phases de ce processus est de placer un risque déterminé sur le graphique fournit à la figure 1.7. La phase de traitement permet de prévenir le risque ce qui revient à diminuer le degré de vraisemblance du risque ou/et de se protéger du risque ce qui revient à diminuer la gravité du dommage associé à ce risque. Enfin l'acceptation du risque fixe les seuils d'acceptabilité du risque en définissant les zones où le risque est acceptable et inacceptable. La zone ALARP (*As Low As Reasonably Practicable*) est une zone dans laquelle le risque est réduit uniquement si le coût du traitement n'est pas trop élevé. C'est une notion qui fait intervenir des critères économiques dans l'acceptation du risque.

1.4.2 Vue d'ensemble des travaux

Cette section expose les activités menées au LESIA pour gérer le risque d'incohérence dans les modèles UML et situe mes travaux. Rappelons que l'événement dommageable considéré est la présence d'une incohérence.

La première partie de ma thèse a consisté à identifier les incohérences pouvant affecter les modèles UML [73]. Ce travail a été mené en collaboration avec Jean-Pierre Seuma Vidal.

La phase d'estimation et d'évaluation a consisté à fournir des valeurs sur la vraisemblance d'une incohérence et la gravité du dommage qu'elle peut engendrer. Au moment où cette thèse est écrite, deux méthodes ont été utilisées dans le laboratoire, par revue manuelle de modèles d'une part et par interview d'experts d'autre part [79, 78]. Dans les deux cas, le document identifiant les incohérences a servi de support. Ces travaux sont prolongés par Roberto Lopez-Toro qui établit des métriques basées sur le métamodèle.

Enfin, la phase de traitement est abordée sous plusieurs angles. D'une part, des guides de modélisation ont été rédigés pour prévenir les incohérences ou pour faciliter leur détection, notamment par l'ajout d'information augmentant la redondance et donc la capacité de détection. Ces travaux sont principalement menés par Jean-Pierre Seuma

Vidal. D'autre part, la détection automatique des incohérences est un bon moyen pour en réduire le risque en permettant par exemple de résoudre l'incohérence ou d'y apporter un traitement adapté afin d'en diminuer la gravité. La deuxième partie de mon travail de thèse a considéré ce thème [49].

Cette thèse a ainsi pour but de répondre à deux étapes de la gestion des risques associés à la cohérence des modèles UML : l'identification des incohérences et la protection par la détection automatique de ces incohérences.

La phase d'identification est détaillée au chapitre 2. Le chapitre 3 propose un état de l'art des méthodes existantes de détection des incohérences. Par ailleurs, il donne une vue d'ensemble de l'approche mise en œuvre dans ma thèse. Les chapitres 4 et 5 détaillent cette approche pour la vérification de la cohérence structurelle et pour la vérification de la cohérence comportementale. Enfin, le chapitre 6 décrit les résultats expérimentaux obtenus sur un modèle industriel.

Chapitre 2

Identification des règles de cohérence

La première étape dans la gestion du risque consiste à identifier l'ensemble des événements dommageables que sont les incohérences dans notre cas. Pour cela nous devons définir ce qu'est une incohérence dans un modèle UML. La section 2.1 fournit un état de l'art concernant la définition de la cohérence des modèles UML. La section 2.2 décrit l'approche que nous suivons pour définir la cohérence des modèles UML. Enfin, la section 2.3 illustre certaines incohérences identifiées.

2.1 Types d'approches

La cohérence des modèles UML est le thème de nombreux travaux. Dans cette section nous étudions les deux approches employées dans la bibliographie pour définir la cohérence des modèles UML. La première spécifie la cohérence au moyen de règles de cohérence (cf. section 2.1.1) alors que dans la deuxième approche, la cohérence est définie au travers d'une transformation vers un langage formel (cf. section 2.1.2).

Cette partie aborde les moyens de définir la cohérence des modèles UML sans aborder les techniques de vérification sous-jacentes qui seront examinées au chapitre 3.

2.1.1 Définition par règles

L'approche par règles considère qu'un modèle est cohérent s'il respecte un ensemble de règles de cohérence. Comme introduit en section 1.3.2, les règles de cohérence expriment la sémantique de vérification des différentes constructions du langage. Cette sémantique est induite par trois aspects différents (cf. figure 2.1 inspirée de [74]).

Règles induites par la sémantique du langage UML Les constructions du langage UML incorporent une sémantique, c'est-à-dire qu'elles ont une signification particulière lorsqu'elles sont utilisées dans un modèle. Cette sémantique est complétée par une sémantique de vérification qu'il est possible d'exprimer par des règles de cohérence. Ces règles sont souvent appelées règles de bonne formation (*well-formedness rules* en anglais). Elles expriment l'usage correct des constructions.

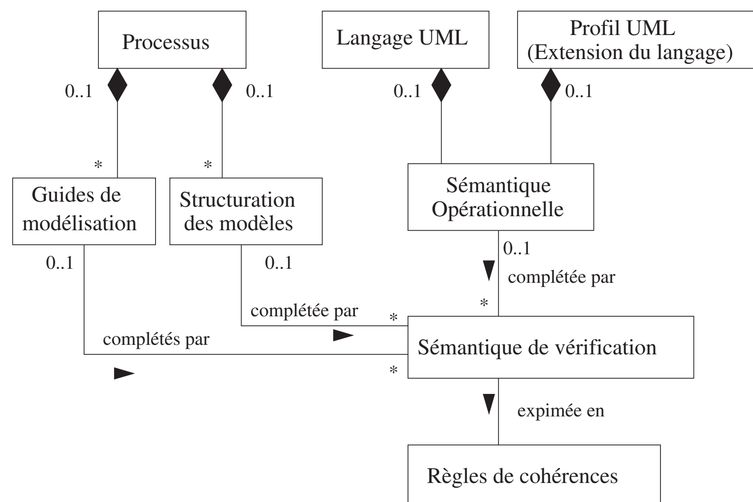


Fig. 2.1 – Les différents types d'incohérence

Prenons l'exemple de la notion de message présente dans les diagrammes d'interactions. D'une part, la structure d'une classe permet de représenter les différentes opérations et les différents signaux supportés par une classe. D'autre part, le diagramme d'interactions permet de représenter des échanges de messages entre classes. La norme UML [59] formule ainsi la sémantique du concept de message : « *(...) A Message reflects either an Operation call and start of execution - or a sending and reception of a Signal (...)* » [1]. Cette sémantique implique la règle suivante « *The signature [of a message] must either refer to an Operation (...) or a Signal (...)* »[2]. Cette règle est également tirée de la spécification d'UML.

Règles induites par une extension d'UML (Profil UML) La définition de profils UML permet d'étendre le formalisme UML avec de nouveaux concepts. À l'instar des concepts déjà intégrés à UML, ces extensions ont une sémantique opérationnelle qui implique une sémantique de vérification. Par exemple [52] est un profil ajoutant à UML des concepts liés au temps-réel. Le concept de durée y est décrit comme un intervalle défini par un instant de début et un instant de fin. Une durée étant une grandeur physique positive, on peut en déduire la règle suivante : « l'instant de début doit être antérieur à l'instant de fin ». C'est aussi l'approche suivie dans [61] qui étend le métamodèle UML pour qu'il prenne en compte des propriétés non fonctionnelles relatives à la qualité de service comme le temps d'exécution ou la précision du résultat. Les exigences des différents composants en terme de temps de calcul doivent alors être cohérentes entre elles.

Règles induites par le processus Un processus de développement définit les différents documents à produire pour obtenir une réalisation du système en partir des spécifications. Au fur et à mesure de l'avancement du projet et des itérations, des modèles de moins en moins abstraits seront ainsi établis. Ces modèles représentent le même système

[1]Un message est soit le reflet d'un appel d'opération et du début d'une exécution, soit l'envoi et la réception d'un signal.

[2]La signature [d'un message] doit référencer soit une opération, soit un signal.

à des niveaux d'abstraction différents et sont donc liés par une relation de raffinement. Cela pose le problème de la cohérence entre les différents modèles qui est qualifiée de cohérence inter-modèle [37]. A contrario, la cohérence qui considère un seul modèle est dite intra-modèle.

Par exemple, [64] transpose des patrons de raffinement (*refinement patterns* en anglais) développés pour le langage Object-Z au langage UML. Ces patrons de raffinement peuvent ensuite être formalisés et ainsi vérifiés automatiquement. Le non-respect de ces patrons crée des incohérences inter-modèles. [16] vérifie des contraintes dynamiques et adaptées pour le processus *KobrA* ; ce processus propose par exemple de spécifier le comportement d'un composant par une machine à états et de décrire la manière dont est réalisé ce comportement par un diagramme d'activité et d'interaction ce qui donne lieu à la contrainte suivante « *the behavior specified in state-charts in the Komponent specification must be consistent with the behavior specified in activity/interaction diagrams in the Komponent realization* »[3].

D'autre part, le processus (ou la méthode) définit éventuellement des guides de modélisation internes à l'entreprise afin de respecter certains standards, d'orienter les modeleurs, etc. Par exemple, [4] fournit une série de guides qui doivent être respectés lors du développement d'applications chez Siemens tel que : « *Use an activity diagram to show all possible scenarios associated with a use case* »[4]. Pour qu'un modèle soit cohérent vis-à-vis d'un processus il faut que ces guides soient respectés.

Cette approche est intéressante dans la mesure où l'ensemble des règles de cohérence est disponible. En effet, vérifier la cohérence d'un modèle UML vis-à-vis de listes incomplètes ne permet pas d'assurer la cohérence globale du modèle.

2.1.2 Définition par transformation

Certains travaux définissent la cohérence des modèles UML par transformation dans un langage formel. Le modèle UML est considéré cohérent si sa représentation dans le langage formel respecte certaines propriétés propres à ce langage.

Par exemple, [67] donne une représentation d'un diagramme de classe et de son diagramme de machine à états associé respectivement en Object-Z et en algèbre de processus CSP. Les auteurs considèrent alors qu'un modèle UML est cohérent si la règle « *A specification consisting of an Object-Z class and an associated state machine has the property of basic consistency iff the corresponding process in the semantic model is deadlock free* » [5] est respectée. [48] propose de transformer un sous-ensemble d'UML dans le langage formel OOL (Object-Oriented specification Language). Les auteurs exploitent ensuite la définition de la cohérence et de la notion de raffinement précisée pour le langage OOL afin de transposer et vérifier ces propriétés sur les modèles UML ou plus exactement sur la spécification OOL déduite des modèles UML.

[3] Le comportement spécifié par la machine à états de la spécification d'un Komponent doit être cohérent avec le comportement spécifié par les diagrammes d'activités et d'interactions de la réalisation du Komponent

[4] Utiliser un diagramme d'activité pour montrer tous les scénarios possibles d'un cas d'utilisation

[5] La spécification consistant en une classe d'Object-Z et en une machine à états associée est simplement cohérente ssi les processus du modèle sémantique ne peuvent pas s'inter-bloquer.

Notons que la définition de la cohérence des modèles UML par des règles ou par transformation n'est pas exclusive. En effet, vérifier une propriété sur le modèle formel revient dans certains cas à vérifier une règle sur le modèle UML.

Cependant, le principal inconvénient de l'approche transformationnelle est que la cohérence du modèle UML n'est pas définie dans son intégralité [38]. Seule est prise en considération la cohérence mettant en jeu des constructions ayant une représentation dans le langage formel. De plus, seules les propriétés évaluables sur les modèles formels sont vérifiables.

2.1.3 Approche proposée

Cette section expose et justifie l'approche choisie pour définir la cohérence des modèles UML.

Comme défini auparavant, deux approches peuvent être employées pour définir la cohérence, une approche par règles et une approche transformationnelle. Nous avons opté pour une approche par règles car notre objectif est de traiter l'ensemble des incohérences possibles. Or comme mentionné au préalable, l'approche transformationnelle n'assure pas la cohérence de l'ensemble des constructions d'UML mais uniquement la cohérence des constructions d'UML qui ont une image dans le langage formel.

Rappelons que les règles de cohérences sont induites par trois sources, la sémantique du langage UML, la sémantique d'éventuelles extensions et enfin par le processus de développement. Comme déjà expliqué lors de l'introduction, UML est considéré dans cette thèse comme un point d'entrée. Nous n'avons en effet ni défini ni étudié de profils particuliers d'UML. Nous ne nous intéresserons donc pas aux règles impliquées par la sémantique de constructions étendant UML. Concernant le processus de développement, nous adoptons la même vision que les concepteurs d'UML. Nous pensons en effet qu'il est extrêmement difficile de découpler l'étude d'un processus indépendamment du domaine d'application ou de l'entreprise réalisatrice. La notion de processus n'est pas considérée dans notre étude, nous nous situons donc dans l'approche intra-modèle. Concernant les règles induites par la sémantique d'UML, nous pensons au contraire que nous avons tous les éléments pour les définir. La sémantique d'UML est en effet décrite dans sa spécification [59]. Cette affirmation doit cependant être modérée. UML pouvant être utilisé dans plusieurs domaines, sa sémantique est parfois laissée volontairement floue.

D'autre part, à l'instar de [41], nous pensons que « *it is necessary to define what should be checked prior to discussing how to perform this actual verification* »[6]. L'étude bibliographique révèle qu'aucun document dont l'objectif est de réunir l'ensemble des règles de cohérence n'est disponible. Notre objectif est de produire un tel document. D'autre part, coupler l'étude de la cohérence des modèles UML à un moyen de vérification donné entraîne inévitablement une restriction sur la définition de cette cohérence : seule les règles qui pourront être supportées par la méthode employée sont prises en

[6] Il est nécessaire de définir ce qui doit être vérifié avant de penser à la méthode de vérification

compte. Pour éviter cet écueil, nous avons exprimé les règles de cohérence en langage naturel (français).

2.2 Vue d'ensemble du résultat

L'élaboration d'un document recensant l'ensemble des règles de cohérence nécessite l'adoption d'une méthode rigoureuse.

Deux personnes (Jean-Pierre Seuma Vidal et moi-même) ont d'abord réalisé une analyse indépendante et en parallèle de la spécification d'UML [59]. Ces analyses avaient pour but d'identifier les contraintes explicites ou implicites pour chacune des constructions d'UML. Les résultats ont ensuite été discutés afin d'être mis en commun.

Une étude bibliographique a également été réalisée afin d'intégrer les règles de cohérence mentionnées dans les différents articles. Les résultats du groupe de travail « *Consistency Problems in UML-based Software Development* » [44, 43, 9] ont représenté un apport important.

La première partie de ma thèse a donné lieu à un document recensant l'ensemble des règles de cohérence identifiées [50]. Ce document de plus de 150 pages est disponible à l'adresse `http ://www.lesia.insa-toulouse.fr/UML`. Cette section décrit le document produit, du point de vue de son organisation (cf. section 2.2.1) et fournit des résultats quantitatifs (cf. section 2.2.2).

2.2.1 Organisation du document

La structure du document est basée sur la classification proposée par [37]. La figure 2.2 qui en est tirée présente une classification des types d'incohérences. Dans cette classification, nos travaux se situent dans les couches 2 et 3, c'est-à-dire que nous ne traitons que des incohérences sur les éléments et les relations entre ces éléments (niveau 3) et sur les diagrammes et les relations entre diagrammes (niveau 2).

Ceci implique :

- une étude de la cohérence de chaque élément d'UML et de chaque relation entre ces éléments, ceci constitue le niveau conceptuel diagramme ;
- une étude de la cohérence de chacun des diagrammes UML et des relations entre diagrammes ce qui constitue le niveau conceptuel modèle.

Le document produit contient une partie pour chaque type de diagramme d'UML 2.0 (diagramme de classes, de déploiements, d'interactions, etc.).

Chaque partie est composée de trois chapitres qui correspondent respectivement aux règles associées aux éléments du diagramme, aux relations du diagramme et aux règles globales à un diagramme.

Chacun de ces chapitres est ensuite découpé en sections qui correspondent à un élément, une relation ou un diagramme particulier. Dans chacune de ces sections, le même canevas d'étude est adopté :

- en premier lieu, nous présentons le « **contexte** », c'est-à-dire les circonstances dans lesquelles une incohérence peut être exprimée lors de la modélisation UML ;
- ensuite, nous définissons les « **règles de cohérence** » (*consistency rules* en anglais) qui expriment par intention la sémantique de vérification que l'élément doit

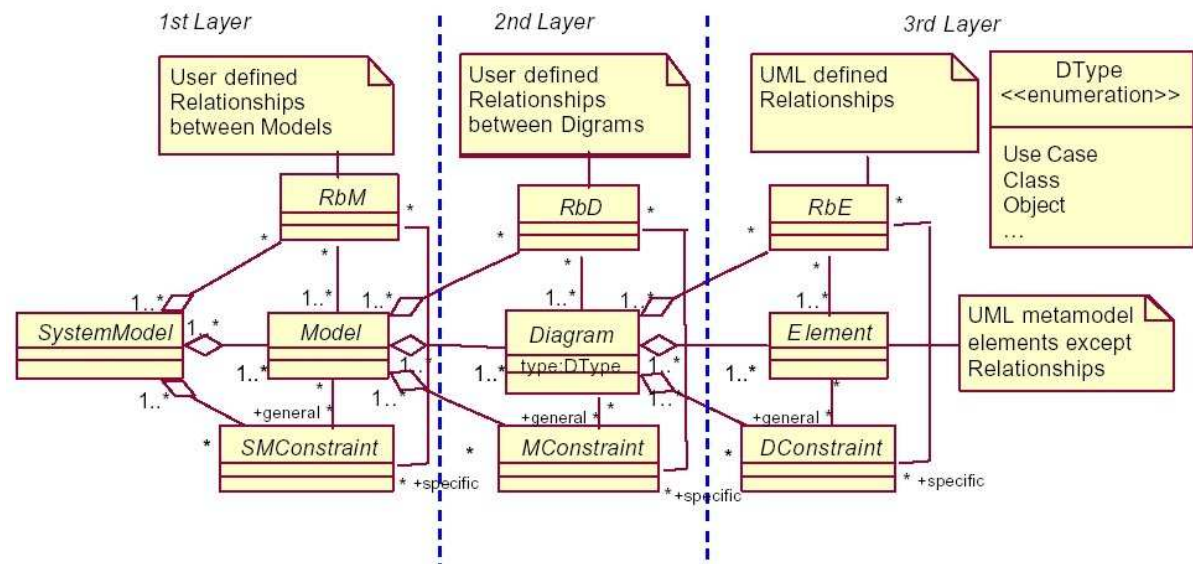


Fig. 2.2 – Types d'incohérences

respecter ;

– lorsque cela est possible, des « **guides** » de style pour faciliter la détection des erreurs sont également fournis ;
– la rubrique « **justification** » apporte une justification pour chaque guide, les guides et les justifications sont numérotés ce qui permet de faire le lien entre un guide et sa justification.

2.2.2 Résultats quantitatifs

Durant l'identification des règles, nous avons trouvé opportun d'associer deux types d'informations à chaque règle de cohérence. Les résultats quantitatifs sont fournis dans cette section en fonction de ces types d'information.

La première information correspond à l'origine de la règle. Quatre sources sont différenciées :

1. Les règles fournies explicitement dans la spécification d'UML 2.0 et directement exprimées en tant que contraintes, par exemple : *"Un arc d'activité doit être contenu uniquement par des activités ou des groupes"*.
2. Les règles déduites du métamodèle UML. Le métamodèle spécifie chaque caractéristique en utilisant UML lui-même [59]. Ces règles expriment de façon textuelle des contraintes incluses dans le métamodèle : des contraintes de multiplicité, des contraintes sur les types des éléments mis en jeu par une relation par exemple. Spécifier ce type de règles n'ajoute aucune sémantique de vérification au langage UML puisque ces contraintes sont contenues dans le métamodèle. Cependant, ces règles permettent de clarifier certains points pour les utilisateurs d'UML qui ne connaissent pas forcément le métamodèle, par exemple : *"Un arc d'activité a exactement un nœud d'activité source et un nœud d'activité cible"*.

3. Les règles extraites de la littérature. Ces règles se trouvent soit dans la spécification mais ne sont pas exprimées en tant que contraintes, soit dans des articles ou des livres, par exemple : *"Tout connecteur avec un label donné doit être apparié avec exactement un autre connecteur qui possède le même label et qui se trouve dans le même diagramme d'activité"* [59].
4. Enfin, les règles qui sont le résultat de notre analyse de la spécification et de la sémantique d'UML. Ces règles ont pour but de définir l'ensemble des interprétations licites d'un modèle [75]. Ce sont de nouvelles règles ; elles représentent la contribution majeure de notre travail, par exemple : *"Seuls les arcs sortants d'un nœud de décision ou d'un nœud de bifurcation (fork en anglais) peuvent avoir des conditions de garde différentes de `true`"*.

Le tableau 2.1 présente l'origine des 650 règles exprimées dans notre document [50].

Origine	Nombre de règles	Pourcentage
Contraintes de la spécification	290	44.62
Déduites du métamodèle d'UML 2.0	56	8.61
Provenant de la littérature	44	6.77
Nouvelles règles	260	40
TOTAL	650	100

Tab. 2.1 – Résultat synthétique de l'origine des règles

La deuxième information associée à chaque règle est la différenciation des règles selon leur niveau d'écriture et d'application. Le métamodèle décrit d'une part les caractéristiques et la sémantique de chaque élément du langage mais aussi la représentation graphique de chacun de ces éléments. La correspondance n'est cependant pas totale vu que certaines notations graphiques n'ont pas de représentation dans le métamodèle et certains éléments d'UML n'ont pas de représentation graphique. Il existe donc trois niveaux d'écriture et d'application des règles. Le premier niveau est constitué des règles pouvant être écrites au niveau « méta » et qui contraignent le métamodèle. Ces règles sont utiles aux concepteurs d'outils mais sont transparentes pour l'utilisateur. Ces règles seront qualifiées de « méta ». Le deuxième niveau correspond aux règles pouvant être écrites au niveau méta et qui contraignent les modèles. Ces règles sont les plus intéressantes et les plus nombreuses. Le troisième niveau correspond aux règles ne pouvant pas être écrites au niveau « méta ». Ceci s'explique par le fait qu'aucune métaclasse ne correspond à l'élément graphique sur lequel s'applique la règle. Ces règles sont appelées « règles utilisateur ». Le tableau 2.2 montre la répartition des règles par niveau d'abstraction.

2.3 Exemples de règles de cohérence

Cette partie permet d'illustrer notre travail. Le nombre de règles de cohérence étant conséquent nous essayons de fournir des exemples représentatifs de chaque type de règles.

Origine	Nombre de règles	Pourcentage
Règles méta	47	7.23
Règles utilisateur	26	4
Règles à la fois « méta » et « utilisateur »	577	88.77
TOTAL	650	100

Tab. 2.2 – Répartition des règles par niveau d'abstraction

Nous présentons une règle déduite du métamodèle à la section 2.3.1, deux exemples de nouvelles règles à la section 2.3.2 et en enfin un exemple de règle tirée de la norme à la section 2.3.3.

Comme expliqué en section 2.2.2, nous associons à chaque règle un niveau d'écriture et d'application. L'exemple 3 de la section 2.3.2 est une règle « utilisateur ». La règle présentée en section 2.3.3 est une règle « méta ». Les autres règles sont des règles qui peuvent s'écrire au niveau métamodèle et qui contraignent les modèles.

2.3.1 Règle déduite du métamodèle

Considérons le sous-ensemble du métamodèle présenté à la figure 2.3 et tiré de [59]. Cette partie du métamodèle implique la règle de cohérence suivante : *"Tout élément inclus dans un paquetage doit être un élément empaquetable."*

La spécification d'UML ne fournit pas explicitement la liste de tous les éléments empaquetables. Afin de faciliter l'identification des incohérences associées, nous avons reconstitué l'arbre de spécialisation (voir figure 2.4) qui fournit la liste des éléments empaquetables. Notons que dans notre document, l'arbre de spécialisation de `Classifier` et celui de `Dependency` sont également détaillés.

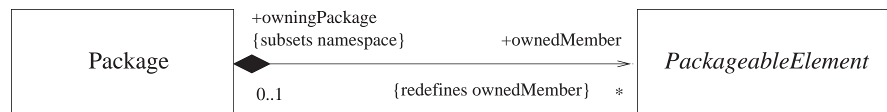


Fig. 2.3 – Relation entre les métaclasses `Package` et `PackageableElement`

Cette règle est déjà spécifiée par le métamodèle. Son expression est cependant intéressante pour deux raisons. Premièrement, certains outils ne suivent pas totalement la norme d'UML et permettent donc de créer des modèles qui ne respectent pas son métamodèle. Or, satisfaire ces règles de cohérence permet de s'assurer du respect du métamodèle ce qui est primordial pour l'interopérabilité des outils, la transformation vers d'autres formalismes ou vers des évolutions d'UML, etc. Enfin, donner l'arbre de spécialisation des éléments empaquetables explicite les éléments pouvant être contenus dans un paquetage ce qui facilite la définition de la cohérence.

2.3.2 Nouvelle règle de cohérence

Cette partie présente trois exemples de nouvelle règle de cohérence.

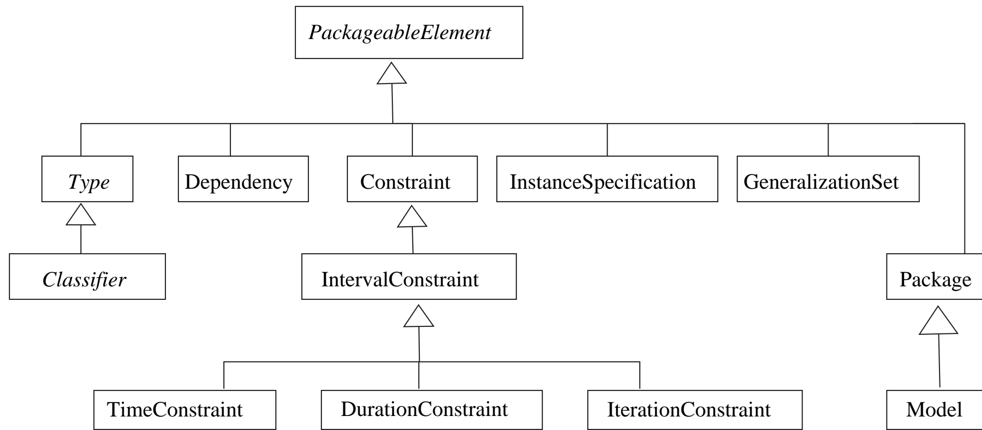


Fig. 2.4 – Arbre de spécialisation de la classe `PackageableElement`

Exemple 1 Dans cette section, nous donnons un exemple sur l'élément du métamodèle `Property`. Il est important de différencier l'élément du métamodèle `Property` de la notion générale de propriété. L'élément du métamodèle `property` représente soit un attribut d'une classe, soit une fin d'association. Nous traduisons donc `property` par possession.

Le modèle de la figure 2.5 présente une incohérence qui aurait pu être levée en respectant la règle suivante : *"La somme des bornes inférieures des multiplicités des possessions qui sont le sous-ensemble d'une possession (nommée A) doit être inférieure à la borne supérieure de la multiplicité de A."*

Ici, le problème de la modélisation provient du couplage entre les multiplicités de la fin d'association jouant le rôle `b` et des fins d'associations qui sont un sous-ensemble de ce rôle. En effet, il y aura au maximum 5 objets instances de FB reliés à FA, et les multiplicités indiquent qu'il faudrait au moins 3 objets instances de SB1 et 3 objets instances de SB2, soit au total 6 instances de FB. En fait, il est impossible d'instancier un modèle qui enfreint cette règle de cohérence sans violer les contraintes imposées par les multiplicités.

Notons que cette règle peut être formalisée en s'appuyant sur le métamodèle et contraint le niveau modèle. Une faute de modélisation violant cette règle est difficile à détecter car elle demande de coupler des informations dispersées dans le modèle. Et ce, d'autant plus que prises séparément, les multiplicités 3..4 des fins d'associations de `asso2` et `asso3` respectent bien la propriété {subsets b} comme l'exprime la contrainte de la spécification d'UML : « *A subsetting property may strengthen the type of the subsetted property, and its upper bound may be less* »[7].

Exemple 2 Les diagrammes d'activités UML 2.0 ont été largement inspirés par les réseaux de Petri. Il a donc semblé naturel de définir la règle : *" Tout arc d'un diagramme d'activité doit être tirable".*

Un réseau de Petri est quasi-vivant si pour toute transition, il existe un marquage

[7]Le type de la possession sous-ensemble doit être conforme au type de la possession sur-ensemble et la borne supérieure de la possession sous-ensemble doit être inférieure à la borne supérieure de la possession sur-ensemble.

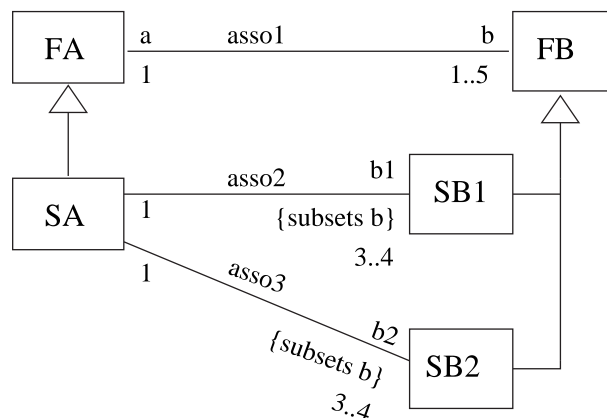


FIG. 2.5 – Incohérence sur la définition de multiplicités dans un diagramme de classes

accessible qui active cette transition.

La figure 2.6 présente une activité qui ne respecte pas cette règle. En effet, pour que le nœud d'union soit traversé et l'arc 4 tiré, il faut que les arcs 1, 2 et 3 offrent un jeton simultanément. Or, les arcs 1 et 2 ne pourront jamais offrir un jeton simultanément car un nœud de décision les précède.

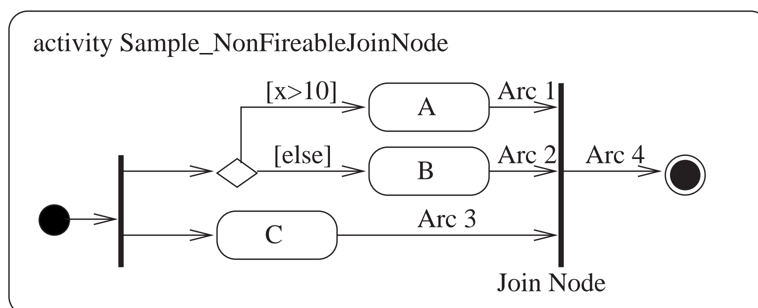


FIG. 2.6 – Incohérence d'un diagramme d'activité

Cette incohérence montre également la diversité des types d'incohérence qui ont été exprimés. Cette règle est comportementale car elle fait intervenir la définition de la dynamique des constructions d'UML. Elle a quand même sa place dans notre document car nous ne faisons aucune hypothèse sur les techniques de détection. Un des enjeux soulevés par notre travail sera d'ailleurs la définition d'un nombre restreint de langages (voire d'un seul langage) qui permettent de détecter des incohérences aussi diverses, et d'avoir un bon taux de couverture des règles.

Exemple 3 L'exemple présenté ci-dessous est une règle de cohérence dite « utilisateur », c'est-à-dire qui ne peut pas être écrite au niveau métamodèle. Ce type de règle existe car certains éléments graphiques d'UML n'ont pas de correspondance dans le métamodèle. Ces règles concernent donc des détails graphiques introduits dans la norme qui ne sont pas le reflet de concepts primordiaux.

La norme d'UML indique la possibilité d'utiliser des points de suspension `(...)` lorsque la liste d'attributs par exemple n'est pas complète : « *An ellipsis (...) as the*

final element of a list of features indicates that additional features exist but are not shown in that list ».[8]

Ceci implique la règle suivante : « *L'emploi de points de suspension dans une liste d'attributs ou d'opérations suppose que des attributs ou des opérations non explicités dans cette vue sont présents dans la classe. D'autres vues du modèle doivent donc les faire apparaître pour que le modèle soit cohérent* ».

2.3.3 Règle tirée de la spécification d'UML

Nous fournissons enfin une règle tirée de la spécification d'UML.

Le diagramme de machine à états fourni par UML permet de décrire le comportement discret d'entités du système. Cette description est hiérarchique puisqu'il est possible de réaliser des états composites qui sont des états qui contiennent d'autres états. La figure 2.7 donne un exemple de spécification du comportement d'un ATM (inspirée de [58]). L'état `ReadAmount` est un état composite car son comportement est spécifié à l'aide de sous-états. Les autres états sont des états simples.

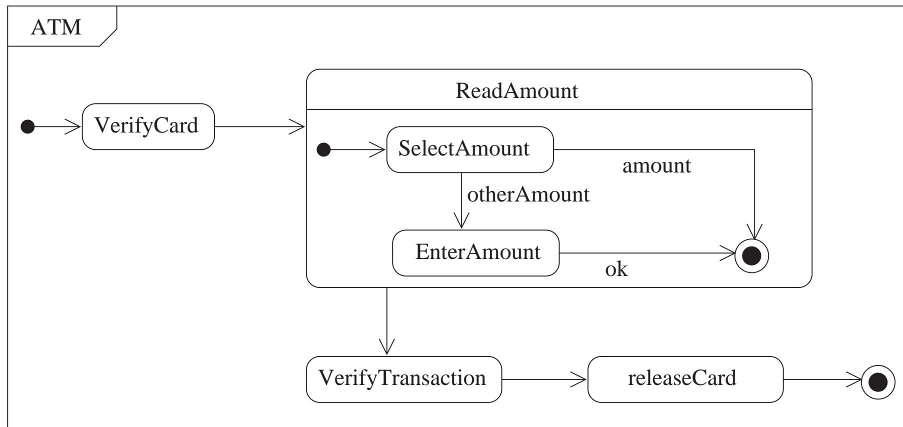


Fig. 2.7 – Exemple de machine à états avec état composite

Considérons maintenant la partie du métamodèle qui définit les différents éléments utilisés par ce modèle et qui est présenté par la figure 2.8. Ce métamodèle spécifie qu'une machine à états contient au moins une région qui elle-même peut contenir des états ou des pseudo-états (un état initial par exemple). Il existe deux types d'états, les états simples ou composites. Le type de ces états est marqué par les méta-attributs `isSimple` et `isComposite`. Ces méta-attributs sont dérivés (signe `/`), c'est-à-dire que leur valeur peut être calculée à partir d'autres informations contenues dans le modèle. La spécification d'UML décrit la manière de calculer ces méta-attributs par les règles suivantes :

- « *A simple state is a state without any regions* »[9]
- « *A composite state is a state with at least one region.* »[10]

[8] Des points de suspension (`...`) utilisés en dernier élément d'une liste de caractéristiques [par exemple des attributs ou des opérations d'une classe] indiquent que d'autres caractéristiques non représentées existent.

[9] Un état simple ne contient pas de région.

[10] Un état composite contient au moins une région.

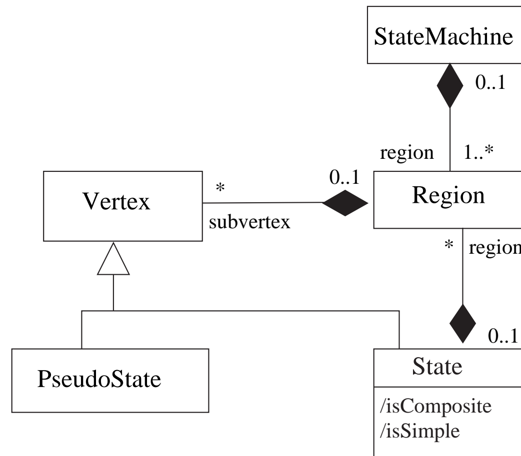


Fig. 2.8 – Métamodèle définissant les relations entre machine à états, états et sous-états

Ces règles sont recensées dans notre document et sont qualifiées de « méta » car elles sont transparentes pour l'utilisateur. Le modeleur n'a en effet pas les moyens de spécifier la valeur des méta-attributs. Le respect de ces règles revient à la charge de l'outil. Si le modeleur a spécifié un état qui ne contient pas de région, on doit avoir $(isComposite, isSimple) = (false, true)$ et dans le cas contraire $(isComposite, isSimple) = (true, false)$.

2.4 Discussion

Cette partie fournit des compléments d'information sur le travail présenté dans le chapitre. Nous présentons les principales limites de notre étude en section 2.4.1. Nous considérons ensuite l'utilisation des règles de cohérences identifiées pour évaluer certains outils UML en section 2.4.2 puis nous concluons.

2.4.1 Limites

Nous détaillons ici les trois principales limites de notre étude.

Complétude et cohérence du document L'ensemble des règles de cohérence obtenu est conséquent (650 règles). On peut cependant supposer que certaines règles n'ont pas été identifiées. Il n'existe en tout cas aucun moyen théorique d'affirmer que l'exhaustivité a été atteinte. Le seul moyen de tendre vers l'exhaustivité serait d'avoir des retours extérieurs sur les règles produites. Certains retours ont déjà eu lieu lors de l'interview d'experts.

D'autre part, malgré un effort important de relecture, le nombre considérable de règles amène à un volume important d'information qu'il est dur de maîtriser. Il est donc possible que certaines règles soient redondantes mais aussi contradictoires ce qui serait extrêmement gênant. Notons que certains travaux dont [74] s'intéressent à la cohérence des règles entre-elles.

Moyen d'expression des règles Les règles de cohérence ont été exprimées en langage naturel (français) afin de ne pas être limité par un langage formel donné.

Il est évident que ces règles devront être traduites en anglais si l'on veut les faire remonter à l'OMG ou si l'on veut que l'ensemble de la communauté UML y ait accès.

La revue manuelle de modèles à l'aide de ces règles [79] est un travail difficile. Cette difficulté est d'autant plus importante que les modèles sont grands. Or, l'intérêt des techniques de modélisation est justement de gérer la complexité croissante des systèmes informatiques. Dans les chapitres suivants nous nous intéressons donc à des moyens de détection automatiques de ces incohérences.

Évolutions Les évolutions de la spécification d'UML sont relativement rapides puisque depuis 1996 on peut compter au moins cinq évolutions importantes du langage (0.9, 1.0, 1.1, 1.2, 1.4 et 2.0). On peut tout de même espérer qu'UML atteint une maturité suffisante et que les évolutions prochaines ne concerneront pas les concepts fondamentaux. La principale partie de notre étude a été réalisée sur une version temporaire de la spécification 2.0 [58]. Aussi nous n'avons aucun moyen souple pour faire évoluer les règles en même temps que le langage lui-même.

2.4.2 Évaluation d'outils

La détection d'incohérences est une activité fastidieuse lorsqu'elle est réalisée manuellement. C'est une activité qui doit être automatisée au maximum. La capacité des outils à détecter des incohérences est donc un critère d'évaluation important. Notre document peut servir de base pour créer des modèles erronés, c'est-à-dire qui ne respectent pas certaines règles de cohérence. Ces modèles peuvent alors servir de benchmark pour tester de manière équivalente la capacité des outils à détecter les incohérences. Une telle étude a été réalisée.

Nous avons testé la capacité de trois outils industriels : Visual Paradigm (VP, Professional Edition version 3.1, Mars 2004), Rational Rose Enterprise Edition (IBM version Juillet 2004), Ameos (Aonix version Juillet 2004). L'évaluation ne concerne que les diagrammes de classes, de machines à états, et d'interactions.

Chaque incohérence est associée à une des notations suivantes :

- *Détectée* si l'outil détecte une occurrence de l'incohérence ;
- *Non détectée* si l'outil ne la détecte pas ;
- *Ne peut pas être testée* si l'outil ne supporte pas une des constructions mises en jeu dans la règle et que l'évaluation n'est donc pas possible.

Le tableau 2.3 donne les résultats synthétisés de cette évaluation.

Le taux de détection est bas : 25% à 30% pour les éléments et 12% à 25% pour les relations. Trois causes ont été dégagées pour expliquer ces résultats décevants :

- une insuffisance du pouvoir de détection des incohérences (notamment du checker) des outils, ceci montre que la marge de progression est importante ;
- les outils ne prennent pas en compte de nombreux mots clé, nous pensons que c'est une grosse lacune car les mots clés sont la description graphique de métaclasses différentes et donc de concepts différents ;
- au moment de l'étude, la spécification 2.0 d'UML était récente ; il serait donc intéressant de réaliser à nouveau cette étude afin d'apprécier si les versions plus

	Visual Paradigm	*Rational Rose*	*Ameos*
ÉLÉMENTS			
Détectée	25.00 %	30.83 %	25.42 %
Non détectée	48.33 %	35.00 %	44.17 %
Ne peut pas être testée	26.67 %	34.17 %	30.42 %
RELATIONS			
Détectée	12.17 %	13.04 %	20.87 %
Non détectée	73.91 %	64.35 %	52.17 %
Ne peut pas être testée	13.91 %	22.61 %	26.96 %
TOTAL			
Détectée	20.85 %	25.07 %	23.94 %
Non détectée	56.62 %	44.51 %	46.76 %
Ne peut pas être testée	22.54 %	30.42 %	29.30 %

Tab. 2.3 – Taux de détection des outils

évoluées de ces outils obtiennent de meilleurs résultats ;
– notre étude introduit de nombreuses nouvelles règles, ces règles ne sont pas détectées par les outils car les concepteurs des outils n'avaient pas connaissance de ces règles lors de leur réalisation.

2.4.3 Conclusion

L'ensemble des règles de cohérence identifiées est conséquent. Le document produit [50] est le résultat de la première étape de la gestion du risque, à savoir l'identification des événements dommageables que sont les incohérences. À notre connaissance nous avons réalisé la liste la plus complète des incohérences. Elle a tout d'abord été utilisée pour estimer le risque de chaque incohérence par des techniques d'interviews d'experts et des revues manuelles [79, 78]. Elle a aussi été utilisée pour tester la capacité d'outils de modélisation industriels à détecter des incohérences.

Ce document n'est cependant pas figé puisqu'il faudra le faire évoluer en fonction des versions d'UML. De plus, l'expression en langage naturel des règles de cohérence a un inconvénient majeur : il ne permet pas d'automatiser la détection des incohérences. Ceci représente une limite importante vu le nombre de règles et la complexité importante des modèles industriels.

Chapitre 3

Analyse de la cohérence

Ce chapitre s'intéresse à l'étape de détection des incohérences. C'est une activité qu'il faut automatiser au maximum. Il est en effet très difficile de réaliser cette activité manuellement à cause du grand nombre de règles à vérifier et de la complexité des modèles industriels.

Les résultats présentés précédemment définissent des règles de cohérence en langage naturel ce qui empêche leur détection automatique. Nous nous intéressons maintenant aux méthodes qui permettent de vérifier automatiquement ces règles. Le but de ce travail consiste à définir et à mettre en œuvre une méthode dont le taux de couverture des règles effectivement traitées est le plus important possible.

La détection automatique des incohérences est une activité à associer au traitement du risque d'incohérence. Identifier la présence d'une incohérence permet en effet d'y apporter un traitement. Ce traitement peut consister à résoudre l'incohérence mais aussi à diminuer la gravité du dommage engendré par une documentation adaptée par exemple.

La section 3.1 présente les concepts liés à la cohérence des modèles UML ainsi qu'un état de l'art des techniques employées pour leur détection. La section 3.2 présente l'approche proposée pour pallier les limites établies à la section précédente.

3.1 Techniques de détection

Cette section présente dans un premier temps le cadre générique des techniques de détection (cf. partie 3.1.1) puis fournit un tour d'horizon des travaux réalisés sur le sujet (cf. partie 3.1.2).

3.1.1 Cadre de la vérification de cohérence

Le langage UML étant une notation graphique, en dehors des techniques de détection manuelle, la détection des incohérences est basée sur une représentation des modèles UML dans un langage formel (cf. figure 3.1).

Les *modèles UML* sont basés sur le *formalisme UML* qui est décrit par le métamodèle d'UML. Les *modèles UML* sont transformés ou encodés dans un formalisme basé sur un *langage formel*. On obtient ainsi un *modèle formel*. Une distinction est réalisée entre encodage et transformation du modèle UML vers le modèle formel : l'encodage peut rendre compte de toutes les constructions des modèles UML alors qu'une transformation est la source de perte d'informations.

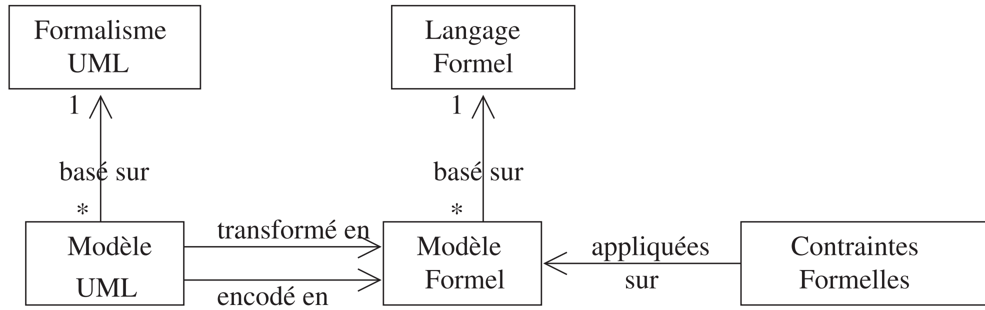


FIG. 3.1 – Vérification de cohérence des modèles UML (1/2)

Une fois le *modèle formel* obtenu il est possible d'appliquer les *contraintes formelles* que ce modèle doit respecter. Dans notre cas, ces contraintes formelles sont une formalisation des *règles de cohérence*, i.e. des règles génériques que tout modèle doit respecter. Nous abordons successivement les notions d'encodage, de transformation et de contraintes formelles.

Encodage La différence entre encodage et transformation vient du niveau auquel sont définies les relations entre modèle UML et modèle formel. Comme le représente la figure 3.2 l'encodage est défini au niveau méta-métamodèle (ou méta-langage). Nous verrons plus tard que pour la transformation, cette définition est effectuée au niveau métamodèle (ou langage).

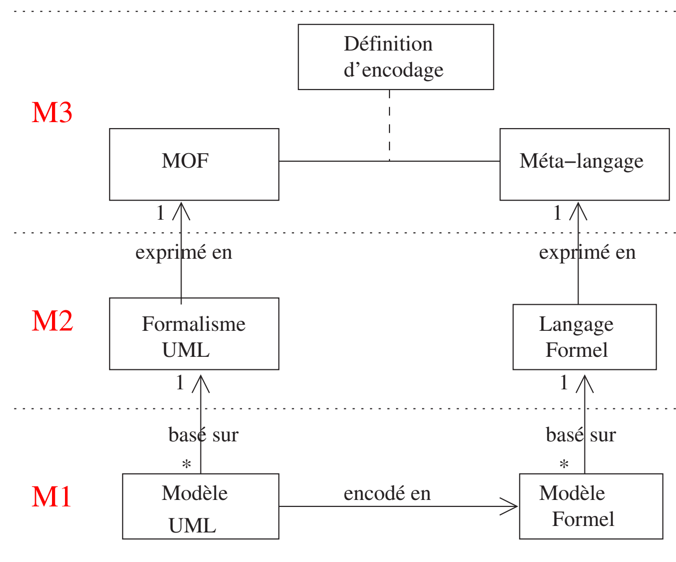


FIG. 3.2 – Encodage

L'encodage le plus courant des modèles UML car normalisé par l'OMG est celui qui consiste à représenter un modèle UML en XMI (*XML Metadata Interchange*) [60]. D'après l'OMG, « *XMI provides a mapping from MOF to XML* »[1]. La définition du MOF étant réalisée au niveau M3, cette correspondance l'est aussi. Ceci permet d'encoder tout modèle basé sur un langage exprimé en MOF par un document XML. Concrètement,

[1]XMI définit une correspondance entre le MOF et le XML

cette correspondance est définie par des règles EBNF (Extended Backus-Naur Form) permettant de générer des schémas XML pour tout métamodèle exprimé en MOF (niveau M2). Au niveau M1, XMI définit la génération de document XML qui encode le modèle considéré. Les détails concernant les règles EBNF qui permettent une telle définition sont fournis en annexe A.

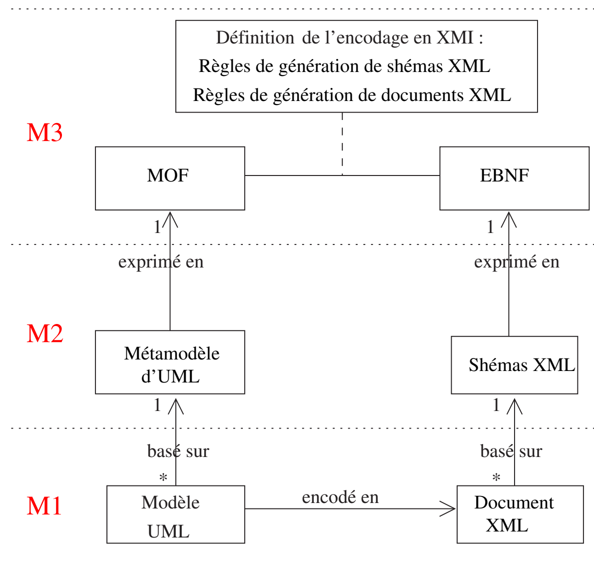


FIG. 3.3 – Définition de l'encodage de modèles en XMI

Notons que [60] spécifie explicitement que retranscrire l'ensemble des informations d'un modèle est primordial : « *All significant aspects of the metadata are included in the XML document and can be recovered from it. No information is lost* »[2].

Transformation Au contraire la définition d'une transformation est définie au niveau métamodèle (cf. figure 3.4). Le principe des transformations consiste en effet à

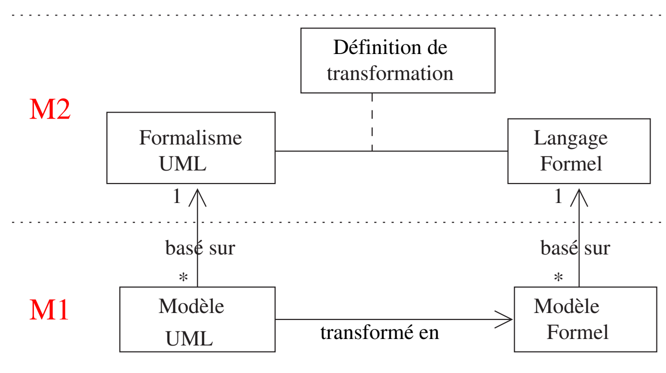


FIG. 3.4 – Transformation

définir des relations entre les concepts de deux langages pour pouvoir passer de l'un à

[2]Tous les aspects significatifs des méta-données sont inclus dans le document XML et peuvent en être extraites. Aucune information n'est perdue.

l'autre. Ainsi [55] transforme des modèles UML dans le langage IF [10]. Cette transformation consiste par exemple à transformer chaque classe X du modèle UML en un type de processus contenant des variables locales qui correspondent à chaque attribut ou association de X. La transformation est cependant source de perte d'information. Par exemple, [55] précise que certains concepts d'UML ne sont pas représentés dans le modèle formel, comme par exemple les cas d'utilisation car ils n'ont pas de sémantique opérationnelle clairement définie. Les techniques de transformation ne peuvent pas assurer la vérification de la cohérence des modèles UML dans leur intégralité. Ceci provient du fait que les paradigmes ou concepts du langage UML n'ont pas de correspondance dans le langage cible (ici IF) alors qu'un langage d'expression des besoins pourrait formaliser les cas d'utilisation. D'autres exemples de transformation seront fournis à la section 3.1.2.

Contraintes formelles Les contraintes formelles définissent des règles exprimées sur le langage formel. Elles doivent formaliser les règles de cohérence. Deux types de règles de cohérence sont distingués :
- les *règles de cohérence UML* qui sont induites par le langage UML lui-même, le processus utilisé ou d'éventuelles extensions d'UML (cf. partie 2.1.1) ;
- les *règles de cohérence du langage formel* qui sont des propriétés que doivent respecter tout modèle exprimé dans ce langage ; en effet le langage cible possède lui-même une sémantique de vérification.

La figure 3.5 complète la vue d'ensemble des éléments intervenant dans la vérification de la cohérence des modèles UML.

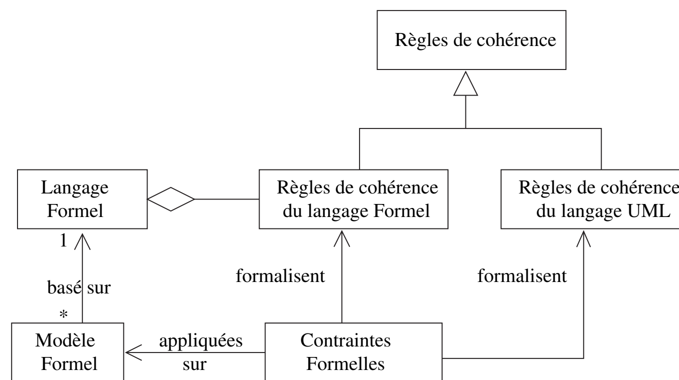


Fig. 3.5 – Vérification de cohérence des modèles UML (2/2)

3.1.2 Techniques proposées

Dans cette partie nous présentons les techniques de détection proposées dans la littérature en les plaçant dans la cadre introduit en 3.1.1. Nous présentons séparément les travaux basés sur la transformation des modèles UML et les travaux basés sur leur encodage.

Approche par transformation des modèles UML [3] présente le profil d'UML nommé TURTLE (**T**imed **U**ML and **RT-L**OTOS **E**nvironment) doté d'une sémantique formelle donnée en RT-LOTOS (basé sur une algèbre des processus). Ces travaux portent sur le diagramme de classes et le diagramme d'activités. Le profil décrit des extensions pour le temps-réel sur ces diagrammes. La construction d'un graphe d'atteignabilité permet de détecter des erreurs de timing et de logique. Par exemple, les auteurs vérifient que la périodicité d'envoi de signaux est bien respectée.

Les travaux décrits en [51] consistent à transformer des modèles UML/OCL en spécification formelle B. Ces travaux concernent les diagrammes de classes. Ils permettent notamment de traduire des pre/post-conditions et des invariants exprimés en OCL vers le langage B. Une fois cette représentation obtenue, les obligations de preuves associées au langage B permettent d'assurer la cohérence des pre/post-conditions vis-à-vis des invariants. La méthode est illustrée par le modèle d'une entreprise qui met en jeu des départements, des projets et des salariés. Un certain nombre d'invariants sont exprimés et vérifiés vis-à-vis des Pre/Post conditions des opérations, par exemple un département a au moins autant d'employés que chaque projet contrôlé par le département.

Les travaux de [32] transforment les modèles UML vers CSP (Communicating Sequential Processes). Les auteurs s'intéressent à l'analyse de propriétés induites par l'héritage entre deux classes et leurs machines à états respectives. La cohérence des séquences d'appel de méthodes spécifiées par les deux machines à états est ainsi vérifiée par des règles comme « *Hence, each sequence observable with respect to a subclass must result (under projection to the methods known) in an observable sequence of its superclass.* »[3].

[25] étudie les diagrammes d'activités en les transformant en système de transitions pour la vérification de propriétés dynamiques exprimées en CTL (Computation Tree Logic). Cette logique introduit des opérateurs temporels permettant de déclarer des propriétés sur l'évolution des états d'un système. Les propriétés vérifiées ici ne sont pas des règles de cohérence mais des propriétés spécifiques à une application. Ce type de technique pourrait cependant être utilisé pour formaliser des règles de cohérence. Les auteurs analysent un diagramme d'activité modélisant le fonctionnement d'une entreprise. Ils vérifient par exemple que si la production d'un objet a été planifiée, celui-ci sera bel et bien produit.

Les techniques transformationnelles sont intéressantes car elles permettent d'utiliser les capacités d'analyse des langages formels cibles. Ces langages formels sont souvent associés à des techniques de vérification évoluées, par exemple des techniques de model-checking avec réduction partielle, d'analyse statique, etc. De plus, ces techniques permettent de fournir une sémantique opérationnelle à UML par la transformation vers le langage cible. Cependant plusieurs limites sont relevées :

- ces études ne concernent que certaines caractéristiques de certains diagrammes d'UML dont la traduction peut être faite dans le langage formel choisi. Par exemple, certains aspects dynamiques des modèles UML vers les réseaux de Petri ;
- les propriétés vérifiées sont restreintes à celles analysables par le langage formel, par exemple, la « liveness » pour les réseaux de Petri ; toutes les contraintes UML

[3] Chaque séquence observable [des traces correspondant à l'appel de méthodes] d'une classe spéciale doit résulter (en ne tenant compte que des méthodes connues par la classe générale) d'une séquence observable de la classe générale (super-classe).

ne sont donc pas prises en compte ;
- une fois une propriété du modèle formel niée, les outils signalent très rarement les éléments du modèle UML qui sont à l'origine de l'incohérence révélée mais font apparaître l'effet des incohérences sur le modèle formel vérifié ;
- la relation sémantique entre le diagramme UML de base et le langage cible n'est pas toujours explicite.

Approche par encodage des modèles UML Rappelons que l'approche par encodage de modèles a pour but de traduire les modèles UML dans une notation formelle en conservant l'ensemble des informations contenues dans les modèles. XMI (XML Metadata Interchange) est la notation formelle la plus utilisée. Elle a été standardisée pour faciliter l'échange de modèles UML entre les outils et est basée sur la technologie XML. Un tel formalisme permet de représenter toutes les constructions du langage UML : les modèles UML sont encodés en XMI. Cette solution a l'avantage d'être standardisée et supportée par la plupart des outils de modélisation. OCL (Object Constraint Language) est le langage de contrainte normalisé par l'OMG et qui s'applique sur le format XMI. Il est fréquemment utilisé comme langage formel d'expression de contraintes. De nombreux travaux s'appuient donc sur l'encodage des modèles en XMI et la formalisation des contraintes UML en OCL.

Par exemple [8] utilise OCL pour vérifier des contraintes liées au processus USDP.

[45] présente une démarche outillée dont le but est de vérifier le respect de règles méthodologiques. L'expression des règles est réalisée en OCL. Ce papier fait suite au projet Neptune [6] dont le but est de développer des méthodes et des outils associés au langage UML. Il présente un certain nombre de règles méthodologiques parmi lesquelles : « *tout cas d'utilisation est illustré par au moins un scénario* » ou « *lorsque deux classes sont en association et qu'elles appartiennent à deux noeuds physiques différents alors ces deux noeuds sont en communication* ».

Une autre approche est décrite en [33]. L'outil *xlinkit* permet de vérifier la cohérence de documents basés sur XML avec un langage autre qu'OCL basé sur la logique du premier ordre. Cette approche peut donc être utilisée pour la vérification de modèles encodés en XMI. Par exemple, la règle « *No opposite association ends may have the same name within a classifier* »[4] est vérifiée.

Les travaux présentés dans cette partie restent cependant limités à des propriétés de type structurel. Ces limitations viennent du langage d'expression des propriétés. C'est pourquoi certaines recherches étendent le langage OCL afin de permettre la vérification de propriétés dynamiques par l'ajout d'opérateurs temporels [7, 29]. L'outillage correspondant reste cependant très spécifique et insuffisant.

Une autre approche consiste à encoder les modèles UML à l'aide d'un langage formel qui offre des capacités d'analyse propre. La définition de la notion d'instance permet d'encoder les modèles UML avec ce langage ; tout modèle UML est alors vu comme une instance du métamodèle formalisé. La mise en œuvre de stratégies de maintien de la cohérence des modèles grâce à la capacité d'analyse du langage choisi est alors possible. Pour que cette méthode soit efficace, il faut que le langage choisi permette de représenter

[4] Le nom des fins d'associations navigables d'un classificateur ne doivent pas avoir le même nom.

toutes les constructions du métamodèle d'UML et que la capacité d'analyse du langage soit suffisante pour exprimer les propriétés voulues.

[74] utilise le moteur d'inférence *sherlock* afin d'encoder les modèles UML et de formaliser des contraintes UML. Cette approche a l'avantage de mettre en œuvre des règles qui comprennent une partie action. Ainsi, lorsqu'une contrainte est violée des actions dont le but est de traiter l'incohérence sont exécutées. Ces actions peuvent par exemple correspondre à des suggestions fournies à l'utilisateur pour rectifier l'incohérence, à la mise en valeur des éléments incohérents, etc. Ces travaux permettent de vérifier des propriétés statiques et d'écrire certaines règles sur les règles elles-mêmes, comme par exemple que deux règles ne doivent pas être contradictoires.

[5] décrit une formalisation du métamodèle d'UML par un système de transitions étiquetées (*labelled transitions systems* en anglais). Cette formalisation prend en compte des aspects comportementaux. Un modèle est dit cohérent si c'est une instance du métamodèle ce qui permet de vérifier automatiquement certaines propriétés dynamiques.

Les travaux de [41, 26] utilisent Object-Z comme méta-langage, c'est-à-dire que le métamodèle d'UML est exprimé en Object-Z. [41] utilise cette formalisation du métamodèle pour écrire les contraintes de cohérence comme des invariants sur les méta-classes : toute instance (dans le modèle) d'une méta-classe décrite en Object-Z doit respecter les invariants. Cette formalisation sert par exemple à prouver le respect de cohérence structurelle. L'auteur s'intéresse plus particulièrement aux contraintes inter-diagramme, comme par exemple « *For an event in general, there should be a state machine that includes a transition that is trigged by the event describing the detailed effects of the reception of the event* »[5]. [26] utilise cette formalisation de métamodèle pour appliquer les étapes des preuves réalisées en Z sur les modèles UML eux-mêmes. En Z, une preuve est le résultat de l'application de règles d'inférences ou d'axiomes. Chaque étape de la preuve construit ainsi une nouvelle formule. Ce principe est retranscrit à UML en appliquant aux modèles des règles qui transforment les diagrammes. Ces travaux font parti des activités du groupe pUML (*precise UML*).

3.1.3 Limites des travaux actuels et besoins

La principale limite des travaux actuels est qu'il est impossible d'appliquer sur les *modèles formels* dérivés des modèles UML toutes les *contraintes UML* exprimées par les règles de cohérence. Deux étapes sont la cause de cette limite, la formalisation des contraintes UML d'une part, et la transformation des modèles UML d'autre part. Tout d'abord, la formalisation des contraintes UML est souvent limitée à un certain type de contraintes (par exemple OCL ne permet la formalisation que des contraintes structurelles). Dans ce cas c'est le langage d'expression des contraintes qui est limitatif. D'autre part la transformation des modèles UML vers un langage cible ne concerne pas l'intégrité des modèles UML, les propriétés vérifiées sur les modèles cibles sont donc limitées aux constructions qui ont leur correspondance dans le langage cible. L'application des contraintes sur un modèle formel obtenu par transformation n'assure donc pas la cohérence des modèles UML dans leur intégralité. De plus, le lien entre les *règles de cohérence du langage formel* et les *règles de cohérence du langage UML* n'est pas toujours expli-

[5]Pour un événement en général, il doit exister une machine à états qui contient une transition activée par cet événement ce qui décrit l'effet de la réception de cet événement.

cité (cf. figure 3.5). Il est donc difficile de projeter sur le modèle UML les raisons de la détection d'une incohérence liée au langage formel. Dans ce cas c'est la transformation qui est limitative.

UML est dit unifié car il permet de couvrir tous les aspects de la modélisation d'un logiciel, certains pouvant être rajoutés par le biais des stéréotypes. Notre point de vue est que la vérification d'un langage unifié doit être réalisée par une sorte de vérificateur unifié qui couvre l'ensemble des besoins d'expression et d'analyse des contraintes sur l'ensemble des diagrammes UML. En particulier, deux grands types de contraintes peuvent être dégagées, des contraintes structurelles (eg. « *tous les membres d'un espace de nommage doivent être distinguables* ») et des contraintes comportementales (eg. « *toutes les transitions d'une machine à états doivent être tirables* »). De plus, tous les diagrammes sont concernés pour le maintien de la cohérence.

Par ailleurs, la détection d'incohérence doit fournir un diagnostic suffisamment détaillé pour être exploité par un utilisateur ou d'autres outils afin de traiter les incohérences.

3.2 Principe de l'approche

La section 3.2.1 présente les raisons du choix de CLP (*Constraint Logic Programming*) comme langage formel à la fois d'encodage des modèles et d'expression des contraintes. La justification se base sur les limites des travaux actuels présentées précédemment et sur un état de l'art de travaux concernant la vérification de systèmes en utilisant les CLP. La partie 3.2.2 présente les CLP.

3.2.1 Choix du langage d'analyse

Nos travaux ont pour but d'assurer la cohérence des modèles UML dans leur intégralité. Nous choisissons donc de mettre en œuvre la technique de l'encodage plutôt que celle de la transformation. Cette technique est mise en œuvre par la formalisation du métamodèle. De plus, nous désirons vérifier l'ensemble des règles de cohérence exprimées en langage naturel. Le langage d'expression des contraintes choisi doit donc pouvoir formaliser l'ensemble des contraintes associées.

Le choix du langage d'encodage des modèles UML et du langage d'expression des contraintes a abouti à la sélection de CLP (*Constraint Logic Programming*). Nous présentons donc quelques travaux concernant la vérification de systèmes en utilisant les CLP pour justifier notre choix.

Ce type de programmation est tout d'abord capable d'analyser des contraintes structurelles. En effet, lors de l'analyse de propriétés structurelles, les modèles UML sont vus comme des données. C'est ce point de vue qui permet de les encoder en XML. Or les CLP peuvent être utilisés comme un langage de requêtes de base de données [76, 69]. La détection d'une incohérence peut donc être réalisée par une requête correspondant à une incohérence. La résolution par le moteur d'inférence d'une telle requête est possible si et seulement si l'incohérence est présente dans le modèle. La manière de représenter les modèles UML en CLP et de formaliser les incohérences structurelles est détaillée dans le chapitre 4.

La détection d'incohérences comportementales implique la connaissance de la sémantique opérationnelle d'UML, c'est-à-dire le comportement des modèles UML. Le comportement dynamique des systèmes discrets est fréquemment décrit par des règles de changement de configuration [47]. Nous adoptons ici cette approche et nous proposons d'encoder ces règles sous forme de règles de programmes logiques. Cette méthode a déjà fait ses preuves dans de nombreux travaux. Par exemple, [39] cite les travaux [28] qui ont utilisé la programmation logique pour définir la sémantique opérationnelle de Templog, une logique temporelle. [21] décrit la transformation de systèmes concurrents en CLP. Les auteurs présentent ensuite comment exprimer des propriétés qui s'inspirent de la logique temporelle CTL (*Computation Logic Tree*) en CLP. Ceci leur permet par exemple de vérifier que la spécification d'un algorithme qui vise à assurer l'exclusion mutuelle de deux processus à une section critique est valide. L'expression et la vérification de propriétés temporelles sont alors permises. [40] utilise les CLP pour vérifier des propriétés comportementales sur les automates temporisés grâce à une traduction systématique. L'automate considéré dans le papier modélise le fonctionnement d'un passage à niveau de train. La propriété de sûreté spécifiant que la barrière doit être baissée avant que le train n'entre dans la section critique est vérifiée. Pour finir, les travaux [66, 18] décrivent le model-checker XMC [13]. Ce model-checker est basé sur le système de programmation logique tabulé (TLP : *Tabled Logic Programming*) XSB [83]. L'implantation en TLP de CCS (Calculus of Communicating Systems) et de la logique temporelle utilisée (μ-calculus) pour exprimer les propriétés a été réalisée en 200 lignes de code. Ceci laisse présager d'une commodité d'implantation pour notre outil de vérification. [66] constate que les performances de cette approche rivalisent avec celles des model-checkers éprouvés tels que SPIN. Notons que le principe de résolution des TLP est légèrement différent du principe de résolution utilisé pour LP. Ces différences seront montrées en section 3.2.2.1. De plus, l'expression de la sémantique en XSB est très proche de la sémantique opérationnelle et structurelle (*Structural Operational Semantics*) qui définit la sémantique de CCS et de la logique temporelle utilisée. Ceci est dû à l'aspect déclaratif de la programmation logique. Cette approche se démarque des techniques de model-checking habituelles qui utilisent des techniques algorithmiques et non déclaratives.

3.2.2 Introduction aux CLP

Nous introduisons dans un premier temps la programmation logique tabulée puis de manière succincte la programmation logique avec contraintes. Cette section s'appuie principalement sur [27, 30].

L'idée centrale de la programmation déclarative (dont fait partie la programmation logique), est que la programmation se résume à la partie logique du problème sans se préoccuper de la partie contrôle. On ne s'intéresse donc pas à comment le problème est résolu (aspect procédural) mais à ce qu'est la solution (aspect déclaratif). Au contraire, en programmation classique le *quoi* (logique) n'est pas séparé du *comment* (contrôle) : un algorithme est constitué de composants logiques (qui représentent la connaissance vis-à-vis d'une application) et de structures de contrôle (qui déterminent comment cette connaissance est acquise). L'essence est donc de définir des règles de logique mathématique au lieu de fournir une succession d'instructions que l'ordinateur exécuterait. Ces règles permettent d'expliciter le modèle d'inférence défini par un programme logique.

3.2.2.1 Programmation logique

Un programme LP est un ensemble fini de clauses de la forme $H \leftarrow B_1, ..., B_n$ où $H, B_1, ..., B_n$ sont des atomes.

Un atome est de la forme $p(t_1, ..., p_n)$ où p est un symbole de prédicat et t_i sont des termes.

terme — *Soit $\mathcal{V}$ un ensemble des symboles appelés variables, $\mathcal{C}$ un ensemble de symboles appelés constantes et $\mathcal{F}$ un ensemble de symboles appelés fonctions. L'ensemble $\mathcal{T}$ des termes du premier ordre est défini inductivement comme le plus petit ensemble vérifiant les trois règles de fermeture suivante :*
- *$\mathcal{V} \subset \mathcal{T}$;*
- *$\mathcal{C} \subset \mathcal{T}$;*
- *si f symbole de fonction n-aire et $t_1, ..., t_n$ sont des termes alors $f(t_1, ..., t_n) \in \mathcal{T}$*

Un symbole de prédicat est défini dans un programme s'il apparaît en tête d'une clause.

Deux types de clauses sont à différencier : les faits sont des clauses pour lesquelles $n = 0$ et les règles sont des clauses telles que $n > 0$.

L'exécution d'un programme logique est la résolution d'un but de la forme $B_1, ..., B_n$, le but vide ou but succès est noté $\top$ et le but échec est noté $\bot$. Nous verrons par la suite quand est-ce qu'un but est en échec.

L'*interprétation déclarative* d'une clause $H \leftarrow B_1, ..., B_n$ stipule que H est vrai si $B_1, ..., B_n$ sont vrais. La même clause de programme peut également être interprétée comme une règle de réécriture sur les buts : pour prouver H il suffit de prouver successivement les buts $B_1, ..., B_n$. Cette interprétation est appelée *interprétation procédurale.* Afin d'expliciter ces deux interprétations d'un même programme logique, nous présentons l'interprétation déclarative au moyen d'un exemple et présentons ensuite le principe de résolution SLD qui est le fondement de l'interprétation procédurale.

Exemple intuitif L'interprétation déclarative des programmes logiques est maintenant présentée au travers d'un exemple. Cet exemple s'appuie sur la syntaxe particulière des systèmes de programmation logique. Une variable commence par une majuscule (ou par un underscore ('_') si sa valeur est indifférente). Une constante commence par une minuscule. Nous présentons la syntaxe des faits, des règles et des buts ainsi que leur interprétation déclarative.

FIG. 3.6 – Exemple de graphe (acyclique) et de sa représentation en fait

Les **faits** sont la base de connaissance du système. Ils formalisent ce qui est connu. Considérons, les faits fournis dans la partie gauche de la figure 3.6. Ceux-ci représentent le graphe de la partie droite de la figure 3.6.

Le fait `(e1)` signifie par exemple qu'il existe un arc entre les nœuds `a` et `b`.

Les **règles** représentent la base de déduction du système. Par exemple, les chemins directs et indirects entre deux nœuds sont donnés par les règles :

```
reach(Start,End) :- trans(Start,End).                          (p1)
reach(Start, End) :- trans(Start, Node), reach(Node, End).    (p2)
```

Elles expriment qu'il est possible d'atteindre un nœud `End` à partir d'un nœud `Start` :
- s'il existe un arc (représenté par un fait `trans`) entre ces deux nœuds (règle `p1`) ;
- ou s'il existe un nœud intermédiaire `Node` tel qu'il existe un arc entre le nœud `Start` et le nœud `Node` et qu'il existe un chemin entre ce dernier et le nœud `End` (règle `p2`).

Au sein d'une règle, une virgule est équivalente au *et* logique (conjonction). L'expression d'une disjonction est réalisée par l'écriture de plusieurs règles.

Les **buts** sont une manière d'interroger la base de connaissance et de déduction représentée par un programme logique. La réponse fournie est une substitution qui s'applique aux variables du but telle que le but devient une conséquence logique de la base de connaissance et de déduction du programme. Par exemple, l'exécution du but `reach(b,Y).` permet de connaître les différents nœuds qui sont accessibles à partir de `b`. Le joueur LP donne donc successivement les réponse suivantes : `Y=d` et `Y=e`.

Principe de résolution SLD Nous présentons maintenant le principe de résolution SLD qui est le fondement de la sémantique opérationnelle des programmes logiques.

Un programme LP exécute un but G de la forme $B_1, ..., B_n$. L'état de résolution est un doublet $< G, \theta >$ où G est une conjonction de buts et θ une substitution. L'état initial est de la forme $< G, \epsilon >$ où ϵ est la substitution identité. Un état est un état final succès s'il est de la forme $< \top, \theta >$, c'est un état final en échec s'il est de la forme $< \bot, \epsilon >$.

Le principe de résolution est constitué d'un ensemble de transitions élémentaires appelés "pas" d'inférence. Un pas d'inférence modifie l'état de la résolution à la suite de l'application d'une clause. Appliquer une clause $H \leftarrow G$ consiste à :
- essayer d'unifier la tête H d'une clause du programme avec le sous-but B_i à réduire ;
- remplacer le sous-but B_i à réduire par le corps G de la clause sélectionnée ; remarquons que lorsque la clause sélectionnée est un but, G est vide ;
- composer la substitution courante avec la substitution obtenue lors de l'unification de la tête de clause.

Ce principe est appelé principe de résolution SLD. Il est défini formellement de la manière suivante :

$$\text{SLD} : \frac{(H \leftarrow G)\sigma \in P \quad \theta \in UP(B_i, H)}{< (B_1, ..., B_i, ..., B_n), \beta > \longrightarrow < (B_1, ..., B_{i-1}, G, B_{i+1}, ..., B_k), \theta\beta >}$$

où :
- $(H \leftarrow G)$ est la clause du programme sélectionnée ;
- $< (B_1, ..., B_i, ..., B_n), \beta >$ est l'état de la résolution avant application de la transition ; $(B_1, ..., B_i, ..., B_n)$ est le but et β la substitution avant l'étape de résolution ;
- B_i est le sous-but à résoudre pour l'étape de résolution SLD ;

– $\theta \in UP(B_i, H)$ signifie que θ est un unificateur principal de B_i et de H ;
– $< B_1, ..., B_{i-1}, G, B_{i+1}, ..., B_k), \theta\beta >$ est l'état résolvant, i.e. le nouvel état à résoudre après application de la clause de programme sélectionnée sur le sous but B_i ;
– enfin, σ est une substitution de renommage destinée à éviter les conflits de variables entre la clause de programme et le but sélectionné.

Une dérivation est un ensemble de ces pas SLD. Les dérivations peuvent amener à trois cas de figures :
– à un succès dans le cas où l'état final est l'état succès, c'est le cas lorsque le sous but est vide, une telle dérivation est notée $< G, \epsilon > \mapsto^* < \top, \theta >$;
– à un échec dans le cas où l'état final est l'état échec, c'est le cas lorsqu'un état est atteint avec un but non vide dans lequel un sous-but ne s'unifie avec aucune tête des clauses du programme logique ;
– à une dérivation infinie si aucun état final n'est atteint.

Le système de transitions basé sur la résolution SLD comporte deux sources de non déterminisme, le choix du sous-but à résoudre et le choix de la clause de programme à appliquer.

La stratégie en profondeur d'abord efficace en terme de performance n'est cependant pas complète car même si une réponse au but existe le système peut boucler indéfiniment. Cet inconvénient majeur est cependant en partie résolu grâce à la programmation logique tabulée qui utilise le principe de résolution SLG [15] que nous présentons maintenant. Ce principe de résolution est implanté par le système XSB que nous utiliserons [83, 70, 71].

Principe de résolution SLG Le principe de résolution SLG est intéressant pour le model-checking puisqu'il permet d'empêcher (sous certaines conditions) le bouclage infini inhérent au principe de résolution SLD.

La résolution SLG suit les mêmes principes fondamentaux de la résolution SLD, mais garde en mémoire dans des tables les buts qui ont déjà été explorés. Or, il est possible lors d'une dérivation que la résolution d'un but G amène à la résolution de ce but G. Avec le principe de résolution SLD ceci entraîne une dérivation infinie. L'approche très simple introduite par le principe de résolution SLG consiste à regarder dans les tables si le but n'a pas déjà été résolu et, si c'est le cas, éviter de résoudre ce but à nouveau et exploiter les réponses déjà obtenues.

Pour faire comprendre ce principe nous nous inspirons fortement de [18].

Prenons l'exemple d'un graphe représenté par les faits suivants :

```
trans(a,b).   (e1)
trans(b,c).   (e2)
trans(c,b).   (e3)
```

$$a \;\rightarrow\; b \;\leftrightarrow\; c$$

FIG. 3.7 – Exemple de graphe (cyclique)

Le prédicat `reach/2` spécifie la relation d'atteignabilité entre deux sommets :

```
reach(X,Y) :- trans(X,Y).                        (e4)
reach(X,Y) :- trans(X,Int), reach(Int,Y).        (e5)
```

La figure 3.8 présente l'arbre de dérivation SLD. Cet arbre est infini. Les atomes à gauche du signe `:-` est le nœud de l'arbre qui représente la réponse. La liste des atomes

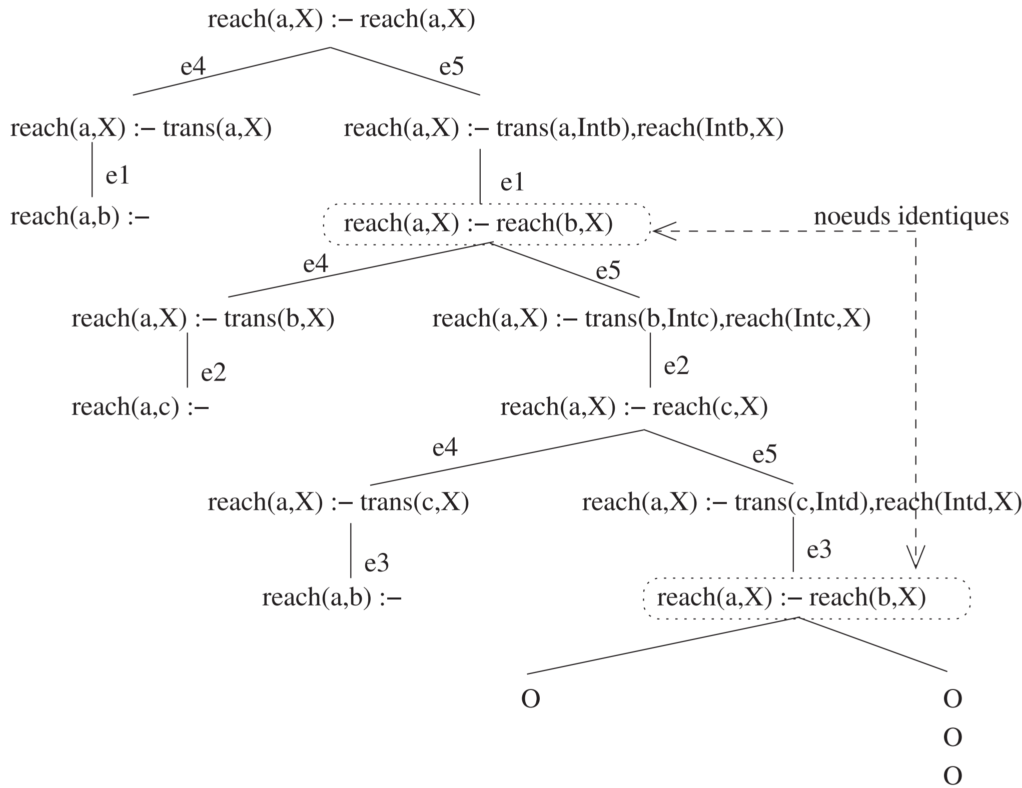


Fig. 3.8 – Arbre de résolution infini pour reach(a,X)

de droite est le but à résoudre. Les branches de l'arbre sont étiquetées par la clause du programme logique appliquée. Chaque chemin partant de la racine de l'arbre à une feuille est une dérivation SLD. Notons que des réponses correctes sont obtenues dans trois feuilles. Il est intéressant de remarquer que deux nœuds de l'arbre sont identiques. Le chemin se répète ainsi de manière indéfiniment, et les résultats obtenus ne permettent donc pas d'assurer que toutes les réponses ont été déduites.

Observons maintenant l'arbre de dérivation obtenu par le principe de résolution SLG. Le programme est le même mais il faut rajouter la directive `:-table reach/2.` qui indique que le prédicat `reach` d'arité 2 est tabulé. Dans le cas de la résolution SLG, chaque résolution d'un but `reach/2` crée un nouveau sous-arbre si cet arbre n'a pas déjà été calculé. La figure 3.9 représente ces sous-arbres. La résolution du but `reach(a,X)` produit le sous-arbre de la figure 3.9 i). Lorsque le sous-but à atteindre est `reach(b,X)` le sous-arbre de la figure 3.9 ii) est créé. De même, l'invocation du sous-but `reach(c,X)` produit le sous-arbre de la figure 3.9 iii). Par contre, l'invoquation du sous-but `reach(b,X)` dans ce dernier ne produit pas le calcul d'un nouveau sous-arbre, car la résolution a été réalisée et sauvegardée dans les tables. Après cette évaluation les réponses sont retournées et aucun nouveau sous-but n'est généré. En fait le calcul se termine lorsqu'il a atteint un point fixe.

Notons tout de suite l'intérêt que ce principe de résolution offre pour la réalisation de model-checkers. Ce paragraphe s'appuie sur [72]. Le but du model-checking est de montrer qu'un modèle vérifie une propriété donnée. Il peut s'agir de propriétés de sûreté, d'accessibité, de vivacité, etc. Ces propriétés sont exprimées au moyen de logiques com-

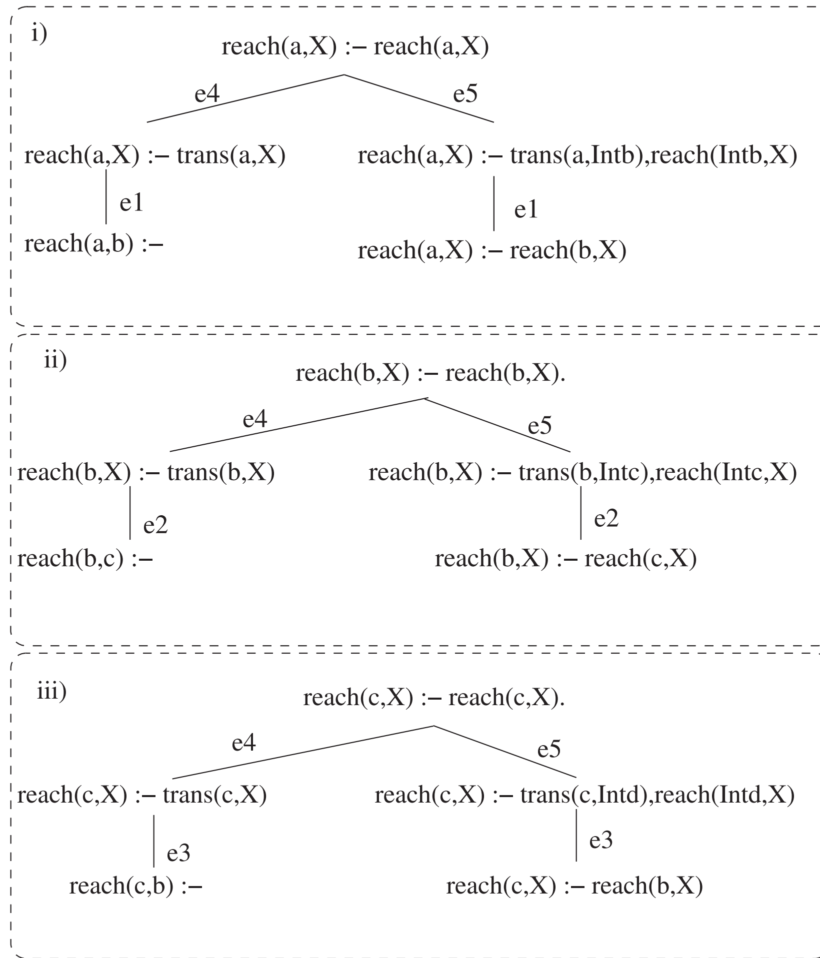


Fig. 3.9 – Les trois sous-arbres de résolution SLG pour le but reach(a,X)

portementales (LTL, CTL, CTL*, μ-calcul, etc.). Une propriété de sûreté énonce que, sous certaines conditions, quelque chose ne se produit jamais pour assurer qu'une situation redoutée ne peut pas être atteinte. Assurer le respect de telles propriétés nécessite donc la connaissance de l'ensemble des états accessibles du modèle. En effet, si un état accessible n'est pas identifié, il est possible que ce dernier ne respecte pas la propriété de sûreté globale. Or, l'ensemble des états accessibles est obtenu par un calcul de point fixe ($x = f(x)$ où x représente ici un ensemble d'états). L'ensemble des états de départ est l'ensemble des états initiaux et la fonction appliquée est la fonction de transition du modèle. Ceci permet de calculer pas à pas les états accessibles depuis l'ensemble des états précédents. Lorsque cet ensemble d'états accessibles ne croît plus le point fixe est atteint ce qui permet d'affirmer que tous les états ont été explorés. La figure 3.10 donne un exemple de calcul de point fixe.

3.2.2.2 Programmation logique avec contraintes

Selon [27] les langages de programmation logique avec contraintes sont une généralisation des langages de programmation logique dans laquelle on peut considérer n'importe quel système de contraintes en plus des contraintes d'égalité sur le domaine de Herbrand.

Le domaine de Herbrand possède en fait comme seule contrainte l'égalité syntaxique,

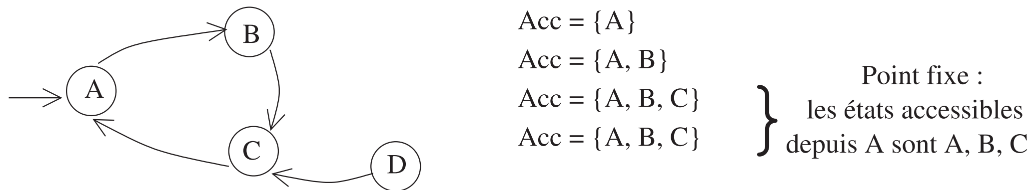


FIG. 3.10 – Exemple de calcul de point fixe

qui se résume aux principes d'unification et de substitution. Comme nous l'avons vu, le long d'un arbre de dérivation SLD, les substitutions sont composées. L'idée est de considérer d'autres structures mathématiques que l'univers de Herbrand en remplaçant la composition des substitutions par le cadre plus puissant de la résolution de contraintes sur ces structures. Les autres principes de résolution restent identiques.

Cette résolution de contraintes dépend des contraintes que l'on considère, d'où l'introduction du domaine de contraintes. Chaque domaine se verra en effet doté de techniques de résolution de contraintes qui lui sont adaptées. Les domaines les plus courants sont le domaine des réels ($CLP(\mathcal{R})$), des ensembles finis $CLP(\mathcal{FD})$ et des booléens ($CLP(\mathcal{B})$).

Dans l'état actuel de nos travaux les contraintes ne sont pas utilisées mais leur utilisation suscite des perspectives à nos travaux que nous citerons plus tard. Conscient de cela nous utiliserons donc la notation (C)LP lorsque les concepts utilisés se limitent à ceux de la programmation logique. Le principe de la programmation logique tabulée et de la programmation logique avec contraintes ne sont en effet pas incompatibles.

3.3 Conclusion

Le but de cette thèse concernant la vérification de cohérence est de fournir un moyen qui permette de :

– vérifier l'ensemble des règles de cohérence sur les modèles UML, notamment les règles qui concernent la structure et le comportement des modèles UML ;
– fournir un diagnostic précis sur ces incohérences de manière à faciliter le retour sur les modèles UML ; identifier une incohérence prend tout son intérêt s'il est possible d'identifier les éléments d'UML en cause dans l'incohérence ;
– fournir une méthode souple et évolutive, c'est-à-dire qui peut être adaptée en fonction des règles à vérifier et du langage de modélisation considéré (pour pouvoir entre autre prendre en compte les évolutions du langage UML).

Pour cela, nous nous proposons d'encoder les modèles UML en programmation logique. Ce langage sera aussi utilisé pour exprimer les règles de cohérence sous forme de contraintes formelles (cf. figure 3.5).

Ce chapitre justifie cette proposition d'approche car (C)LP semble adapté pour exprimer les incohérences structurelles en l'utilisant comme un langage de requêtes de base de données [69]. Il semble également adapté pour vérifier des incohérences comportementales [18]. Le chapitre 4 présentera la méthode d'analyse des incohérences structurelles et le chapitre 5 présentera la méthode d'analyse des incohérences comportementales.

Chapitre 4

Détection des incohérences structurelles

Le but de ce chapitre est de fournir un moyen permettant d'analyser la cohérence de la structure des modèles UML.

Comme expliqué dans le chapitre précédent, la programmation logique sera utilisée pour encoder les modèles UML et exprimer les règles de cohérence.

La section 4.1 donnera une vue d'ensemble des principes mis en œuvre. La section 4.2 décrit la méthode d'encodage des modèles UML en LP. Elle est basée sur le métamodèle. La section 4.3 considère la formalisation des règles de cohérence et le diagnostic associé.

4.1 Vue d'ensemble de la démarche

Cette partie donne une vue d'ensemble de notre démarche dont la figure 4.1 fournit une synthèse. L'étape 1 de cette figure représente la formalisation en (C)LP d'un modèle et de son métamodèle par des clauses (C)LP. Elle définit l'encodage des modèles UML en (C)LP. Cette étape constitue la pierre angulaire de notre méthode de vérification de la cohérence. L'étape 2 consiste à spécifier les contraintes à respecter en (C)LP associées aux règles de cohérence. Le joueur (C)LP peut alors fournir un diagnostic sur la cohérence du modèle UML (formalisé lors de l'étape 1) vis-à-vis des contraintes (exprimées lors de l'étape 2). Cette étape 3 est automatique.

L'exemple suivant illustre de manière intuitive la méthode proposée. Elle sera détaillée et enrichie aux sections suivantes.

La figure 4.2 présente un modèle UML et son encodage par des faits (C)LP. Ceux-ci sont ici déduits intuitivement. Le processus de déduction sera présenté à la section 4.2. Les modèles UML sont constitués d'éléments (ici des classes et des attributs) et de relations entre ces éléments (le fait qu'une classe possède un attribut). Dans l'exemple de la figure 4.2, on traduit les éléments « classes » par les faits `(1)` et `(2)` et les éléments « attributs » par les faits `(3)` à `(6)`. L'appartenance des attributs aux classes est décrite par les faits `(7)` à `(10)` basés sur l'identificateur des éléments mis en relation.

La deuxième étape de la vérification de cohérence consiste à formaliser les règles de cohérence. Ces formalisations sont appelées contraintes car elles contraignent les modèles. Nous exprimons les incohérences associées aux règles. Si le système de program-

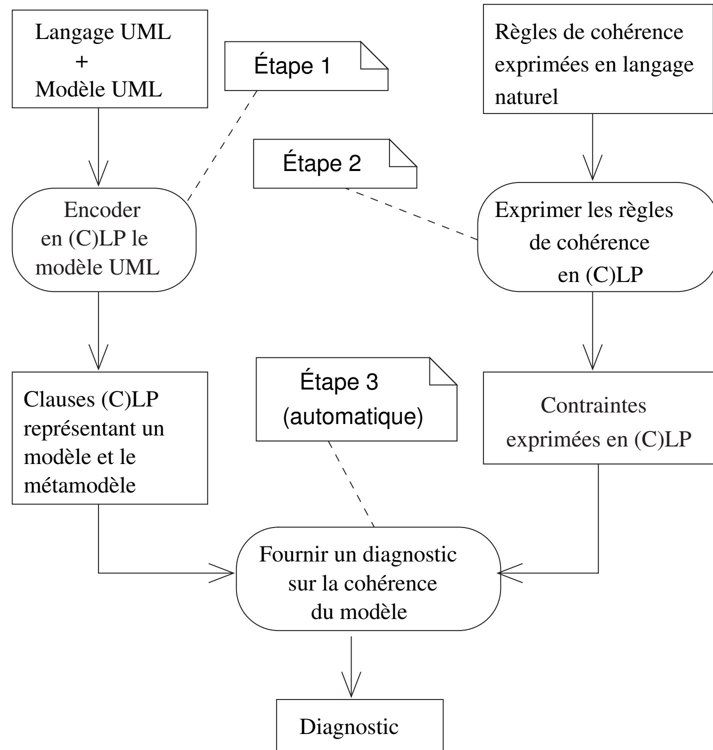


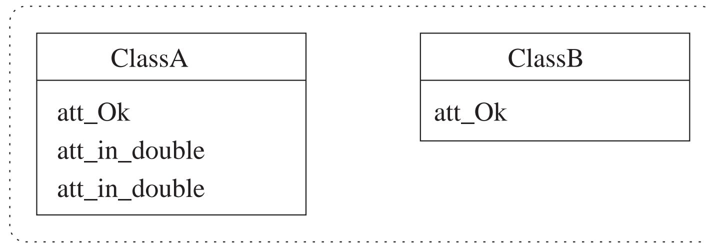
Fig. 4.1 – Démarche de mise en œuvre du vérificateur limité au niveau modèle

mation logique résout la contrainte sur un modèle donné, la présence de l'incohérence dans ce modèle est démontrée. L'incohérence numéro 1 associée à la contrainte « *une classe ne doit pas avoir deux attributs qui ont le même nom* » peut être formalisée par :

$$
\begin{array}{ll}
\multicolumn{2}{l}{incoherence(1, IdClass, IdAtt1, IdAtt2) :-} \\
(1) & class(IdClasse, _), \\
(2) & attribute_of(IdAtt1, IdClass), \\
(3) & attribute_of(IdAtt2, IdClass), \\
(4) & attribute(IdAtt1, Name1), \\
(5) & attribute(IdAtt2, Name2), \\
(6) & IdAtt1 \neq IdAtt2, \\
(7) & Name1 = Name2.
\end{array}
$$

où :

- (1) signifie que la contrainte s'applique à toutes les classes car les paramètres du prédicat sont des variables ;
- (2), (3), (4), (5) représentent le filtre de la contrainte et permettent de récupérer par navigation dans le modèle deux attributs quelconques (`IdAtt1` et `IdAtt2`) d'une même classe (`IdClass`) ainsi que leur nom respectif (`Name1` et `Name2`), ces attributs sont potentiellement les mêmes ;
- (6) est une contrainte qui oblige les deux attributs à être différents et (7) est une contrainte qui exprime que les deux noms doivent être identiques ; ainsi, la règle peut être satisfaite si et seulement si deux attributs différents et de noms identiques se trouvent dans la même classe, ce qui est bien incohérent.




```
(1) class(idClass1,classA).
(2) class(idClasse2,classB).
(3) attribute(idAtt1,att_in_double).
(4) attribute(idAtt2,att_in_double).
(5) attribute(idAtt3,att_OK).
(6) attribute(idAtt4,att_OK).
(7) attribute_of(idAtt1,idClass1).
(8) attribute_of(idAtt2,idClass1).
(9) attribute_of(idAtt3,idClass1).
(10) attribute_of(idAtt4,idClass2).
```

FIG. 4.2 – Modèle considéré pour l'exemple et représentation en faits

Le contrôle de cohérence du modèle vis-à-vis de la contrainte considérée dans l'exemple est obtenu par l'application du but
`?-incoherence(1, IdClass, IdAttribute1, IdAttribute2)`. Le joueur trouve une solution, c'est-à-dire que des éléments du modèle sont compatibles avec l'énoncé de l'incohérence. Plus précisément, il fournit la réponse :

```
IdClass = idClasse1
IdAttribute1 = idAtt1
IdAttribute2 = idAtt2
```

Le diagnostic met en avant que les éléments d'identificateur `idClasse1, idAtt1` et `idAtt2` sont incohérents vis-à-vis de la contrainte analysée. Cette solution a l'avantage de réaliser un diagnostic précis car les différents éléments du modèle UML mis en jeu dans l'incohérence sont identifiés.

4.2 Encodage des modèles UML en programmation logique

Dans la section précédente, l'encodage des modèles en faits (C)LP (étape 1 de la figure 4.1) a été effectué intuitivement. Afin de développer un outil, cet encodage doit être automatisé. Il est donc nécessaire de proposer une démarche systématique. Pour cela, nous nous plaçons dans un premier temps au niveau métamodèle (ou langage) (cf. partie 4.2.1) puis au niveau méta-métamodèle (ou méta-langage) (cf. partie 4.2.2).

4.2.1 Niveau langage

Nous présentons ici la technique d'encodage du métamodèle d'UML [59]. Nous présentons séparément la formalisation des constructions concrètes du métamodèle et

celle des constructions abstraites. Les constructions concrètes du métamodèle sont des constructions qui peuvent avoir des instances dans les modèles à l'inverse des constructions abstraites. Par exemple, les métaclasses (i.e. les classes du métamodèle) peuvent avoir des instances dans le modèle car créer un élément d'UML revient à instancier une métaclasse (se référer à l'exemple de l'introduction section 1.2.3). Par contre, certaines constructions du métamodèle n'ont pas d'instances dans les modèles, c'est par exemple le cas des relations de généralisation entre métaclasses.

4.2.1.1 Constructions concrètes du métamodèle

Comme vu précédemment, les modèles UML doivent être encodés par des faits (C)LP. Cette partie présente la démarche d'obtention générale de ces faits. Pour chaque type d'élément qui peut se trouver dans un modèle UML, c'est-à-dire pour chaque construction du langage UML, nous déduisons du métamodèle un format que devra respecter les faits représentant un élément UML de ce type. Ce format de fait est appelé méta-fait. La définition d'un méta-fait consiste à définir son nom, son arité et la signification de chacun de ses arguments.

Considérons le sous-ensemble du métamodèle présenté à la figure 4.3. Ce métamodèle exprime qu'une classe (métaclasse `Class`) est un espace de nommage (méta-généralisation entre ***`NameSpace`*** et `Class`) et qu'un espace de nommage est un élément nommé. Une classe est caractérisée par un nom, une visibilité (par héritage) et peut être abstraite. Toutes les classes d'un modèle peuvent donc être représentées par des faits de type :

- Méta-Fait 1 : $class(IdClasse, Name, Visibility, IsAbstract)$

où `Name` est le nom de la classe, `Visibility` est la visibilité de la classe et `IsAbstract` indique si la classe est abstraite ou non.

Une classe peut contenir des attributs (fin d'association `ownedAttribute`). Les attributs sont représentés par la métaclasse `Property` et ont comme caractéristiques un nom, une visibilité et peuvent être dérivés. Ils peuvent donc être représentés par des faits de type :

- Méta-Fait 2 : $property(IdProperty, Name, Visibility, IsDerived)$

Les relations entre classes et attributs peuvent être représentées par les méta-faits :

- Méta-Fait 3 : $ownedAttribute(IdClass, IdAttribute)$
- Méta-Fait 4 : $class(IdAttribute, IdClass)$

où `IdClasse` et `IdAttribute` sont les identificateurs de la classe et de l'attribut contenu. On a recours à deux types de faits car les deux fins de la méta-association mettant en relation une classe et son attribut sont navigables. Les méta-faits 1 et 4 ont le même nom mais sont distingués par leur arité. Notons que la définition d'un méta-fait qui représente une métaclasse tient compte des attributs hérités de la métaclasse.

La relation entre le méta-fait et les faits qui en sont dérivés respecte bien le fondement de la méta-modélisation qui veut que chaque couche est une instance de la couche supérieure. Par exemple, la figure 4.4 illustre que

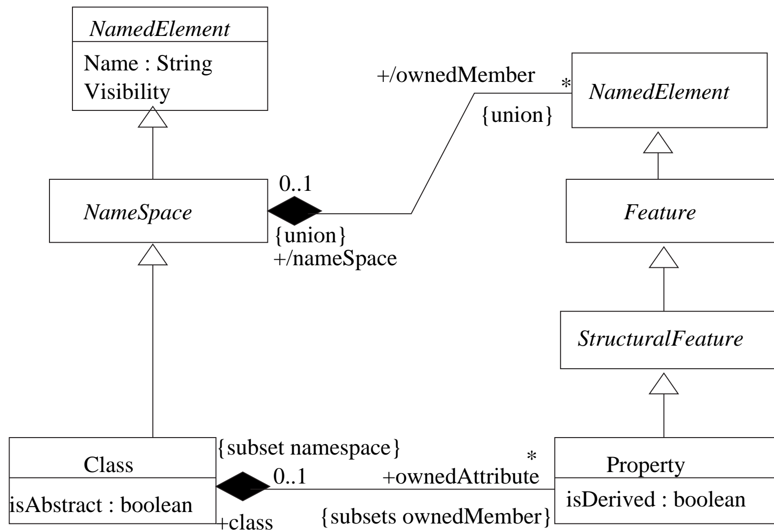


FIG. 4.3 – Sous-ensemble du métamodèle

le fait `property(id1,title,public,false)` est un instance du méta-fait `property(IdProperty,Name,Visibility,IsDerived)` au même titre que l'attribut `+title` est une instance de la métaclasse `property`. L'instanciation d'un méta-fait correspond ainsi à l'attribution de valeurs constantes aux paramètres variables du méta-fait de la même manière que l'instanciation d'une métaclasse correspond à donner des valeurs aux différentes caractéristiques de cette métaclasse.

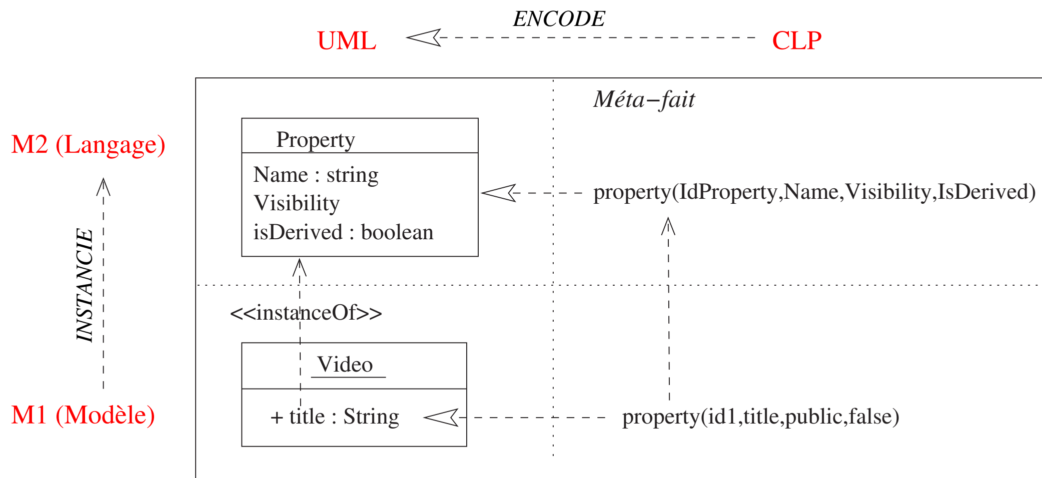


FIG. 4.4 – Relation entre faits, méta-faits, métaclasses et instances de métaclasses

Cette méthode systématique utilisée pour obtenir les faits permet d'effectuer des raisonnements au niveau métamodèle ce qui est nécessaire pour la vérification de cohérence. Cette méthode permet en particulier, d'accéder à des valeurs de méta-attributs, de naviguer entre les éléments du modèle au travers des fins de méta-associations navigables.

4.2.1.2 Constructions abstraites du métamodèle

Le cas des constructions abstraites du métamodèle est particulier.

Selon la figure 4.3 les espaces de nommage sont représentés par la métaclasse abstraite ***NameSpace*** et n'ont pas d'instance propre dans le modèle ; UML ne fournit en effet aucun moyen de représenter directement un espace de nommage. Les espaces de nommage peuvent cependant apparaître sous plusieurs formes (eg. classe, paquetage, etc.) qui correspondent aux métaclasses qui spécialisent ***NameSpace***. Ainsi, à partir de la présence d'une classe on peut déduire la présence d'un espace de nommage. En (C)LP, cette déduction est réalisée par la règle :

- RÈGLE 1 : $nameSpace(Id, Name, Visibility) :-$
  $class(Id, Name, Visibility, _IsAbstract)$

Cette règle correspond en fait à la formalisation de la méta-généralisation entre `Class` et `NameSpace`. Se pose alors la question de savoir si la sémantique des méta-généralisations est bien retranscrite par ce type de règles. [58] décrit la sémantique d'une généralisation par :

1. « *Each instance of the specific classifier is also an indirect instance of the general classifier* »[1].
2. « *Thus, the specific classifier inherits the features of the more general classifier* »[2]. Dans le cadre du métamodèle d'UML, les seules caractéristiques utilisées sont des caractéristiques structurelles que sont les attributs et les fins d'association navigables.

Nous expliquons maintenant pourquoi la règle 1 reflète bien les deux points sémantiques décrits précédemment.

Créer une classe dans un modèle revient à instancier la métaclasse `Class`. Cette règle décrit par projection de l'identificateur que si l'instance d'une classe est présente dans le modèle alors c'est aussi l'instance d'un espace de nommage. Cette règle décrit donc que chaque instance de la métaclasse `Class` est aussi une instance de la métaclasse `NameSpace` (point 1).

De plus les caractéristiques de l'instance de l'espace de nommage doivent correspondre aux caractéristiques de la classe. C'est pourquoi tous les méta-attributs de la classe qui sont aussi des méta-attributs de l'espace de nommage sont projetés sur ce dernier, c'est le cas des attributs `Name` et `Visibility`. Certains attributs de la classe spécifique n'appartiennent cependant pas aux attributs de la classe générale. C'est par exemple le cas du caractère abstrait de la classe représenté par `IsAbstract`. L'inverse est par construction faux car la définition des méta-faits tient compte des méta-attributs hérités. Dans ce cas, la valeur de l'attribut n'a aucun impact pour la règle et commence donc par '_' (point 2 pour les attributs).

D'autre part, les méta-associations accessibles par l'espace de nommage doivent également l'être par la classe. Or, puisque celles-ci sont encodées en se référant aux identificateurs et que l'identificateur de l'espace de nommage et de la classe sont identiques, la classe aura accès aux fins de méta-associations accessibles depuis l'espace de nommage.

[1] Chaque instance du classificateur spécifique [une classe par exemple] est aussi une instance indirecte du classificateur général.

[2] Le classificateur spécifique hérite donc des caractéristiques du classificateur général.

Ceci met en œuvre de manière tout à fait naturelle l'héritage des méta-associations (point 2 pour les fins d'association).

De même que les méta-généralisations, les relations de sous-ensemble entre fins de méta-associations n'ont pas d'instance dans le modèle. Par exemple, le sous-ensemble du métamodèle de la figure 4.3 décrit que la fin de méta-association `ownedAttribute` est un sous-ensemble de la fin de méta-association `ownedMember`. La spécification d'UML [58] en fournit la sémantique suivante :« *the collection associated with an instance of the subsetting property must be included in (or the same as) the collection associated with the corresponding instance of the subsetted property.*[3] » Il faut donc que les attributs d'une classe accessibles via la fin de méta-association `ownedAttribute` soient également accessibles par la fin d'association `ownedMember` de l'espace de nommage déduit de la classe. La sémantique de cette construction nous amène à définir la règle :

- Règle 2 : $ownedMember(IdClass, IdAttribute) : -$
  $ownedAttribute(IdClass, IdAttribute)$

qui représente que tout attribut d'une classe est aussi membre de cette classe.

Les faits représentent le modèle, c'est-à-dire les éléments et les relations du métamodèle qui ont été instanciés. À partir de ces faits, les règles qui encodent les constructions abstraites du métamodèle permettent de déduire les éléments instanciés de manière indirecte. La figure 4.5 montre ce mécanisme de déduction. Créer une classe dans le modèle revient à instancier la métaclasse `Class` mais aussi indirectement à instancier la métaclasse `NameSpace`. Cette déduction est réalisée par la règle 1. De la même manière créer un attribut (métaclasse `Property`) implique indirectement la création d'un élément nommé. Des règles du même type que la règle 1 opéreront ces déductions. Enfin les règles qui auront la même forme que la règle 2 permettront de déduire les relations entre ces éléments instanciés indirectement. La règle 2 exprime qu'une relation entre un espace de nommage et un de ses membres existe si une relation entre une classe et un attribut est présente dans le modèle. Ce mécanisme permet de faire des raisonnements

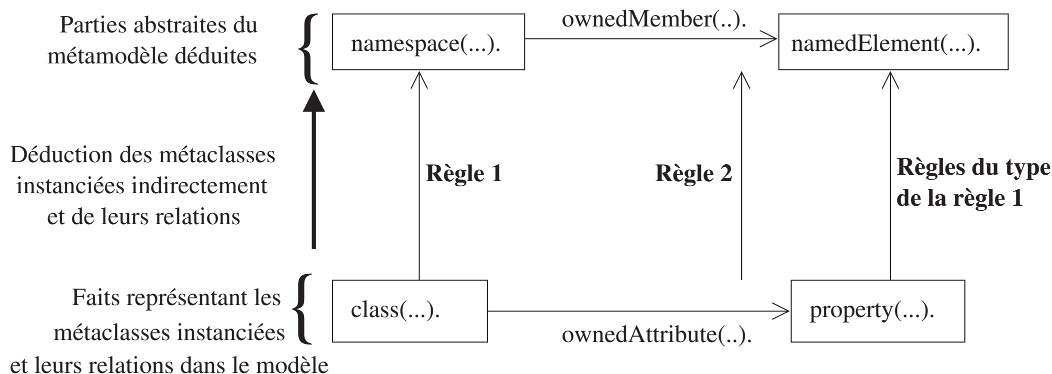


Fig. 4.5 – Mécanisme de déduction des parties du métamodèle non instanciées directement

[3]L'ensemble des éléments définis par une possession [une fin d'association navigable par exemple] sous-ensemble doit être inclus dans l'ensemble des éléments définis par la possession sur-ensemble.

sur l'ensemble du métamodèle et notamment sur ses parties abstraites ce qui est très utile pour la vérification de cohérence. Par exemple, ce mécanisme permet de formaliser et vérifier la contrainte « *les membres d'un espace de nommage doivent avoir des noms différents* ». Vérifier cette contrainte revient à vérifier l'ensemble des instances possibles de cette contrainte, par exemple « *deux attributs qui appartiennent à la même classe doivent avoir des noms différents* », « *deux classes qui appartiennent à un même paquetage doivent avoir des noms différents* », etc. Un exemple illustrant cette affirmation est fourni section 4.3.1. Outre la facilité de mise en œuvre, ce mécanisme garantit que les contraintes écrites sont appliquées exhaustivement sur tous les éléments du modèle auxquels elles se réfèrent.

4.2.2 Niveau méta-langage

Le métamodèle d'UML est décrit en MOF [56]. Comme vu en section 3.1.1, la technique de l'encodage est basée sur des schémas de transformation au niveau méta-langage. L'enjeu est donc d'exprimer des schémas de définition par des clauses (C)LP de toutes les constructions offertes par le MOF en retranscrivant de manière correcte leur sémantique. Cette partie est en fait une généralisation de ce qui a été présenté dans la section précédente.

L'expression du langage UML par un méta-langage a un autre avantage important : les futures évolutions du langage UML pourront être prises en charge par les mêmes schémas. L'outil de détection des incohérences pourra alors être conçu en les réutilisant. Cette possibilité est essentielle dans un contexte industriel tel que l'avionique. En effet, un modèle d'avion a une durée de vie de 40 à 50 ans ce qui est supérieur à la durée de vie d'une version d'UML.

Encore une fois, nous présentons séparément la formalisation des constructions concrètes du MOF et la formalisation des constructions abstraites.

Constructions concrètes du MOF Les constructions concrètes du MOF sont représentées par des méta-méta-faits qui sont instanciés en méta-faits en fonction du métamodèle (ou langage) à analyser.

La figure 4.6 montre la représentation par un méta-méta-fait de la construction offerte par les classes du MOF :

- MÉTA-MÉTA-FAIT 1 : $Meta_Classe_Name(Identificateur, A_1, ..., An)$.

où $Meta_Classe_Name$ est le nom de la métaclasse, $Identificateur$ est l'identificateur qui nous sert à identifier l'instance de la métaclasse et $A_1, ..., A_n$ représentent les attributs de la métaclasse. Ils sont calculés en tenant compte des attributs hérités. Cette figure montre également la relation d'instanciation entre les méta-méta-faits et les méta-faits qui correspond à définir le nom, l'arité du méta-fait et les noms des différents paramètres.

Constructions abstraites du MOF Les constructions abstraites sont représentées par des formats de règles (appelées méta-règles) qui sont ensuite instanciées en règles en fonction du métamodèle à analyser.

Par exemple, les méta-généralisations mettent en relation deux métaclasses. Elles doivent rendre compte de la projection des caractéristiques de la métaclasse spéciale vers

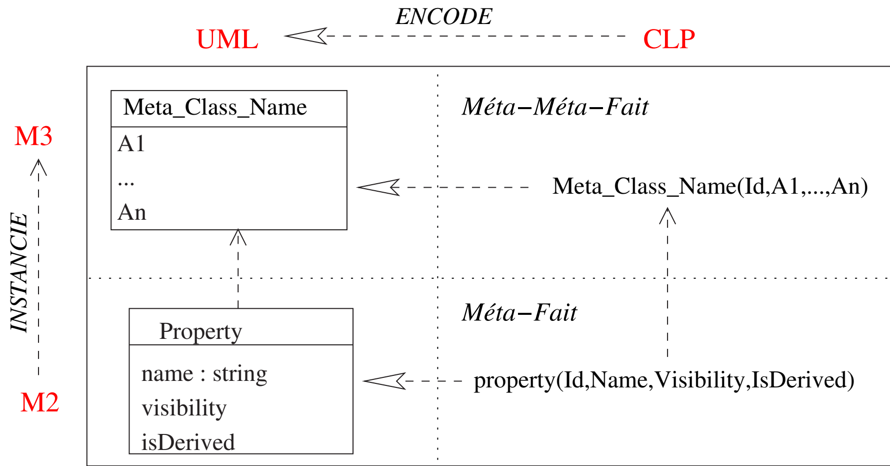


FIG. 4.6 – Formalisation des classes du MOF

la métaclasse générale et doivent mettre en œuvre le principe d'héritage des fins d'associations (en donnant le même identificateur aux deux métaclasses mises en relation). Toute méta-généralisation pourra donc être représentée par une règle qui suit le format :

- MÉTA-RÈGLE 1 : $General_Class(Id, A_1, ..., A_n) : -Special_Class(Id, B_1, ..., B_m)$.

où $General_Class$ et $Special_Class$ sont respectivement les noms de la métaclasse générale et de la métaclasse spécifique, Id est l'identificateur de l'élément spécialisé et de l'élément général. $A_1, ..., A_n$ représentent les attributs de l'élément général et $B_1, ..., B_m$ représentent les attributs de l'élément spécifique. Deux cas de figure sont différenciés pour définir ces derniers. Si le méta-attribut de la classe spécifique est également un méta-attribut de la classe générale alors le méta-attribut apparaît tel quel dans la méta-règle. Si le méta-attribut n'apparaît que dans la classe spécifique alors sa valeur nous est indifférente. De manière plus formelle : soit $General_Class(Id, A_1, ..., A_n)$ et $Special_Class(Id, C_1, ..., C_m)$ respectivement les méta-faits de la classe générale et spécifique on a : si $\exists i, j | A_i = C_j => B_j = C_j$ sinon $B_j =_$. Par construction on a forcément $m \geq n \wedge \forall i \in \{1..n\}, \exists j \in \{1..m\} | C_i = B_j$ car tout attribut de la classe générale est hérité par la classe spécifique.

La figure 4.7 donne une vue graphique de cette représentation. Cette figure se limite au cas où la classe spéciale n'hérite que d'une seule classe.

Cette méta-règle est ensuite instanciée en fonction du métamodèle à analyser. Par exemple la règle 1 de la section 4.2.1 rend compte de la spécialisation entre les classes et les espaces de nommage et est une instance de cette méta-règle. L'instanciation d'une méta-règle en règle consiste à définir pour chaque prédicat rencontré son nom, son arité et les noms de ses paramètres.

4.2.3 Synthèse

Afin de donner au lecteur une vue d'ensemble de la représentation des modèles UML en (C)LP, le tableau 4.1 présente l'encodage d'un modèle. Le modèle et le métamodèle considérés sont respectivement ceux des figures 4.2 et 4.3. Le métamodèle permet de déduire les artefacts produits au niveau M2, c'est-à-dire :

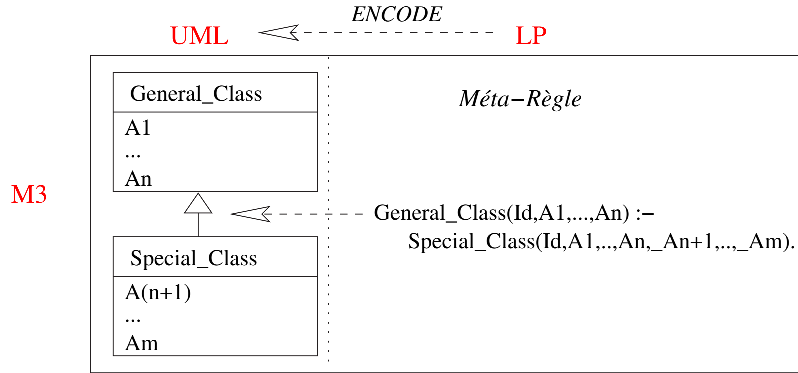


FIG. 4.7 – Représentation des méta-généralisations

1. les méta-faits qui encodent les métaclasses (colonne 1 du tableau 4.1) ;
2. les méta-faits qui encodent les fins d'association navigables (colonne 2 du tableau 4.1) ;
3. les règles qui permettent de déduire les éléments instanciés indirectement dans le modèle (colonne 3 du tableau 4.1) ; ces règles sont déduites à partir des méta-généralisations ;
4. les règles qui permettent de déduire les relations du métamodèle instanciées indirectement (colonne 4 du tableau 4.1) ; ces règles découlent des relations de sous-ensemble entre fins de méta-associations.

La production des faits est ensuite réalisée en instanciant les méta-faits produits au niveau M2 en fonction du modèle. Ces faits se situent au niveau M1 et deux types peuvent être distingués :

1. les faits qui représentent les éléments du modèle : ils sont obtenus en instanciant les méta-faits qui encodent les métaclasses en fonction du modèle (colonne 1) ;
2. les faits qui représentent les relations entre les éléments : ils sont obtenus en instanciant les fins de méta-associations navigables en fonction du modèle (colonne 2).

Concernant l'outillage, seuls les faits de niveau M1 (colonnes 1 et 2) et les règles de niveau M2 (colonnes 3 et 4) sont pris en compte.

	Encodage des éléments du modèle (méta-classes)	Encodage des relations du modèle (fins de méta-association navigables)	Encodage des méta-généralisations, déduction des éléments instanciés indirectement	Encodage des relations de sous-ensembles entre fins de méta-associations, déduction des relations entre éléments non directement instanciés
M3	$MetaClassName(Id, A_1, .., A_n).$	$AssociationEndName(Id1, Id2).$	$General_Class(Id, A_1, ..., A_n) :- Special_Class(Id, B_1, ..., B_m).$	$SubsettedAsso(Id1, Id2) :- SubsettingAsso(Id1, Id2).$
M2	$namedElement(Id, Name, Visibility).$ $nameSpace(Id, Name, Visibility).$ $class(Id, Name, Visibility, IsAbstract).$ $feature(Id, Name, Visibility).$ $structuralFeature(Id, Name, Visibility).$ $property(Id, Name, Visibility, IsDerived).$	$nameSpace(IdNamedElement, IdNameSpace).$ $ownedMember(IdNameSpace, IdNamedElement).$ $ownedAttribute(IdClass, IdProperty).$ $class(IdProperty, IdClass).$	$namedElement(Id, Name, Visibility) :- nameSpace(Id, Name, Visibility).$ $nameSpace(Id, Name, Visibility) :- class(Id, Name, Visibility, _).$ $namedElement(Id, Name, Visibility) :- feature(Id, Name, Visibility).$ $feature(Id, Name, Visibility) :- structuralFeature(Id, Name, Visibility).$ $structuralFeature(Id, Name, Visibility). property(Id, Name, Visibility, _).$	$nameSpace(Id1, Id2) :- class(Id1, Id2).$ $ownedMember(Id1, Id2) :- ownedAttribute(Id1, Id2).$
M1	$class(idClass1, classA, public, false).$ $class(idClass2, classB, public, false).$ $property(idAtt1, att_in_double, public, false).$ $property(idAtt2, att_in_double, public, false).$ $property(idAtt3, att_OK, public, false).$ $property(idAtt4, att_OK, public, false).$	$class(idAtt1, idClass1).$ $class(idAtt2, idClass1).$ $class(idAtt3, idClass1).$ $class(idAtt4, idClass2).$ $property(idClass1, idAtt1).$ $property(idClass1, idAtt2).$ $property(idClass1, idAtt3).$ $property(idClass2, idAtt4).$		

Tab. 4.1 – Encodage du modèle de la figure 4.2 et du métamodèle de la figure 4.3

4.2.4 Outillage et conclusion

Cette section décrit brièvement l'outil développé permettant d'encoder des modèles UML en LP et fournit un certain nombre de conclusions sur l'encodage proposé.

L'outil est paramétré par deux fichiers au format XMI [60] (cf. « Données d'Entrée » à la figure 4.8).

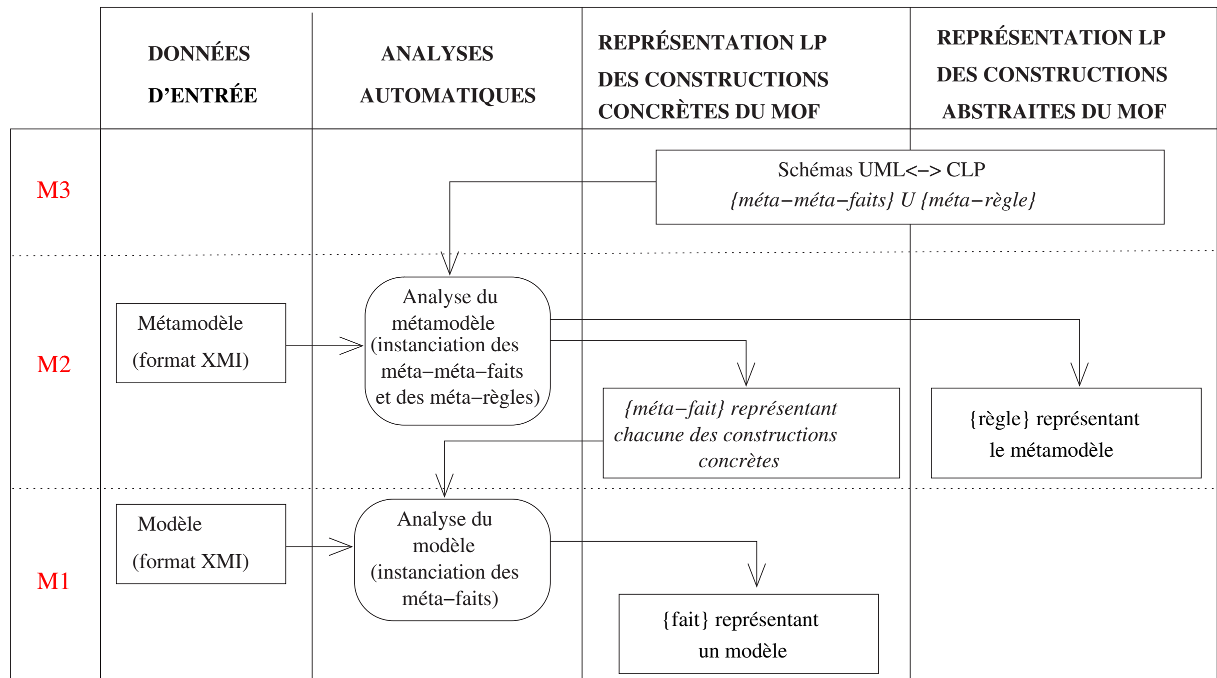


FIG. 4.8 – Processus d'encodage des modèles et du métamodèle

Le premier fichier est le métamodèle considéré. L'analyse de ce dernier permet de construire l'ensemble des méta-faits destinés à être instanciés et l'ensemble des règles LP qui représentent ce métamodèle. Le deuxième fichier est le modèle UML à analyser. Celui-ci permet d'instancier les méta-faits déduits du métamodèle en faits qui encodent le modèle à analyser.

Cet outil a été développé sous la plate-forme Eclipse [24] en Java. Il utilise l'outil d'analyse de fichiers XML Dom4J [22] qui inclut une implantation de XPath (XML Path Language [82]). Le métamodèle utilisé est celui fourni par l'outil UML2 d'Eclipse [80]. Plus de détails concernant l'outil de traduction des modèles UML en (C)LP sont donnés en annexe B.

Pour conclure, un encodage des modèles UML en LP a été défini. L'ensemble des informations d'un modèle UML est présent dans son encodage en LP. Comme défini en section 3.1.1, l'encodage est réalisé par des schémas de correspondance au niveau M3 (méta-langage), c'est ce qui a été réalisé entre le MOF et LP. LP ne disposant pas de méta-langage, ces schémas sont abstraits et ne font pas partis de la représentation du modèle et du métamodèle. Ces schémas sont ensuite instanciés en fonction du métamodèle puis du modèle considéré. Concrètement un modèle UML est représenté par des faits. Certains faits représentent les éléments du modèle (instances des métaclasses), et d'autres les relations entre ces éléments (instances de fin de méta-associations). Le

métamodèle est représenté par des règles LP qui permettent de reconstruire les parties plus abstraites du métamodèle à partir des faits du modèle. Cette reconstruction est réalisée par la formalisation des méta-généralisations et des relations d'ensemble entre fins de méta-associations.

L'intérêt d'une telle représentation est de pouvoir profiter du pouvoir d'analyse des (C)LP et permet d'effectuer des raisonnements au niveau métamodèle (et éventuellement sur les parties « abstraites » de ce métamodèle) valables pour tout modèle UML.

Nous utilisons ces possibilités de raisonnements pour la vérification de cohérence structurelle (section suivante) et comportementale (chapitre suivant).

4.3 Formalisation des règles de cohérence et diagnostic

Après avoir traduit en (C)LP le modèle UML étudié, il convient de formuler en (C)LP les contraintes que les modèles doivent respecter (étape 2 de la figure 4.1). En fait, nous exprimons leur négation, à savoir l'incohérence associée à la contrainte. Ainsi, si le solveur (C)LP trouve une solution à l'incohérence sur le modèle encodé en (C)LP, ceci démontre la présence de cette incohérence dans le modèle UML. La substitution des variables du but fournie par le système logique en réponse de la résolution de celui-ci constitue le diagnostic de l'incohérence.

La section 4.3.1 fournit un exemple de formalisation d'une règle de cohérence. La section 4.3.2 présente ensuite des règles supplémentaires qui facilitent l'expression des incohérences. Enfin, la section 4.3.3 s'intéresse à la génération automatique des règles de cohérence qui assurent le respect du métamodèle.

4.3.1 Exemple

La négation de la règle stipulant qu'un espace de nommage ne doit pas contenir deux membres de même nom est énoncée par : [4]

$incoherence(1, IdNameSpace, IdMember1, IdMember2) :-$	
(1)	$nameSpace(IdNameSpace, _, _),$
(2)	$ownedMember(IdNameSpace, IdMember1),$
(3)	$ownedMember(IdNameSpace, IdMember2),$
(4)	$namedElement(IdMember1, Nom1, _),$
(5)	$namedElement(IdMember2, Nom2, _),$
(6)	$Nom1 = Nom2,$
(7)	$IdMember1 \neq IdMember2.$

Cette règle est proche de celle présente en partie 4.1. La principale différence vient du contexte d'écriture de la règle qui remplace les classes par les espaces de nommage.

Nous appliquons maintenant la règle (C)LP de détection d'incohérence aux faits et règles d'encodage présentés par le tableau 4.1. Le diagnostic est fourni par l'exécution du but :

[4] La règle présente dans la norme stipule que deux membres sont indistinguables s'ils sont de même nom et de même type, nous nous limitons au nom pour plus de clarté.

`?-incoherence(1, IdNameSpace, IdMember1, IdMember2)` dont la réponse est :
`IdNameSpace = idClass1`
`IdMember1 = idAtt1`
`IdMember2 = idAtt2`

Ce qui signifie que l'espace de nommage d'identificateur `idClass1` contient deux membres qui ont le même nom et d'identificateurs respectifs `idAtt1` et `idAtt2` ; ce qui est bien le cas dans le modèle. Le système fournit en diagnostic l'ensemble des éléments mis en jeu dans l'incohérence. Ces informations obtenues de manière automatique permettent de mettre en œuvre un traitement adapté pour l'incohérence détectée. Notons que ce diagnostic est meilleur qu'avec l'utilisation d'OCL car ce langage ne permet que de fournir l'élément contexte de la règle, les autres éléments n'étant pas visibles à l'extérieur de celle-ci. Par exemple, la règle qui exprime qu'un espace de nommage ne doit pas contenir deux membres qui ne sont pas distinguables est formalisée comme suit dans [59] :

```
context Namespace inv ::
let membersAreDistinguishable =
    self.member → forAll( memb |
        self.member → excluding(memb) → forAll(other |
            memb.isDistinguishableFrom(other, self))) in
self.membersAreDistinguishable()
```

Le diagnostic fournit par la vérification de la règle par les outils est alors uniquement l'élément contexte, ici l'espace de nommage. Sur l'exemple du modèle considéré, le diagnostic se résumerait donc à l'identificateur de la classe nommée `classeA`.

Cet exemple illustre la possibilité de réaliser des raisonnements sur les parties abstraites du métamodèle. Ces raisonnements sont ensuite hérités au travers des méta-généralisations. En effet, cette règle est écrite pour les espaces de nommage et détecte des incohérences sur des classes. Ceci est réalisé de manière tout à fait naturelle et découle de la formalisation du métamodèle présentée à la section précédente. En effet, la règle 1 exprime qu'une classe est un espace de nommage et la règle 2 qu'un attribut d'une classe est membre de l'espace de nommage correspondant (cf. section 4.2.1.2).

4.3.2 Règles supplémentaires facilitant l'expression des incohérences

Afin que notre outil puisse être adopté par la communauté, nous devons faciliter au maximum l'écriture des règles de cohérence. Nous présentons dans un premier temps, une manière d'alléger la manipulation des prédicats formalisant les éléments du modèle. Nous présentons ensuite des clauses LP qui fournissent des fonctionnalités utiles à l'expression des incohérences comme par exemple des opérations ensemblistes.

Faciliter la manipulation des éléments du modèle Les faits qui représentent les éléments du modèle peuvent contenir un grand nombre de paramètres. Par exemple, l'arité du méta-fait représentant une classe est 7 :
$class(Id, IsAbstract, IsActive, IsLeaf, Name, QualifiedName, Visibility)$.

Il est donc important de faciliter la manipulation de ces faits. Deux points sont traités, d'une part l'accès aux méta-attributs et d'autre part le test du type d'un élément.

Vu le méta-fait qui les représente, il est possible d'accéder au nom d'une classe Id en écrivant « $class(Id, _, _, _, Name, _, _)$. » ce qui nécessite de connaître l'arité du fait, et la position du méta-attribut `Name`. Or, ces informations ne sont pas forcément connues de l'utilisateur. C'est pourquoi, lors de l'analyse du métamodèle, des clauses permettant d'accéder facilement aux différentes caractéristiques sont générées. Par exemple, la clause « $getName(Id, Name) : -namedElement(Id, Name, _)$. » permet de récupérer le nom de tout élément nommé d'identificateur `Id` (ou inversement). Or, nous avons vu que la présence d'un élément nommé est déduite lorsque que cette métaclasse est instanciée indirectement, par exemple lorsqu'une classe est créée. Cette règle permet donc d'accéder au nom d'une classe. Ce type de règle est généré à chaque déclaration dans le métamodèle d'un nouveau méta-attribut.

D'autre part, tester le type d'un élément est aussi très utile à l'expression des incohérences. Par exemple, pour sélectionner les éléments qui sont des classes on écrira « $class(Id, _, _, _, _, _, _)$ ». Encore une fois ceci nécessite de connaître le méta-fait de représentation des classes. Pour chaque métaclasse, des règles du type « $isClass(Id) : -class(Id, _, _, _, _, _, _)$. » sont donc générées. Tester si un identifiant est l'identifiant d'une classe est donc possible par « $isClass(Id)$. ».

Fournir des fonctionnalités avancées L'écriture de règles de cohérence est grandement facilitée par l'utilisation de fonctionnalités avancées. OCL (Object Constraint Language) [57] offre un certain nombre de ces fonctionnalités sous forme d'opérateurs prédéfinis. Pour que notre outil soit intéressant il faut fournir ce type d'opérateurs.

Nous pensons que LP est bien adapté à l'écriture de ces fonctionnalités et il nous semble que l'ensemble des opérateurs encodés en OCL peut l'être en LP. Cependant, aucune étude exhaustive n'a pour le moment été menée.

Nous illustrons maintenant comment encoder par des règles LP, certains des opérateurs d'OCL. L'écriture des opérateurs classiques sur les ensembles comme l'union, l'intersection est trivial, certains de ces opérateurs sont d'ailleurs fournis par défaut par les systèmes de LP. L'écriture des opérateurs OCL `Select, Reject, Collect, Exists` et `ForAll` nécessitent une explication. Leur encodage est réalisé au moyen de termes Hilog (*Higher-Order Logic Programming* [14]) qui permettent une forme de programmation d'ordre supérieur [70], c'est-à-dire que des noms de prédicats peuvent être variables. Par exemple, l'opérateur `Select` prend une collection d'éléments en entrée et fournit en sortie une sous-collection contenant les éléments qui respectent une condition. Cette opérateur inspiré d'OCL est encodé comme suit :

$$
\begin{array}{lr}
: -hilog\ selection. & (1) \\
 & \\
select(_)([], []). & (2) \\
select(Predicat)([X|L], Res) : - & (3) \\
\qquad select(Predicat)(L, R), & (4) \\
\qquad (Predicat(X) - > & (5) \\
\qquad\qquad Res = [X|R] & (6) \\
\qquad\qquad Res = R). & (7)
\end{array}
$$

où :
- `(1)` spécifie que `select` est un terme Hilog ;
- `(2)` spécifie qu'une opération de sélection sur une liste vide fournit une liste vide ;
- on spécifie ensuite que le résultat d'une sélection sur une collection correspond à la sélection de cette collection auquel on a enlevé le premier élément `(3 et 4)`, auquel il faut ajouter l'élément supprimé si et seulement si cet élément respecte le prédicat (`(5, 6 et 7)`).

Par exemple, soit L une liste contenant les identificateurs d'éléments d'un modèle, appliquer le but `select(isClass)(L,LRes)` fournit dans la variable `LRes` les seuls éléments de `L` qui sont des classes.

De même la fermeture transitive (qui évite les cycles) a été définie par la règle :

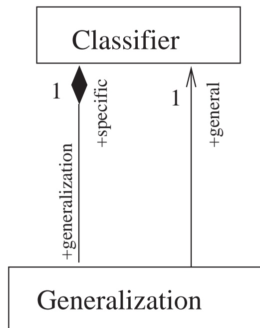
$$
\begin{array}{lr}
:-hilog\ closure_aux. & \\
:-table\ apply/3. & (1) \\
 & \\
closure(Predicat, X, Liste) :- & \\
\quad\quad findall(Elem, & (2) \\
\quad\quad\quad\quad closure_aux(Predicat)(X, Elem), & \\
\quad\quad\quad\quad Liste). & \\
 & \\
closure_aux(Predicat)(X, Y) :- Predicat(X, Y). & (3) \\
closure_aux(Predicat)(X, Y) :- Predicat(X, Int), closure_aux(R)(Int, Y). & (4)
\end{array}
$$

où :
- `(1)` définit que le principe de tabulation sera utilisé pour éviter les cycles ;
- `(2)` définit la fermeture transitive en formant la liste de tous les éléments $Elem$ qui satisfont le prédicat $closure_aux(P)(X, Elem)$;
- la fermeture transitive est ensuite appliquée de la même manière que le prédicat `reach/2` (cf. section 3.2.2.1) excepté que le prédicat appliqué est maintenant passé en paramètre.

Pour illustrer cette fonctionnalité, la figure 4.9 présente la règle qui exprime qu'un arbre de généralisation est acyclique ainsi que le métamodèle sur lequel cette règle s'applique. Le prédicat `parent` donne à partir d'un classificateur spécifique un de ses parents par navigation dans le métamodèle. La ligne `3` exprime que cette incohérence s'applique sur les classificateurs. La ligne `4` construit transitivement la liste `Liste_Resultat` de tous les parents du premier paramètre `Id_Element`. La règle stipule ensuite qu'il y a incohérence dans le cas où un élément appartient à la liste de ses parents (ligne `5`). La liste des parents est obtenue par l'application du prédicat de fermeture transitive. Notons qu'un des apports de [7] est la proposition d'ajouter l'opérateur de fermeture transitive à OCL, celui-ci étant absent de la norme [57]. Nous n'avons cependant pas connaissance de l'outillage correspondant.

4.3.3 Règles vérifiant le respect du métamodèle

Le métamodèle définit le langage UML. Pour qu'un modèle UML soit bien formé il faut que ce modèle respecte les contraintes imposées par le métamodèle. Nous proposons donc d'exprimer explicitement les contraintes intrinsèques au MOF. Le respect du métamodèle est à la charge des outils de modélisation UML. La conformité au métamodèle




```
parent(IdClassifierSpecial,IdClassifierGeneral) :-
(1) generalization(IdClassifierSpecial,IdGene),
(2) general(IdGene,IdClassifierGeneral).

incoherence(4,IdElement,Liste_Resultat) :-
(3) isClassifier(IdElement),
(4) closure(parent,IdElement,Liste_Resultat),
(5) member(IdElement,Liste_Resultat).
```

Fig. 4.9 – Généralisations acycliques

est un critère important dans le choix d'un outil. C'est elle qui permet d'échanger des modèles entre outils, de définir des transformations de modèles, etc.

Deux types de contraintes imposées par le métamodèle ont été identifiés :

- les contraintes liées aux multiplicités du métamodèle ; par exemple, une généralisation doit référencer exactement un classificateur général et un classificateur spécifique (cf. figure 4.9) ;
- les contraintes liées aux types des éléments mis en relation par une méta-association ; par exemple « *les éléments mis en relation par une généralisation sont des classificateurs* » (cf. figure 4.9).

Ces règles peuvent être générées automatiquement lors de l'analyse du métamodèle considéré. Nous fournissons un exemple de chacune de ces règles afin d'illustrer notre discours et montrer comment la génération automatique de ces règles est possible.

Contrainte sur les multiplicités La règle suivante formalise l'incohérence associée à la règle « une généralisation a exactement 1 élément général » :

$incoherence_meta(1, Id_Generalization, Liste_Elem_General) :-$
(1) $isGeneralization(Id_Generalization),$
(2) $findall(X, general(Id_Generalization, X),$
(3) $Liste_Elem_General),$
(4) $length(Liste_Elem_General, NombreDElemGeneraux),$
(5) $NombreDElemGeneraux \neq 1.$

(1) exprime que la règle se base sur les généralisations (relation d'héritage). (2) et (3) construisent la liste de tous les éléments généraux de la généralisation (par navigation dans le métamodèle). (4) affecte à $NombreDElemGeneraux$ le nombre d'éléments généraux associés à la généralisation considérée. Enfin, (5) exprime qu'une incohérence

est présente si ce nombre est différent de 1.

Le même type de règle peut être généré dès que la multiplicité est différente de *. En fait on peut écrire des règles génériques que l'on instancie à chaque fois que l'on trouve une fin d'association dont la multiplicité est différente de *. Cette règle est en fait paramétrée par le type de l'élément (`Generalization`), le nom de la fin d'association (`general`) et la contrainte de multiplicité ($\neq 1$).

Contrainte sur le type des éléments mis en jeu par une méta-association La règle exprimant qu'une généralisation doit mettre en jeu des classificateurs comme exprimé par la figure 4.9 est formalisée de la manière suivante :

$incoherence_meta(3, Id_Gene, IdElementGeneralPasClassifier) :-$	
(1)	$isGeneralization(Id_Gene, _, _, _),$
(2)	$general(Id_Gene, IdElementGeneralPasClassifier),$
(3)	$not(isClassifier(IdElementGeneralPasClassifier)).$

Là aussi il est possible d'écrire des règles génériques destinées à être instanciées. Les paramètres sont ici le type de l'élément source (`Generalization`), le type de l'élément cible (`Classifier`) et la fin d'association navigable (`general`).

Ce type de règles, bien qu'intéressantes ne sont pas générées dans la version actuelle du prototype. En pratique leur génération semble cependant tout à fait réalisable.

4.4 Conclusion

L'encodage des modèles UML proposé dans ce chapitre permet d'effectuer des raisonnements sur les modèles UML en (C)LP. Cet encodage est défini au niveau M3 et est calqué sur les techniques de métamodélisation. Il est donc indépendant du langage considéré (UML2 dans notre cas). Ceci autorise son utilisation pour tout langage défini en MOF.

Les possibilités de raisonnement sont utilisées pour vérifier la cohérence de la structure des modèles UML. Plusieurs avantages sont à souligner. D'une part, il est possible de réaliser des raisonnements sur les parties abstraites du métamodèle, les raisonnements étant ensuite hérités sur tous les éléments sur lesquels ils peuvent être appliqués. La substitution fournie par le joueur (C)LP identifie l'ensemble des éléments mis en jeu dans l'incohérence. Le diagnostic sur le modèle UML est aisé car les éléments manipulés sont ceux du modèle. Ceci facilite la mise en place de traitements spécifiques à chaque incohérence.

Les possibilités en terme de vérification ne se limitent pas aux règles de cohérence. Par exemple [61] décrit l'utilisation des (C)LP pour vérifier des propriétés extra-fonctionnelles grâce au raisonnement symbolique permis par les (C)LP. Le mécanisme introduit dans [77] permet de décrire des choix architecturaux d'un élément d'un modèle en accédant à son métamodèle. Ce mécanisme est appelé mécanisme d'introspection. OCL a été étendu avec un tel mécanisme. Notre encodage permet d'écrire de telles propriétés en écrivant des règles qui s'appliquent sur une classe particulière plutôt que sur toutes les classes.

D'autre part, il semble que l'expressivité de notre outil soit au moins équivalente à celle d'OCL. Une étude exhaustive doit cependant être réalisée.

Dans le chapitre qui suit, nous proposons de tirer parti de l'encodage défini précédemment pour vérifier des propriétés comportementales sur les modèles UML.

Chapitre 5

Détection des incohérences comportementales

Le chapitre précédent décrit l'encodage de modèle UML en (C)LP. Cet encodage tire parti du métamodèle d'UML. La description formelle fournie par celui-ci se cantonne à la structure des modèles. L'encodage proposé ne permet ainsi que l'écriture et la vérification de règles de cohérence ne faisant intervenir que la structure des modèles.

UML est dit unifié car il permet entre autres de représenter le comportement des systèmes dynamiques. Un système est dynamique s'il évolue au cours du temps. Les systèmes logiciels critiques sont de tels systèmes car en fonction des événements reçus ceux-ci doivent réagir en réalisant les actions voulues (dans un temps contraint pour les systèmes temps-réels). Vérifier le bon comportement de ces systèmes est donc essentiel pour apporter les garanties nécessaires. Aussi, valider le comportement des modèles permet d'apporter certaines de ces garanties et la détection en amont d'erreurs comportementales éventuelles.

Le but de ce chapitre est de montrer comment notre méthode permet de telles validations en nous concentrant sur les règles de cohérence.

Détecter automatiquement des incohérences comportementales nécessite de spécifier formellement le comportement des modèles UML, c'est-à-dire la sémantique opérationnelle des différentes constructions du langage UML. Le comportement dynamique des systèmes discrets est fréquemment décrit par des règles de changement de configuration [47]. Nous adoptons ici cette approche en les exprimant au moyen de règles (C)LP. Le comportement des systèmes discrets est en effet défini par un ensemble de traces [17]. Une trace représente l'évolution des configurations du système au fur et à mesure de son cheminement. Chaque configuration contient les valeurs instantanées qui caractérisent l'état du système, par exemple pour un programme informatique ces informations ont trait aux variables du programme, à l'état de la mémoire, etc. Il existe des traces de longueur finie et de longueur infinie. Une trace finie est notée par $\sigma = \sigma_0..\sigma_{n-1}$ (sa longueur est $|\sigma| = n$), une trace infinie est notée $\sigma = \sigma_0\sigma_1..$ (de longueur $|\sigma| = \infty$). L'ensemble possible des évolutions du système est représenté par le calcul de l'ensemble des traces possibles. Notre but est donc d'obtenir cet ensemble de traces pour tout modèle UML étudié. Pour cela il faut que nous définissions les valeurs qui caractérisent les configurations d'un modèle UML. Il sera alors possible d'exprimer les règles de changement de configuration notées t qui permettent de calculer le passage d'une configuration σ_i à une

configuration σ_{i+1}, ce qui se note $t =< \sigma_i, \sigma_{i+1} >$. Obtenir ces traces nous permettra ensuite d'exprimer les propriétés que le modèle UML doit respecter, les règles de cohérence dans notre cas.

Le présent chapitre est organisé comme suit. La section 5.1 donne une vue globale et intuitive de notre approche. Nous décrivons ensuite la méthode pour exprimer la sémantique opérationnelle d'UML en définissant :

– les informations contenues dans la configuration σ_i d'un modèle UML, c'est-à-dire les informations caractérisant l'état dynamique des modèles UML (cf. section 5.2) ;
– la sémantique opérationnelle au travers de règles qui expriment les évolutions possibles d'une configuration à une autre (cf. section 5.3) ; ceci définit l'ensemble des transitions possibles t du système à partir d'une configuration σ_i telle que $< \sigma_i, \sigma_{i+1} >\in t$; cette section définit également les éléments d'UML qui influencent de manière active l'évolution du comportement des modèles, c'est-à-dire les éléments d'UML dont il faut décrire la sémantique opérationnelle.

Enfin, la section 5.4 montre comment le travail présenté au préalable permet d'exprimer les règles de cohérence comportementale.

Limites d'implantation Le langage UML est extrêmement riche et il semble impossible de donner une sémantique formelle à l'ensemble de ses constructions dans le cadre de cette thèse. Notre méthode a cependant pour but de permettre une telle expression ainsi que son implantation. Afin de montrer la pertinence notre méthode, nous nous limitons donc à un sous-ensemble d'UML. Ce sous-ensemble comprend les diagrammes de classes ainsi que les diagrammes de machines à états. Les exemples fournis dans ce chapitre auront donc trait à ces diagrammes. Les constructions exactes prises en compte par notre outil seront détaillées au chapitre suivant.

5.1 Vue d'ensemble de l'approche

Cette section donne une vue d'ensemble intuitive de notre approche pour la détection d'incohérences comportementales sur les modèles UML. Cette approche est explicitée aux sections 5.2, 5.3 et 5.4.

La figure 5.1 montre la démarche de la mise en œuvre du vérificateur basé sur (C)LP. L'étape A de cette figure représente la formalisation en (C)LP de la structure d'un modèle et de son métamodèle par des clauses (C)LP. Cette étape est la première pour la mise en œuvre du vérificateur unifié. Elle a été présentée au chapitre précédent. La prise en compte des règles de cohérence comportementale nécessite l'expression du comportement des modèles UML. Comme mentionné en introduction, la dynamique des systèmes discrets est fréquemment décrite par des règles de changement de configuration [47]. Nous adoptons ici cette approche. Pour décrire la dynamique des modèles UML, il faut donc spécifier ce qu'est une configuration. C'est le but de l'étape B.1. Le comportement des modèles UML peut ensuite être décrit par la spécification des changements de configuration. Nous proposons de les formaliser en (C)LP ce qui constitue l'étape B.2. L'étape C consiste à spécifier en (C)LP les contraintes à respecter, c'est-à-dire les règles de cohérence. Une fois ces trois étapes effectuées, le joueur (C)LP peut fournir un diagnostic sur la cohérence du modèle UML formalisé lors des étapes A et B vis-à-vis des contraintes exprimées lors de l'étape C.

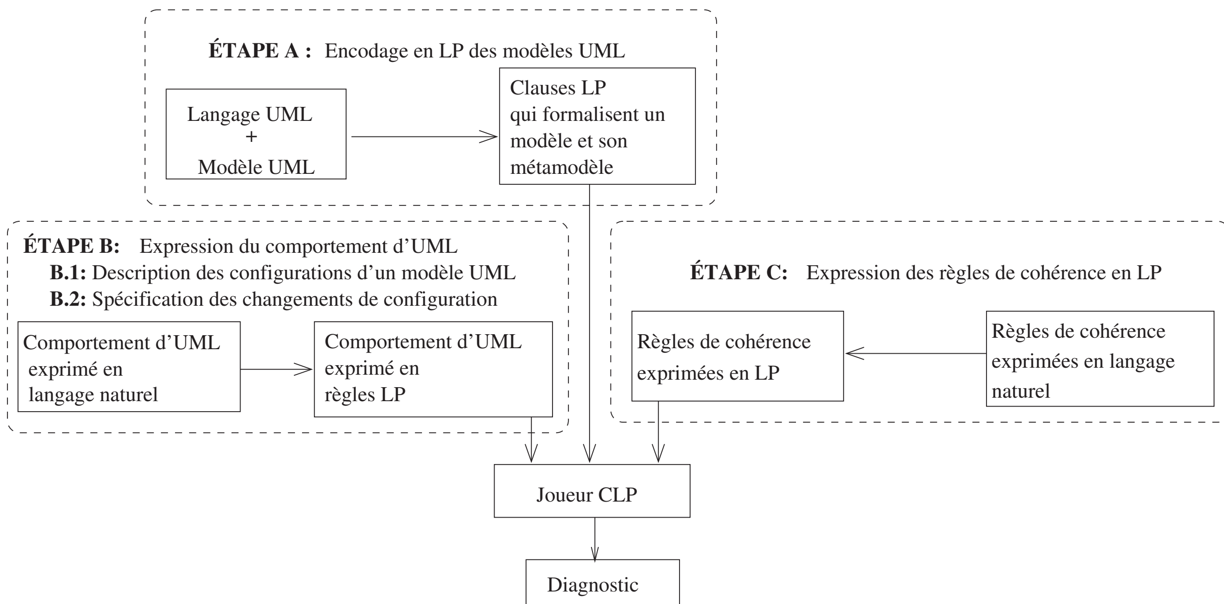


Fig. 5.1 – Vue d'ensemble de notre approche

Nous présentons maintenant un exemple sur lequel nous appliquons notre approche de manière intuitive. Le reste du chapitre systématise les différentes étapes afin de détecter les incohérences des modèles UML de manière automatique.

5.1.1 Étape A

La figure 5.2 présente un exemple d'encodage de la structure d'un modèle UML en (C)LP (étape A). La méthode d'obtention de l'encodage est décrite au chapitre précédent. Son principe consiste à décrire tout modèle UML par les éléments qui le composent d'une part et par les relations existantes entre ces éléments d'autre part. Par exemple le fait `state(idA,a)` décrit la présence dans le modèle d'un état nommé `a` et identifié par `idA`. Le fait `target(idT3,idA)` indique que la cible de la transition identifiée par `idT3` est l'état identifié par `idA`. Cet encodage rend compte de l'intégralité de la structure du modèle UML.

Cet exemple a été choisi pour montrer que les diagrammes comportementaux d'UML sont constitués en premier lieu d'éléments structurels. Les états et les transitions en sont des exemples pour les diagrammes de machines à états.

5.1.2 Étape B

Comme exprimé plus haut, l'étape B consiste à fournir une description du comportement des modèles UML. La méthode de description est nouvelle et est l'objet central de ce chapitre.

Le but de l'étape B.1 est de compléter la description de la structure des modèles UML établie lors de l'étape A en fournissant les informations sur la configuration du modèle. Considérons l'exemple de modèle de la figure 5.2. Ce modèle peut être dans plusieurs configurations. La figure 5.3 présente deux de ces configurations et propose

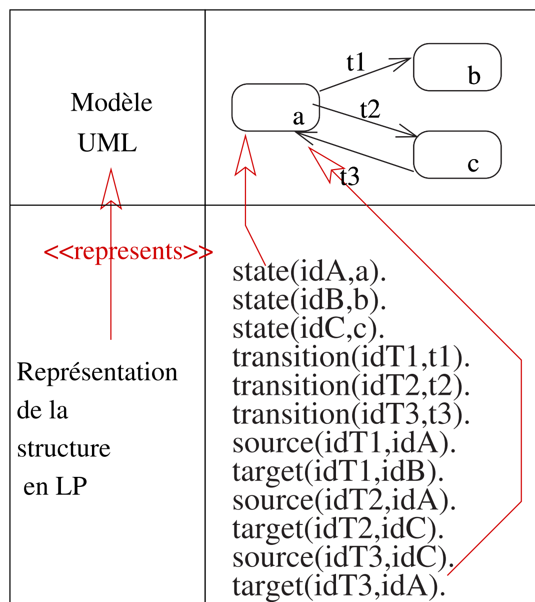


FIG. 5.2 – Représentation en (C)LP des informations structurelles d'un modèle UML

également une représentation en (C)LP de chacune de ces configurations. Chaque état est représenté par un doublon dont le premier terme est l'identifiant de l'état et le second sa configuration. Tous les états présentés étant simples, leur configuration est soit active (valeur 1) soit inactive (valeur 0). La configuration globale de ce modèle (ici une machine à états) est la liste de ces doublons. Nous verrons par la suite comment nous pouvons représenter la configuration d'états non simples. Ces informations ne sont pas formalisées dans le métamodèle actuel. La section 5.2 détaille notre proposition pour décrire ces informations.

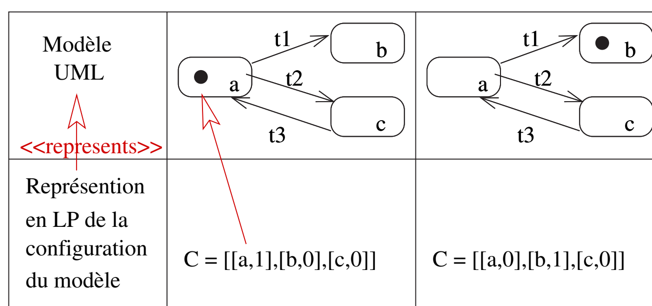


FIG. 5.3 – Représentation en (C)LP des informations dynamiques d'un modèle UML

Afin d'exprimer la sémantique opérationnelle d'UML, nous proposons de décrire lors de l'étape B.2 les changements de configuration en (C)LP. La figure 5.4 donne la règle `transStateMachine` décrivant le changement de configuration dû au tir d'une transition dans un diagramme de machines à états. La règle exprime qu'une transition peut être tirée si :

– `T` est une transition et donc si `isTransition(T)` est vrai ;
– `S1` et `S2` sont respectivement les sommets source et cible de `T`, et donc si

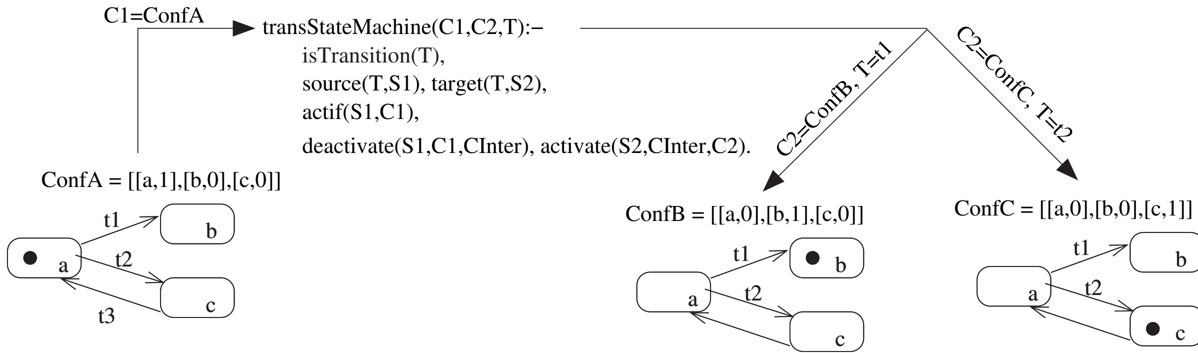


Fig. 5.4 – Expression en (C)LP des règles de changement de configuration

`source(T,S1)` et `target(T,S2)` sont vrais ; ces prédicats sont issus de la description structurelle des modèles UML en (C)LP (étape A) ;
– si `S1` est actif dans la configuration `C1` et donc si `actif(S1,C1)` est vrai ;
– enfin si `C2` correspond au marquage `C1` dans lequel `S1` a été inactivé et `S2` a été activé.

L'application de cette règle à la configuration `C1=[[a,1],[b,0],[c,0]]` permet de déduire que les configurations suivantes possibles sont :
– `C2=[[a,0],[b,1],[c,0]]` par le tir de la transition `t1` ;
– `C2=[[a,0],[b,0],[c,1]]` par le tir de la transition `t2`.

Notons que l'application de cette règle par le joueur (C)LP fournit automatiquement toutes les configurations `C2` qui peuvent être atteintes à partir de `C1` par le tir d'une transition `T`.

La section 5.3 donnera plus de détails sur l'expression en (C)LP de la sémantique opérationnelle d'UML.

5.1.3 Étape C

L'étape C consiste à exprimer les règles de cohérence que doivent respecter les modèles UML.

Dans le cas de la vérification de cohérence comportementale, cette expression nécessite la description du comportement des modèles UML réalisée lors de l'étape B.

Soit la règle de cohérence « un modèle ne doit pas se trouver en état de blocage ». Il est possible de détecter le non respect de cette règle en formalisant l'incohérence associée par la règle :

$deadlock(C) :-$	
(1)	$reachable(C),$
(2)	$not(transition(C, _)).$

qui indique que C est une configuration atteignable (1) à partir de laquelle aucune évolution n'est possible (2). Nous supposons ici que les évolutions de configurations ont été implantées par le prédicat $transition/2$[1]. Le prédicat $reachable/1$ renvoie l'ensemble des configurations atteignables et peut être défini comme suit :

[1]Nous rappelons que les système logiques identifient un prédicat par son nom et son arité qui est le nombre de paramètres du prédicat. Ici, $transition/2$ veut dire que l'arité du prédicat $transition$ est 2.

$reachable(C) :-$
(1) $initialConf(C).$
$reachable(C2) :-$
(2) $reachable(C1),$
(3) $transition(C1, C2).$

où :
– (1) exprime qu'une configuration est atteignable si elle correspond à la configuration initiale du modèle ; celle-ci peut par exemple être donnée par un diagramme d'objets ;
– (2) et (3) expriment qu'une configuration $C2$ est atteignable s'il existe une configuration $C1$ atteignable à partir de laquelle il existe une transition vers $C2$.

Enfin, le diagnostic sur la cohérence du modèle vis-à-vis de la règle de cohérence considérée peut être obtenu par le but `?-deadlock(C)`. Appliquée à l'exemple de la figure 5.2, le joueur CLP donne la réponse `C=[[a,0],[b,1],[c,0]]`, c'est-à-dire la configuration où l'état `b` est actif. Le diagnostic fournit ainsi toutes les informations sur la configuration du système qui ne respecte pas la règle de cohérence. Il est également possible de fournir la trace qui a abouti à l'incohérence. Cette trace fournirait ici la réponse `t1`. L'expression des règles de cohérence est détaillée en section 5.4.

5.2 Description de la configuration des modèles UML

Cette section présente notre proposition pour spécifier les valeurs caractérisant la configuration des modèles UML (étape B.1 de la figure 5.1). Ce type d'information n'est pas formalisé dans le métamodèle actuel bien qu'il soit nécessaire à l'expression de la sémantique du langage. Notre but est d'adopter une approche systématique qui s'intègre totalement au métamodèle structurel existant [59].

La figure 5.5 montre la démarche. Nous proposons de réaliser un métamodèle « parallèle » (`Proposed Extension`) au métamodèle existant (`Existing MetaModel`) qui pour chaque élément (`Element`) possédant une configuration en fournit une description. Notre proposition consiste à donner une description hiérarchique des configurations. La configuration (`Configuration`) d'un élément pourra donc contenir d'autres configurations (fin d'association `ownedConfiguration`) de la même manière que les éléments d'UML peuvent contenir d'autres éléments (fin d'association `ownedElement`). Soit la configuration `C` d'un élément `E`. La configuration `C` sera décrite par la configuration des éléments contenus par l'élément `E`. Ce qui s'exprime en OCL par :

$context\ Configuration\ ::$
$self.element.ownedElement.includesAll(self.ownedConfiguration \rightarrow element).$

Par exemple la configuration d'un objet sera composée de la configuration de ses attributs ainsi que des configurations de ses caractéristiques comportementales. Lorsque l'élément est « atomique », la configuration de celui-ci peut être décrite par une valeur (`Value`). Par exemple la configuration d'un booléen sera donnée directement par sa valeur. Ces valeurs sont des éléments dynamiques dans le sens où ils ne sont pas constants mais peuvent changer au fil de l'évolution du modèle. Au contraire, certains éléments

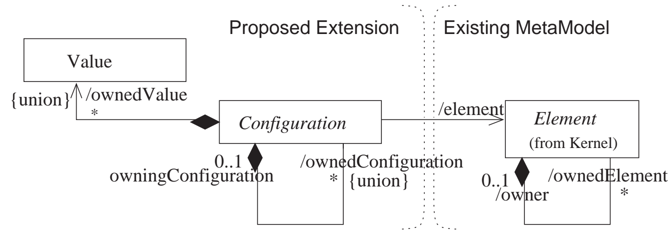


Fig. 5.5 – Diagramme Configuration

sont « purement structurels », c'est-à-dire qu'ils n'évoluent pas au cours du temps. C'est par exemple le cas des généralisations. Définir une configuration pour ces éléments n'a pas de sens car ils sont totalement spécifiés par la structure du modèle.

Afin d'illustrer notre propos, nous donnons des exemples de configurations pour certains éléments d'UML en section 5.2.1. Notre but étant de traiter les modèles UML dans leur intégralité, la section 5.2.2 introduit le concept de configuration globale d'un modèle UML. Nous montrons ensuite à la section 5.2.3 comment tirer parti de la représentation des configurations d'UML en MOF pour en déduire la représentation en (C)LP. La représentation en (C)LP est indispensable afin d'implanter un outil automatique de traitement des modèles (détection des incohérences dans notre cas).

5.2.1 Exemples de configuration

Dans cette partie nous décrivons la configuration de certains éléments des modèles UML. Cette section n'a pas pour but de présenter de manière exhaustive l'ensemble des informations contenues dans les configurations mais de montrer comment nous proposons de structurer ces informations.

La configuration d'un attribut (`PropertyConfiguration` de la figure 5.6) peut être donnée par une collection de valeurs. Les valeurs doivent correspondre au type de l'attribut. Enfin, lorsque l'attribut représente une fin d'association, les valeurs correspondent à des références sur les objets cibles.

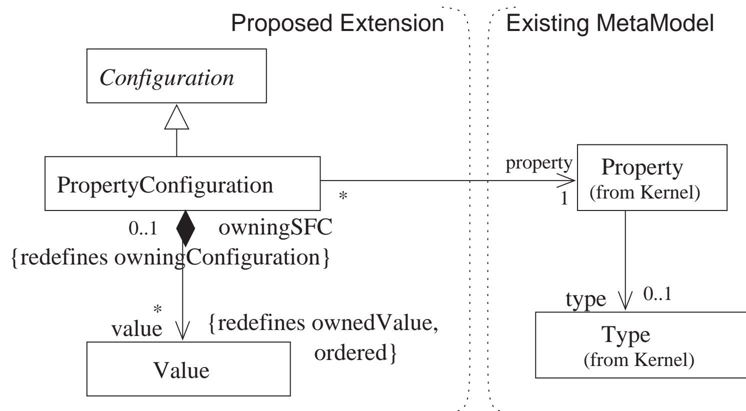


Fig. 5.6 – Diagramme PropertyConfiguration

La figure 5.7 représente la configuration d'une machine à états (`StateMachineConfiguration`). Les régions qui peuvent être contenues par les

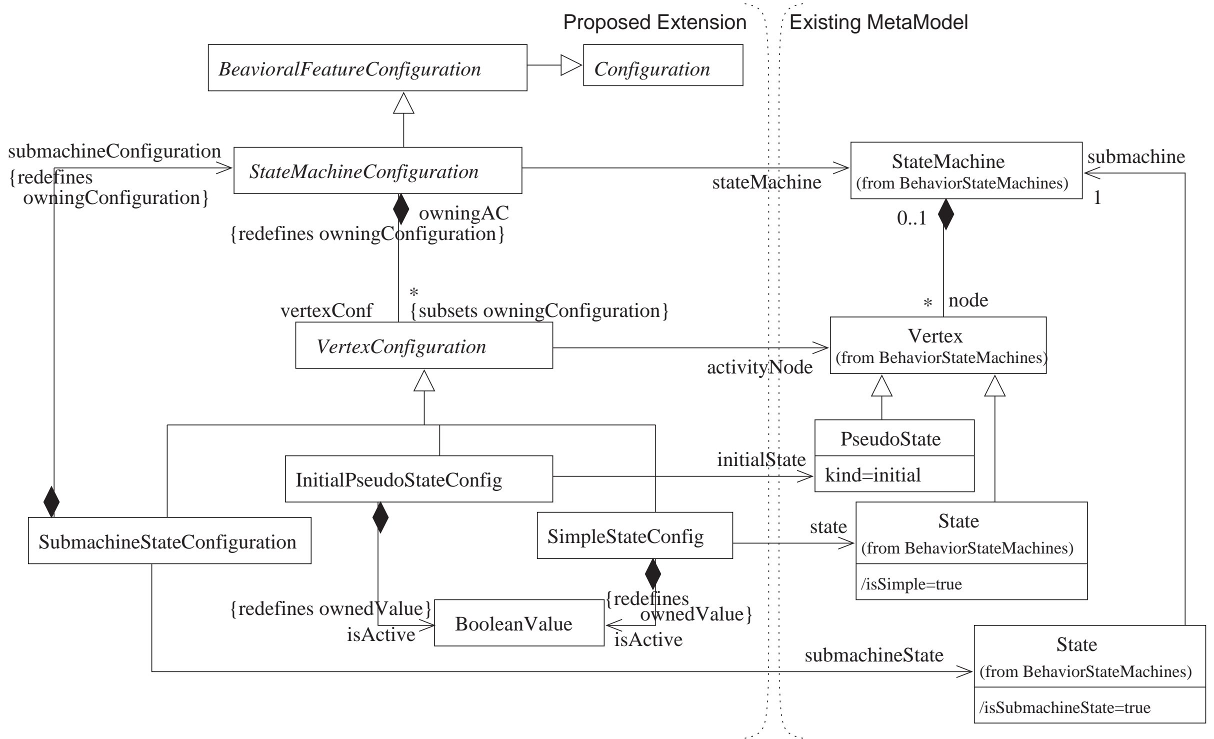


Fig. 5.7 – Diagramme StateMachineConfiguration

machines à états ne sont pas présentées afin d'alléger la figure. La configuration d'une machine à états est donnée par l'ensemble des configurations de ses sommets graphiques (`VertexConfiguration`). La configuration d'un sommet dépend du type de ce sommet. La figure présente trois types concrets de sommets, les pseudo-états initiaux (métaclasse `PseudoState`), les états simples (métaclasse `State` avec `isSimple=true`) ainsi que les états sous-machines (métaclasse `State` avec `isSubmachineState=true`). Les configurations d'un nœud initial et d'un état simple (respectivement les métaclasses `InitialPseudoStateConf` et `SimpleStateConf`) sont représentées par une valeur booléenne qui indique si l'état est actif ou non. En revanche, la configuration d'un état sous-machine (`SubmachineStateConfiguration`) correspond à la configuration de la machine à états associée (association `submachineConfiguration`) à l'état sous-machine (association `submachine`).

La figure 5.8 montre la configuration d'un objet (`ObjectConfiguration`). De nouveau, cette formalisation de la dynamique s'intègre au métamodèle existant. Afin de garder à cette section une taille acceptable, cette vue a été simplifiée : nous supposons que les seules caractéristiques structurelles des classes sont ses attributs (fin d'association `ownedAttribute`) et que la seule caractéristique comportementale d'une classe est le comportement de la classe elle-même (fin d'association `classifierBehavior`). La configuration d'un objet est composée de la configuration de ses attributs (`PropertyConfiguration`) et de la configuration du comportement qui décrit sa classe (`BehavioralFeatureConfiguration`). Les attributs dont les configurations sont contenues par la configuration de l'objet (fin d'association `ownedPC`) correspondent aux attributs de la classe (fin d'association `ownedAttribute`). La configuration des caractéristiques comportementales correspondent aux comportements contenus par la classe. Sur

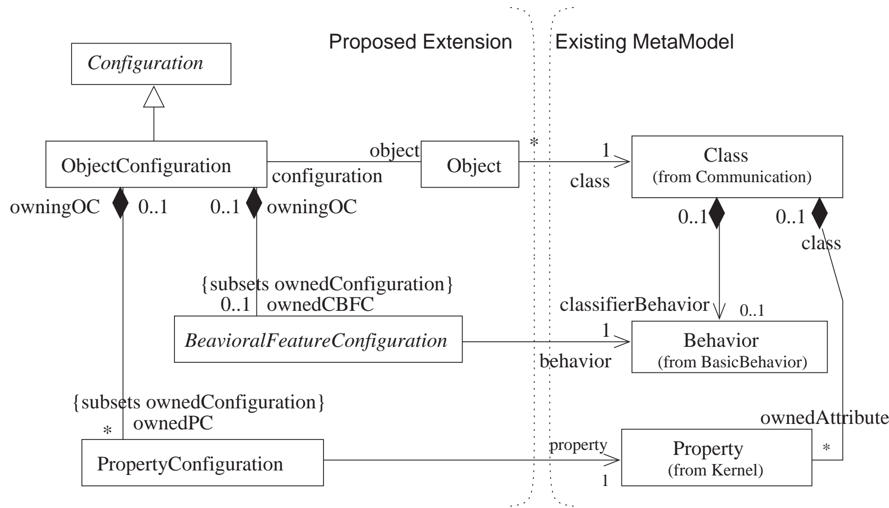


FIG. 5.8 – Diagramme ObjetConfiguration

la figure 5.8 un seul est représenté mais les comportements qui décrivent des méthodes d'opérations devraient aussi figurer. Notons que la spécification du comportement de la classe peut, par exemple, être réalisée par une machine à états, un diagramme d'activité, etc., car ce sont des spécialisations de la métaclasse `Behavior`. Ainsi, dans le cas où le comportement d'une classe est spécifié par une machine à états, la configuration d'un objet contient la configuration de la machine à états qui est décrite figure 5.7. Notons enfin que cette figure ne définit pas la configuration d'un objet dans sa totalité. En effet, afin d'implanter la sémantique des actions d'invocation comme l'envoi d'un signal ou l'appel d'une méthode, nous avons associé à chaque objet une collection d'événements. De plus, pour des raisons pratiques, la classe de l'objet est également spécifiée dans la configuration de celui-ci même si cette information n'est pas modifiée au cours de l'évolution du modèle.

5.2.2 Configuration globale d'un modèle UML

La configuration globale d'un modèle est définie par l'ensemble des configurations des objets qu'elle contient. La figure 5.9 synthétise ces relations.

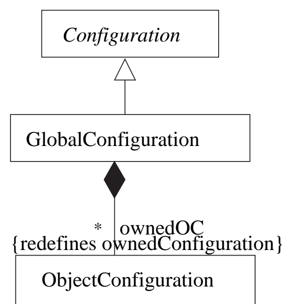


FIG. 5.9 – Diagramme GlobalConfiguration

5.2.3 Transformation en (C)LP

Dans cette partie nous abordons la représentation de la configuration en (C)LP. Celle-ci tire profit de la représentation en MOF de la configuration des modèles UML présentée précédemment.

Pour obtenir une description valable pour tout modèle UML nous avons procédé de manière systématique et donc facilement automatisable à partir de la description réalisée précédemment. La figure 5.10 montre la représentation en (C)LP de la configuration globale d'un modèle. Les configurations sont composées soit d'autres configurations (c'est le cas de `ObjectConf` par exemple) soit de valeurs (comme `PropertyConfiguration` par exemple). Lorsqu'une configuration possède plusieurs sous-éléments du même type nous utilisons une liste d'éléments de ce type. De plus, chaque configuration est précédée de l'identificateur de l'élément auquel la configuration se rapporte.

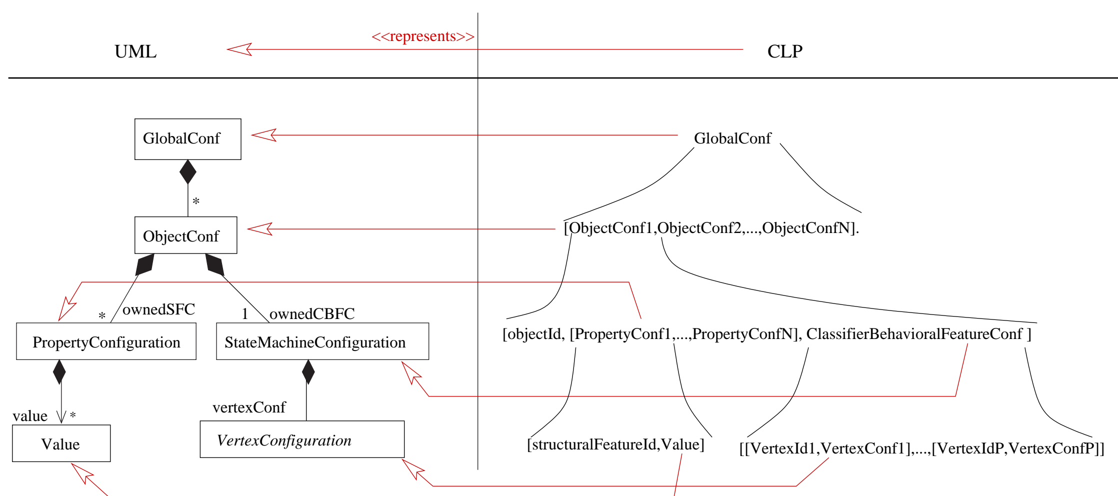


FIG. 5.10 – Représentation des configurations d'un modèle en (C)LP

Exemple de configuration d'un modèle Afin d'illustrer la représentation en LP de la configuration globale d'un modèle, nous introduisons ici le problème des philosophes. C'est un des problèmes les plus classiques de l'informatique pour illustrer l'interblocage. Il a été introduit par Edsger Dijkstra en 1971. Son traitement au moyen de LP a déjà été proposé. Cependant dans notre cas l'expression du problème en LP est déduite automatiquement de l'expression en UML.

Description du problème Supposons que la vie d'un philosophe consiste en une succession de périodes où celui-ci mange et pense. Chaque philosophe a besoin de deux fourchettes pour manger et chaque fourchette est prise une par une. Lorsqu'un philosophe est en possession de deux fourchettes, il mange et les repose lorsqu'il n'a plus faim. La figure 5.11 présente la situation du problème pour 5 philosophes. Ici les fourchettes jouent le rôle des ressources partagées.

La figure 5.12 est un modèle simplifié du problème des philosophes qui conduit à un deadlock. Le modèle complet tient compte du non déterminisme de la prise de fourchette.

La figure 5.13 est un diagramme d'objet qui spécifie l'état du système pour deux

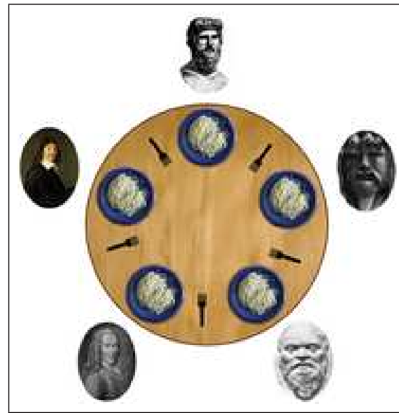


FIG. 5.11 – Représentation du problème des philosophes

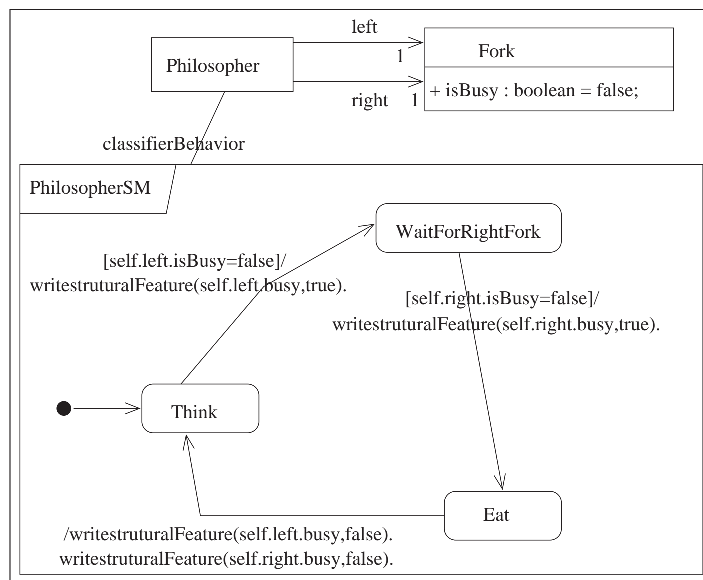


FIG. 5.12 – Modélisation du problème des philosophes - Diagramme de classes et de machine à états

philosophes. Lorsque les philosophes sont créés, l'état actif de leurs machines à états respectives est l'état initial de celle-ci.

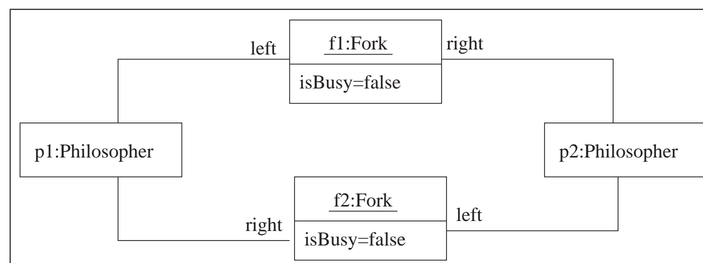


FIG. 5.13 – Modélisation du problème des philosophes - Diagramme d'objets

Exemple de configuration du modèle La représentation structurelle en LP du problème des philosophes obtenue par la traduction du modèle est présente en annexe D.

La configuration dynamique d'un modèle UML fait référence à la structure du modèle définie dans l'encodage. C'est pourquoi, avant de présenter une configuration du modèle des philosophes nous exposons ici une partie de son encodage structurel :

```
1 class(idRpy_12,default,true,default,philosopher,default,default,public).
2 property(idRpy_14,none,default,default,default,default,default,default,
3          default,default,default,1,right,default,1,public).
4 stateMachine(idRpy_18,default,default,default,default,
5              statechartOfPhilosopher,default,default,public).
6 pseudostate(idRpy_27,initial,default,default,public).
```

La configuration `C` présentée ci-dessous est l'encodage de la configuration initiale du problème avec deux philosophes.

```
1 C=[
2    [f2,idRpy_3,[[idRpy_11,[]],[idRpy_16,[]],[idRpy_6,[true]]],[],[]],
3    [f1,idRpy_3,[[idRpy_11,[]],[idRpy_16,[]],[idRpy_6,[true]]],[],[]],
4    [p2,idRpy_12,[[idRpy_14,[f1]],[idRpy_9,[f2]]],
5      [[idRpy_18,[[idRpy_27,1],[idRpy_20,0],[idRpy_24,0],[idRpy_26,0]]]],[]],
6    [p1,idRpy_12,[[idRpy_14,[f2]],[idRpy_9,[f1]]],
7      [[idRpy_18,[[idRpy_27,1],[idRpy_20,0],[idRpy_24,0],[idRpy_26,0]]]],[]]
8   ]
```

Comme exposé en section 5.2.2, nous avons défini la configuration globale d'un système par un ensemble de configurations d'objets. En LP ceci se traduit par une liste de configurations d'objets. La configuration d'un objet est également représentée par une liste. Considérons par exemple la configuration du philosophe `p2` lignes 4 et 5. Celle-ci contient :

- l'identifiant de l'objet `p2`,
- l'identificateur de la classe de l'objet `idRpy_12`, cette classe est spécifiée par la ligne 1 de l'extrait d'encodage (`class(idRpy_12,...,philosopher,...).`) ;
- la configuration de ses caractéristiques structurelles, par exemple la fourchette droite de ce philosophe est la fourchette `f1` (`[idRpy_14,[f1]]`), cette caractéristique structurelle est définie dans l'encodage par la ligne 2 et 3 (`property(idRpy_14,...,right,...).`) ;
- la configuration de sa machine à états (identifiée par `idRpy_18`), la machine à état est définie par le fait de la ligne 4 et 5 de l'encodage (`stateMachine(idRpy_18,...,statechartOfPhilosopher,...)`) ; celle-ci est dans l'état initial `idRpy_27`, l'état initial est défini par le fait de la ligne 6 (`pseudostate(idRpy_27,initial,...)` ;
- un file de messages qui est vide.

5.2.4 Conclusion

Nous avons introduit dans cette section, une manière de représenter et de structurer les informations contenues dans une configuration. Pour cela nous avons présenté le concept de configuration et des exemples de configurations pour certains éléments d'UML. Il est à noter que notre proposition s'intègre au métamodèle existant auquel elle ajoute des informations sur la sémantique d'UML.

Comme expliqué dans [11] la définition de la partie comportementale dépend de la partie structurelle et cette relation est unidirectionnelle. La figure 5.14 représente l'organisation entre le métamodèle existant et le paquetage proposé. Cette vision montre que la méthode décrite s'intègre au métamodèle existant. Nous proposons en effet de

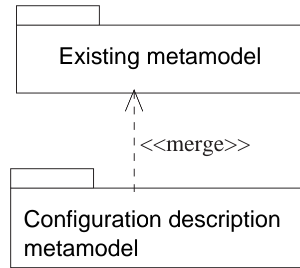


Fig. 5.14 – Organisation des paquetages du métamodèle existant et du paquetage de description de configuration

reprendre les concepts déjà décrits et de leurs adjoindre un paquetage qui décrit la configuration de ces concepts.

Cette approche avait déjà été suivie par [34]. Cependant, notre description a l'avantage de décrire les configurations à partir de sous-configurations ce qui permet de structurer l'ensemble des informations d'une configuration à partir de la description statique du modèle. [34] se limite à l'état de la machine à états qui décrit le comportement de la classe. Notre méthode est plus systématique et peut être appliquée sur tout modèle indépendamment des diagrammes employés car elle suit la hiérarchie décrite par le métamodèle existant. En particulier, notre méthode permet de décrire la configuration des états sous-machine en utilisant la configuration de la machine à états référencée, la configuration des états composites en fonction de la configuration des états contenus, etc.

Enfin, une telle description indépendante du langage d'analyse offre la possibilité d'en déduire de manière systématique une description en fonction du langage d'analyse choisi. Dans notre cas nous avons défini une transformation vers (C)LP mais ces travaux pourraient s'avérer utiles pour définir une traduction vers d'autres langages d'analyse.

5.3 Expression de comportement en (C)LP

Dans les sections précédentes nous avons vu qu'il était possible de représenter un modèle en (C)LP. Cette représentation est constituée de la structure du modèle ainsi que de sa configuration. Notre but étant de vérifier des règles de cohérence comportementales, nous devons également spécifier le comportement des modèles UML. Dans notre cas, ceci revient à spécifier les changements de configuration possibles à partir d'une configuration donnée (étape B.2 de la figure 5.1).

La figure 5.15 introduit ce concept (`ConfigurationChange`). Un changement de configuration transforme une configuration globale `before` en une configuration globale `after`. Ainsi, un changement de configuration peut être vu comme une transition entre deux configurations.

Avant toute expression de la sémantique opérationnelle des constructions d'UML, il est nécessaire d'identifier les constructions d'UML pour lesquelles une telle définition a un sens. C'est le but de la section 5.3.1. La section 5.3.2 montre ensuite comment le concept de changement de configuration peut être décrit en (C)LP. Le code complet est fourni en annexe C. Cette description se situe au niveau langage (ou métamodèle).

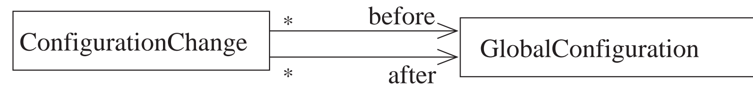


Fig. 5.15 – Diagramme d'évolution entre Configuration du Paquetage Dynamic

Comme expliqué plus tôt, nous illustrons les différents concepts en nous limitant aux machines à états. La section 5.3.3 a pour but de faire ressortir les concepts généraux nécessaires à la description de la sémantique opérationnelle. Ces concepts se situent donc au niveau méta-métamodèle. Enfin, la section 5.3.4 introduit le concept de point de variation sémantique.

5.3.1 Éléments agissant sur le comportement des modèles UML

Cette section présente les éléments d'UML agissant de manière active sur le comportement des modèles UML. Cette définition est importante car ce sont les éléments dont il faut traduire en (C)LP la sémantique opérationnelle. Deux types d'éléments sont concernés, les actions (cf. section 5.3.1.1) et les diagrammes dynamiques (cf. section 5.3.1.2).

5.3.1.1 Actions

Le chapitre 11 de la norme UML décrit les actions qu'un modèle UML peut exécuter. Il résulte des travaux menés par l'Action Semantics Consortium [2].

Par exemple, l'action « *CreateObjectAction is an action that creates an object that conforms to a statically specified classifier and puts it on an output pin at runtime* »[2]. La représentation de cette action est donnée par le sous-ensemble du métamodèle de la figure 5.16.

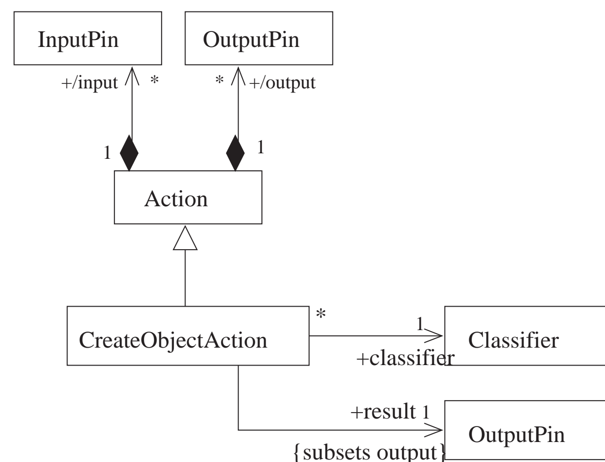


Fig. 5.16 – Description de l'action de création d'objets

[2]L'action de création d'un objet créé un objet conforme à un classificateur spécifié statiquement et le met dans un pin de sortie lors de l'exécution.

Le métamodèle exprime ainsi qu'une action de création d'objet prend en entrée un classificateur, crée un objet qui correspond au classificateur pour ensuite le placer dans son pin de sortie.

Le fait que la création d'un objet modifie la configuration du modèle UML est implicite dans la norme.

D'autres actions sont également décrites. Par exemple certaines actions décrivent la manipulation de liens (*CreateActionLinks, DestroyLinkAction*), l'appel d'opérations (*CallOperationAction*), l'envoi de signaux (*SendSignalAction*), etc.

Chacune de ces actions modifie la configuration du modèle UML. Pour exprimer le comportement des modèles UML, il faut donc exprimer les effets de l'exécution de ces actions sur la configuration du modèle.

5.3.1.2 Diagrammes dynamiques

Le langage UML décrit un certain nombre de diagrammes dynamiques. Par exemple, les machines à états permettent de modéliser des comportements discrets et donc de décrire le comportement d'une classe (et donc de ses objets) en fonction des événements reçus de son environnement.

Si un modèle UML contient une machine à états, le comportement effectif du modèle UML dépend de la sémantique des constructions (features) du diagramme de machine à états (état, transition, etc.). La description du comportement des modèles UML requiert donc la description du comportement de ces constructions, et plus généralement des constructions des diagrammes qui expriment le comportement des systèmes modélisés.

La spécification formelle du comportement des constructions dynamiques d'UML est en particulier indispensable pour établir la cohérence ou détecter des incohérences comportementales. Définir le comportement des modèles UML passe donc par la formalisation de la sémantique opérationnelle des actions et des transitions des diagrammes dynamiques. La section qui suit donne des exemples d'expression de sémantique opérationnelle afin d'en montrer la faisabilité en (C)LP.

5.3.2 Exemple de règles de changement de configuration

Nous introduisons ici des exemples de spécification de changements de configuration par des règles (C)LP.

Exemple 1 : Tir d'une transition d'une machine à états

Comme exprimé précédemment, un changement de configuration doit faire correspondre une configuration `before` à une configuration `after`. C'est le cas de la règle `stateMachineTrans` présentée ci-dessous qui à une configuration globale `ObjectConf1` fait correspondre un ensemble de configurations possibles `ObjectConf2`.

$stateMachineTrans(ObjectConf1, ObjectConf2, [Object|TransitionPath]) :-$
(1) $member([Object, Class, SFC, ConfigSM1, InputEvents], ObjectConf1),$
(2) $isActive(SourceStatePath, ConfigSM1),$
(3) $sourcePath(TransitionPath, SourceStatePath),$
(4) $targetPath(TransitionPath, TargetStatePath),$
(5) $guardIsOK(TransitionPath, ObjectConf1, Object),$
(6) $deactivate(SourceStatePath, ConfigSM1, ConfigSMInter),$
(7) $activate(TargetStatePath, ConfigSMInter, ConfigSM2),$
(8) $replace([Object, Class, SFC, ConfigSM1, InputEvents],$
$[Object, Class, SFC, ConfigSM2, InputEvents],$
$ObjectConf1, ObjectConfInter1),$
(9) $executeExitActions(SourceStatePath, Object, ObjectConfInter1, ObjectConfInter2),$
(10) $consumeEvent(TransitionPath, Object, ObjectConfInter2, ObjectConfInter3),$
(11) $executeActions(TransitionPath, Object, ObjectConfInter3, ObjectConfInter4),$
(12) $executeEntryActions(TargetStatePath, Object, ObjectConfInter4, ObjectConf2).$

Rappelons que la notation $[Tete|Queue]$ est la liste dont le premier élément est $Tete$ suivi des éléments de la liste *Queue*. Par exemple, on a $[a|[b, c]] = [a, b, c]$.

Cette règle exprime la sémantique opérationnelle d'une transition `TransitionPath` de la machine à états de l'objet `Object` avec les étapes suivantes :

- sélection d'un objet `Object` dans la configuration globale `ObjectConf1` du modèle (ligne (1)) ; rappelons que la configuration d'un objet est composée du nom de l'objet, de sa classe, la configuration de ses caractéristiques structurelles (`SFC`), de la configuration de sa machine à états et de la configuration de sa file d'événements ;
- sélection d'un état actif `SourceStatePath` dans la configuration de la machine à états `ConfigSM1` (ligne (2)) ; `SourceStatePath` contient le chemin pour aller jusqu'à l'état et pas uniquement l'identificateur de l'état car il est possible à cause des états sous-machines que le même état défini dans la partie structurelle soit activé dans plusieurs sous-machines à états ;
- récupération d'une transition sortante `TransitionPath` et d'un état cible de cette transition `TargetStatePath` par les lignes (3) et (4) ;
- test de la garde de la transition par la ligne (5) ; si la garde est fausse cette ligne fait échouer le prédicat ;
- désactivation de l'état source `SourceStatePath` par la ligne (6) ; cette ligne modifie la configuration initiale de la machine à états `ConfigSM1` et lui fait correspondre la configuration `ConfigSMInter` où l'état `SourceStatePath` est inactif ;
- activation de l'état cible `TargetStatePath` ce qui fournit la configuration `ConfigSM2` par la ligne (7) ;
- remplacement de l'ancienne configuration de la machine à états `ConfigSM1` par la nouvelle `ConfigSM2` par la ligne (8) ; on obtient ainsi une nouvelle configuration globale `ObjectConfInter1` ;
- exécution des actions de sortie (*exit actions* en anglais) de l'état source par la ligne (9) ; à partir de la configuration globale `ObjectConfInter1` on obtient ainsi la configuration `ObjectConfInter2` ;
- consommation de l'événement lié à la transition par la ligne (10) ; si l'événement n'est pas disponible cette ligne fait échouer le prédicat ;
- exécution des actions liées à la transition par la ligne (11) ;
- exécution des actions d'entrée (*entry actions* en anglais) de l'état cible par la ligne

(12).

Cette implantation de la sémantique opérationnelle du tir d'une transition fait un certain nombre de suppositions. Par exemple, cette expression suppose que les transitions sont tirées de manière atomique. Cette sémantique rend donc impossible l'enchevêtrement des tirs de plusieurs transitions. Ces choix sont délicats à réaliser car il est possible que certains utilisateurs aient besoin d'une autre sémantique. Une discussion sur le sujet est présentée en section 5.3.4.

L'expression en (C)LP permet ainsi de décrire l'évolution d'un diagramme d'états. Un atout de l'expression en (C)LP est qu'elle permet de réaliser du model checking par énumération des configurations accessibles et ce de manière intrinsèque. En effet dans notre exemple, le joueur (C)LP trouve toutes les valeurs possibles pour les variables `ObjectConf2` et `[Object|TransitionPath]` pour une valeur donnée de `ObjectConf1`. Ceci revient à trouver toutes les configurations atteignables `ObjectConf2` (configuration `after`) à partir d'une configuration `ObjectConf1` (configuration `before`) par le tir d'une transition `TransitionPath`.

Exemple 2 : Implantation de l'action SendSignalAction[3]

Nous avons vu en 5.3.1.1 qu'exprimer la sémantique opérationnelle d'UML requiert la prise en compte des actions disponibles. Nous présentons ici l'encodage de la sémantique opérationnelle de l'action d'envoi de signaux. La norme d'UML la décrit par le schéma présenté par la figure 5.17. L'association `signal` spécifie qu'un signal est associé à l'envoi d'un signal et l'association `target` spécifie l'objet auquel est associé le signal.

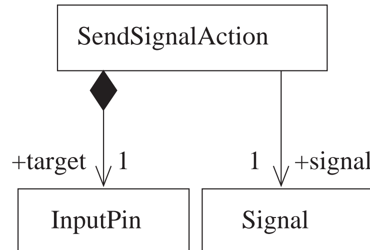


Fig. 5.17 – Description de l'action d'envoi de signal

L'implantation de cette action dans notre outil consiste à ajouter le nom du signal dans la pile de l'objet cible. Celle-ci est spécifiée de la manière suivante :

$sendSignalAction(Signal, TargetObject, ObjectConf1, ObjectConf2) :-$
(1) $getName(Signal, SignalName),$
(2) $member([TargetObject, Classifier, SFC, ConfigSM1, InputEvents], ObjectConf1),$
(3) $replace([TargetObject, Classifier, SFC, ConfigSM1, InputEvents],$
$[TargetObject, Classifier, SFC, ConfigSM1, [[SignalName, 0]|InputEvents]],$
$ObjectConf1, ObjectConf2).$

où :

– (1) permet de récupérer le nom du signal `SignalName`, c'est ce nom qui sera inséré dans la pile de l'objet cible ;

[3] Action d'envoi d'un signal

- (2) permet de récupérer la configuration courante de l'objet cible `TargetObject` dans la configuration globale initiale `ObjectConf1` ; notamment `InputEvents` l'état de la pile de cet objet ;
- (3) modifie la configuration de l'objet cible dans la configuration globale en ajoutant le nom du signal suivi d'un 0 dans la pile de ses événements ; on obtient ainsi la configuration globale `ObjectConf2` qui joue le rôle de configuration `after` à partir de la configuration globale `ObjectConf1` qui joue le rôle de la configuration `before`.

Nous avons choisi d'associer un 0 aux signaux pour pouvoir les différencier d'un appel d'opération auquel on associe un 1.

Pour chacune des actions, la norme décrit les différents paramètres d'entrées appelés « `input pin` » et de sorties appelés « `output pin` ». Comme remarqué à juste titre dans [23], aucune syntaxe concrète pouvant être intégrée dans les outils n'est normalisée pour les actions. Nous avons dû en définir une. Lors de cette définition, nous avons eu comme soucis de nous rapprocher au maximum des spécifications fournies par la norme. Les prédicats qui implantent les actions ont donc comme paramètres les différents pins associés à ces actions. Dans l'exemple fourni précédemment, les pins correspondent à l'objet récepteur du signal et au signal lui-même. Ils sont matérialisés par les paramètres `TargetObject` et `Signal`. L'apprentissage de la syntaxe des prédicats correspondants aux actions s'en trouve facilité.

Hiérarchisation des règles de changement de configuration

De plus, il est utile de pouvoir associer des priorités aux règles de changement de configuration. Dans le cas des machines à états par exemple, des priorités sur le tir de transition ont été définies dans certains cas [59] : *« By definition, a transition originating from a substate has higher priority than a conflicting transition originating from any of its containing states »*[4]. Ce type de priorité peut être géré en exprimant qu'on ne peut tirer la transition originaire d'un état uniquement si on ne peut pas tirer de transition originaire d'un de ses sous-états.

Ce type de priorité peut être implanté par les structures de contrôles « `if then else` » présentes dans la (C)LP.

Conclusion

L'expression de la sémantique en (C)LP est facilitée par la représentation des modèles UML et du métamodèle UML en (C)LP définie en [49]. Nous avons montré que l'encodage sous forme de (C)LP du métamodèle et des modèles UML permet de définir la sémantique opérationnelle d'UML via la définition de règles de changements de configuration. Cette possibilité sera utilisée pour réaliser la vérification de cohérence dynamique. De plus nous tirons parti de l'aspect déclaratif du paradigme de programmation logique qui permet une déclaration concise de la sémantique.

[4]Par définition une transition dont l'état source est un sous-état a une priorité plus grande qu'une transition en conflit dont la source est un état qui contient le sous-état.

5.3.3 Concepts de niveau méta-métamodèle

Dans cette partie nous nous intéressons aux concepts nécessaires à l'expression de la sémantique au niveau langage. En effet, notre but est de fournir une méthode qui s'intègre totalement aux techniques de métamodélisation. Un des inconvénients de la description des changements de configuration exprimée en (C)LP directement est qu'elle est dépendante du langage d'analyse que nous avons choisi. Afin que la sémantique soit intégrée à la norme nous pensons que son expression doit être réalisée de manière graphique et indépendante du langage d'analyse utilisé. Ces travaux sont en cours d'investigation et nécessitent une extension du MOF car, actuellement, seuls les aspects structurels peuvent être décrits par ce méta-langage. Nos travaux rejoindront ainsi le concept de métamodélisation dynamique (cf. [32, 81] par exemple). La description de la sémantique opérationnelle de manière graphique pourrait ainsi servir de base pour une transformation vers le langage d'analyse choisi. Dans notre cas, ceci permettrait une traduction entièrement automatisée d'UML vers (C)LP pour la partie dynamique, de la même manière que cela a été réalisé pour la partie structurelle. Le but serait ainsi d'intégrer à UML un paquetage qui décrit la sémantique opérationnelle de ses différentes constructions comme présenté par la figure 5.18.

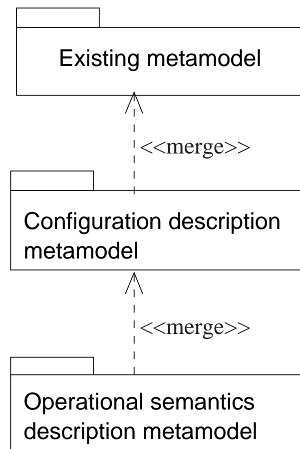


FIG. 5.18 – Organisation des paquetages du métamodèle existant, du paquetage de description de configuration et du paquetage de description de la sémantique opérationnelle

En vue de l'extension du MOF, nous nous intéressons maintenant aux concepts de niveau méta-langage nécessaires à l'expression de la sémantique au niveau langage. Comme illustré par la figure 5.19, la règle simplifiée qui décrit le tir d'une transition contient trois types de clauses :

- des <u>contraintes structurelles</u> qui pour un modèle donné sont soit toujours vraies soit toujours fausses, c'est par exemple le cas de `transition(T)`, qui est vrai uniquement si `T` est une transition ;
- des <u>contraintes dynamiques</u> qui dépendent de la configuration du système, c'est par exemple le cas de `isActive` qui dépend du marquage courant ;
- des <u>actions</u> qui modifient la configuration du système, c'est par exemple le cas `deactivate(S1,C1,CInter)` qui désactive `S1` dans `C1` ce qui donne `CInter`.

D'autre part, comme vu à la section précédente, il est parfois nécessaire de

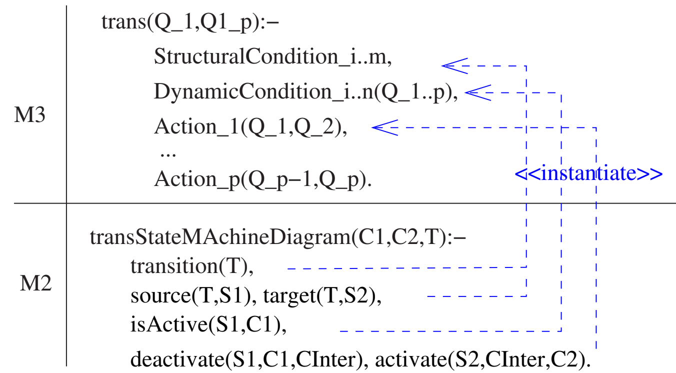


Fig. 5.19 – Concepts de niveaux M3 nécessaires à l'expression de la sémantique au niveau M2

hiérarchiser les règles de changement de configuration. Il est en effet possible que deux règles distinctes puissent s'appliquer à une configuration mais qu'une de ces deux règles soit plus prioritaire ou qu'une même règle qui puisse s'appliquer sur deux éléments différents doivent l'être sur un élément en priorité, etc. C'est par exemple le cas lorsque deux transitions `t1` et `t2` sont validées simultanément et que l'état source de `t1` est un sous-état de l'état source de `t2`.

Dans cette partie nous avons présenté les concepts principaux qui doivent pouvoir être supportés par le méta-langage pour exprimer la sémantique d'UML. Les (C)LP semblent être de bons candidats pour exprimer cette sémantique car ils permettent d'exprimer tous les concepts requis. La définition de ces concepts guidera nos travaux futurs qui consistera à étendre le MOF pour qu'il puisse les prendre en compte. Cette extension nous permettra ainsi de fournir une méthode totalement compatible avec les techniques de métamodélisation. De plus, la traduction vers (C)LP nous permettrait de fournir une méthode complètement outillée.

5.3.4 Points de variation sémantique

UML est un langage de modélisation dont le but initial est de couvrir l'ensemble des domaines du logiciel (base de données, systèmes de gestion, systèmes embarqués, etc.). Au fur et à mesure de son utilisation, son champ d'application s'est également ouvert à d'autres domaines comme l'ingénierie système. Afin de pouvoir répondre aux spécificités de chaque domaine, certains points de la sémantique d'UML sont laissés volontairement ouverts ce qui est appelé « point de variation sémantique ». Par exemple, la sémantique de l'action *AcceptEventAction* contient un point de variation sémantique : *« The arrangement of detected event occurrences is not defined, but it is expected that extensions or profiles will specify such arrangements »*[5]. De plus, la formalisation de la sémantique des modèles UML fait ressortir des points flous qui ne sont pas clairement identifiés comme tel. Ceci provient de la définition en langage naturel de la sémantique.

La prise en compte de ces points de variation sémantique est importante car les choix

[5]Les dispositions concernant la détection d'occurrences d'événements ne sont pas définies, mais il est prévu que des extensions ou des profils spécifient ces dispositions.

réalisés conduisent à des interprétations différentes des modèles UML. [12] propose pour cela de modéliser les points de variation sémantique afin de rendre les choix réalisés explicites et accessibles par l'utilisateur. Lors de la formalisation de la sémantique opérationnelle d'UML, nous avons été obligé de réaliser un certain nombre de choix. Cette partie sort quelque peu du sujet central de cette thèse, c'est pourquoi uniquement deux de ces choix sont présentés ici.

Atomicité du tir des transitions Comme introduit en 5.3.2, nous avons pris le parti de considérer que le tir d'une transition est atomique. Ce choix est discutable. En effet, le tir d'une transition peut-être associé à des actions. Or, dans l'implantation finale du système, rien n'assure que l'exécution de ces actions est atomique. Par exemple, dans le cas où les actions associées à une transition correspondent à l'écriture de deux attributs, il est possible qu'un processus concurrent prenne la main. L'entrelacement des actions peut alors créer des comportements qui ne seront pas explorés par notre outil.

Notons, que notre outil permettrait de prendre en compte des transitions dont le tir n'est pas atomique. Pour cela, il faudrait associer une configuration à une transition ce qui permettrait de connaître l'état d'avancement du tir de la transition. Ceci revient à introduire la notion « d'état de transition » qui serait distingué des « états stables ». Le tir d'une transition serait alors réalisé par une succession de configurations globales du modèle.

Gestion des événements Comme énoncé plus haut, la sélection des événements est un point de variation sémantique identifié. Pour être précis, il faudrait associer à la collection d'événements de chaque objet, une taille définie, une politique de sélection, de rejet, etc. Ce type d'information peut être rajouté grâce à la notion de profil.

Pour combler ce manque de sémantique, nous avons fait le choix d'empiler les événements dans l'ordre où ils arrivaient. De plus, un événement est considéré disponible s'il est présent dans la file. Enfin, sa consommation provoque la suppression de l'événement de la collection des messages.

Enfin, la taille de la collection des événements a été bornée de manière arbitraire. Ceci permet d'éviter d'atteindre des collections de messages de taille infinie ce qui provoque un nombre de configurations atteignables non borné et donc des calculs infinis.

5.4 Expression des propriétés

La dernière étape de notre approche (étape C de la figure 5.1) consiste à écrire en (C)LP les règles de cohérence dynamiques que doivent respecter les modèles UML.

Les étapes précédentes permettent de décrire les valeurs caractéristiques d'une configuration σ_i d'un modèle UML et de spécifier les transitions $< \sigma_i, \sigma_{i+1} > \in t$ entre deux configurations. Notre but est de pouvoir réaliser des analyses sur le comportement des modèles UML, ce qui implique de connaître l'ensemble des traces possibles du système ou au moins d'avoir une approximation de cet ensemble de traces. La section 5.4.1 présente la vérification de règles de cohérence faisant intervenir uniquement des invariants d'états et introduit la notion d'approximation. La section 5.4.2 traite du cas plus général.

5.4.1 Cohérence faisant intervenir des invariants de configuration

Comme intuitivement présenté à la section 5.1.3, nous exprimons les règles de cohérence en calculant l'ensemble des configurations atteignables. Ceci revient en fait à réaliser une approximation de la sémantique concrète en abstrayant les traces (notées $\sigma_0 ... \sigma_{n-1}$ ou $\sigma_0\sigma_1 ...$) par l'ensemble $\alpha_{\mathcal{S}}$ des configurations de cette trace [17] : $\alpha_{\mathcal{S}}(\sigma) = \{\sigma_k : k \in [0, |\sigma|[\}$. L'approximation de l'ensemble $\mathcal{X}$ des traces possibles est alors l'ensemble des configurations figurant sur au moins une trace de l'ensemble : $\alpha_{\mathcal{S}}(\mathcal{X}) = \bigcup_{\sigma \in \mathcal{X}} \alpha_{\mathcal{S}}(\sigma)$. Cette abstraction permet de réaliser des analyses d'invariants sur les configurations atteignables (« *safety analysis* »). La science de l'approximation est plus connue sous le nom d'interprétation abstraite.

Nous introduisons ici un exemple afin d'expliquer en quoi nous approximons le comportement du modèle étudié. Imaginons deux compteurs 2 bits dont un compteur modulo 4 et un compteur-décompteur. Ces deux compteurs peuvent être représentés par les graphes i) et ii) de la figure 5.20. Le schéma iii) de cette figure représente une approximation de ces deux comportements qui consiste à ne tenir compte que des états atteignables. Ce type d'abstraction permet de vérifier des propriétés d'invariants sur les états, comme par exemple, « l'état où les deux bits sont à 1 n'est pas atteignable ». Cette propriété n'est ici pas respectée. Par contre, cette approximation ne permet pas de répondre à la propriété « en considérant l'état `00` comme initial, l'état `01` est forcément atteint avant l'état `11` ». Notons que cette propriété est vraie pour l'automate i) mais pas pour l'automate ii).

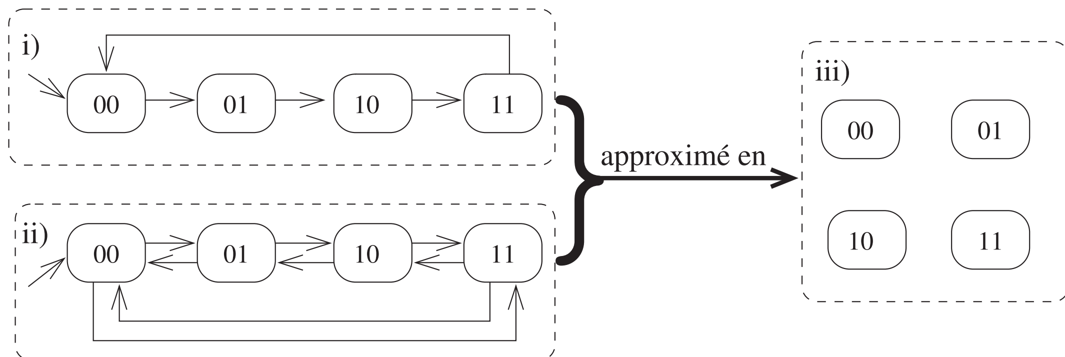


Fig. 5.20 – Comportements de compteurs et approximation de ces comportements

Rappelons que l'ensemble des configurations atteignables pour un modèle UML est défini par le prédicat `reachable/1` de la manière suivante :

$reachable(C) :-$	
(1)	$initialConf(C).$
$reachable(C2) :-$	
(2)	$reachable(C1),$
(3)	$transition(C1, C2).$

Ce prédicat offre ainsi la possibilité d'écrire un ensemble de propriétés sur les configurations redoutées que le modèle peut atteindre. On peut par exemple détecter une

situation de blocage du modèle en résolvant le prédicat `deadlock` présenté en 5.1.3. Un autre exemple de règle de cohérence comportementale est que tout état d'une machine à états doit pouvoir être activé. La négation de cette règle peut être détectée grâce au prédicat $nonReachableState/1$:

$nonReachableState(State) :-$	
(1)	$state(State),$
(2)	$not(isAReachableState(State)).$
$isAReachableState(State) :-$	
(3)	$reachable(Conf),$
(4)	$objectInState(Conf, State).$

où $objectInState$ fournit l'ensemble des états actifs $State$ pour une configuration $Conf$ donnée. Ce prédicat exprime qu'un état peut être activé si il existe une configuration atteignable où cet état est activé `(3 et 4)`. Un état ne peut pas être activé dans le cas contraire `(2)`.

Il est également possible d'exprimer des propriétés sur les états atteignables en faisant référence à un état atteignable suivi d'un changement de configuration. C'est par exemple utile pour détecter une transition d'un diagramme d'états non tirable :

$nonFirable(T) :-$	
(1)	$transition(T),$
(2)	$not(isFirable(T)).$
$isFirable(T) :-$	
(3)	$reachable(Conf),$
(4)	$transStateMachine(Conf, _, T).$

Une transition tirable est une transition qui pour une configuration atteignable peut donner lieu à un changement de configuration `(3 et 4)`. Une transition n'est pas tirable dans le cas contraire `(2)`.

Notons que cette solution ne permet pas l'expression de propriétés sur des traces de longueur quelconque. Ceci peut en effet aboutir à une dérivation infinie si à partir d'une configuration `C` il existait une suite de changements de configuration qui aboutisse de nouveau à la configuration `C`. Ceci est dû à l'approximation de la sémantique que nous avons réalisée.

Intérêt de la détection des règles de cohérence comportementale Ce type de détection est par exemple intéressant pour le domaine avionique. Les autorités de certification exigent en effet l'absence de code mort. Or, dans le cas où les diagrammes de machines à états seraient utilisés pour la génération de code, il est raisonnable de penser que le code généré à partir d'un état non atteignable ou d'une transition non tirable serait du code mort. S'assurer de ces propriétés au niveau modèle permet donc d'éviter la génération de code mort. D'autre part, les systèmes logiciels critiques sont souvent implantés par des processus communiquant. Détecter l'inter-blocage de ces processus au niveau modèle permet de rectifier en amont les erreurs de modélisation aboutissant à cette situation redoutée.

Diagnostic Comprendre l'évolution du modèle UML pour aboutir à la situation redoutée est extrêmement complexe à réaliser intellectuellement. Le diagnostic doit donc fournir à l'utilisateur les informations pertinentes sur les changements de configuration qui aboutissent à la situation redoutée à partir de l'état initial. Dans le cas des machines

à états, nous proposons de mettre en évidence les transitions tirées. Cette évaluation est réalisée par le parcours du graphe des états atteignables et est construite après la détection de l'incohérence. Une autre approche consisterait à construire ce diagnostic lors de l'évaluation de l'incohérence. Le diagnostic pourrait alors être déduit en fournissant certains détails de la preuve logique comme expliqué en [62]. Plus de détails concernant le diagnostic sera fourni au chapitre suivant.

5.4.2 Expression de cohérence sur l'ensemble des traces

Notre prototype permet également de réaliser des raisonnements sur l'ensemble des traces du modèle UML analysé. Rappelons qu'une trace est une succession de configurations qui représentent une évolution possible du modèle. Ceci peut être réalisé en utilisant le prédicat `reachable/2` :

$: -table\ reachable/2.$
$reachable(C1, C2) : -$
(1) $transition(C1, C2).$
$reachable(C1, C2) : -$
(2) $transition(C1, CInter),$
(3) $reachable(CInter, C2).$

Celui-ci exprime qu'une configuration `C2` est atteignable à partir d'une configuration `C1` s'il existe une transition entre `C1` et `C2` (ligne 1) ou s'il existe une transition de `C1` vers une configuration `CInter` (ligne 2) et que `C2` est atteignable à partir de `CInter` (ligne 3). Ce prédicat est tabulé pour éviter le problème d'arbre de dérivation infini exposé en 3.2.2.1. Nous supposons que le prédicat `transition/2` définit l'ensemble des transitions entre configurations possibles comme par exemple le changement de configuration induit par le tir d'une transition de machine à états.

Expressivité Utiliser le prédicat `reachable/2` qui mémorise dans les tables l'ensemble des transitions entre configurations permet l'analyse de propriétés sur des traces de longueurs quelconque. Nous ne détaillerons pas plus cette possibilité car nous ne l'avons pas explorée suffisamment au cours de cette thèse. C'est cependant la solution employée par [18] qui a permis d'encoder la logique temporelle « $\mu - calculus$ ». Ceci permet d'exprimer des propriétés sur les chemins possibles et non plus uniquement sur les états atteignables. Ce prédicat permet par exemple l'expression et la vérification des propriétés de vivacité. Une propriété de vivacité énonce que sous certaines conditions, quelque chose finira par avoir lieu [72], par exemple que « si l'on appelle l'ascenseur, la cabine finira par arriver ».

Performances D'un point de vue temps de calcul le prédicat `reachable/1` (de la section 5.4.1) est meilleur car dans la plupart des cas le nombre de configurations atteignables par un modèle UML est inférieur au nombre de transitions entre ces configurations. De plus, la mémorisation dans les tables d'une transition nécessite de garder en mémoire deux configurations (les configurations cible et source), la solution utilisant le prédicat `reachable/1` est donc plus économe en octets. De plus, les règles de cohérence étant des propriétés génériques, cette solution plus performante nous permet d'en exprimer la plupart.

5.5 Conclusion

Notre méthode permet l'expression formelle du comportement des modèles UML. Le caractère déclaratif des règles de changement de configuration conduit à atteindre naturellement l'exhaustivité dans l'exploration des transitions possibles entre configurations. Ceci permet de mettre en œuvre des techniques de model-checking et non pas de simple simulation. Le calcul de l'ensemble des traces du modèle UML analysé est alors possible. Les propriétés de cohérence peuvent alors être vérifiées. L'explosion combinatoire rend cependant difficile l'exploration de l'ensemble de ces traces. Pour remédier en partie à ce problème nous avons eu recours à une approximation de ces traces qui consiste à ne considérer que les configurations atteignables. Cette approximation réduit cependant le type des propriétés vérifiables à des analyses qui ne mettent pas en jeu des traces de longueur quelconques comme par exemple l'analyse d'invariants d'états. Réaliser de telles analyses est utile dans un contexte de certification, par exemple en prévenant la génération de code mort. Nous pensons que d'autres approximations nous permettraient de vérifier d'autres propriétés tout en maîtrisant l'explosion combinatoire.

L'expression de la sémantique opérationnelle soulève cependant un certain nombre de difficultés. La principale est de faire des choix concernant les points de variation sémantique qui ne sont pas clairement identifiés comme tels dans la norme [59]. L'approche proposée par [12] qui consiste à modéliser ces points de variation sémantique nous semble intéressante et pourrait être incorporée à notre méthode. Nous pensons cependant que bon nombre de ces points d'ombre devraient être résolus en amont. Ceci pourrait être réalisé en exprimant la sémantique opérationnelle des différentes constructions d'UML au sein de la norme. Cette expression doit être indépendante d'un langage d'analyse ou d'une plate-forme. Ce concept est appelé métamodélisation dynamique, il conduit à des travaux qu'il faudrait mener à la suite de cette thèse. Pour le moment, seuls les concepts ont été identifiés.

Parmi les limites de l'outil, il est bon de faire remarquer que l'expression de la sémantique des modèles UML n'a pour le moment pas fait l'objet d'analyses théoriques. Seule une phase de mise au point expérimentale a été réalisée. Si cet outil devait être utilisé dans un contexte industriel critique, une phase de validation de l'outil serait donc nécessaire. De plus, à l'heure actuelle, nous ne proposons aucun moyen théorique permettant de prouver cette sémantique.

Le temps de réponse des systèmes critiques est souvent contraint. Notre outil ne permet pour le moment pas de prendre en compte le temps. Comme le montre un certain nombre de travaux, [21] par exemple, les CLP permettent de réaliser des analyses temporelles. Pour cela le domaine de contraintes sur les réels est utilisé. En l'état, notre outil n'utilise en fait que la programmation logique. Une évolution possible serait donc d'utiliser les contraintes afin de prendre en compte la modélisation du temps.

Chapitre 6

Applications

Le but de ce chapitre est de présenter les résultats expérimentaux obtenus par l'outil. La section 6.1 présente les constructions prises en compte par le prototype. Nous avons ensuite réalisé deux analyses de performances. La section 6.2 expose l'analyse sur le problème des philosophes. L'étude d'un problème « universitaire » est importante pour pouvoir réaliser des comparaisons entre outils et ainsi comparer les méthodes sous-jacentes. La section 6.3 présente les résultats obtenus sur un modèle UML industriel.

6.1 Constructions prises en compte

Cette partie présente les éléments d'UML traités par l'outil développé pour valider notre approche. Le code correspondant est énuméré dans l'annexe C.

États Les trois types d'états d'UML sont pris en compte (cf. figure 6.1i). Les états simples qui sont soit actifs soit inactifs. Les états sous-machines qui sont associés à une machine à états. La configuration d'un état sous-machine correspond en fait à la configuration de la machine à états référencée. Les états composites qui sont composés de plusieurs autres états. La configuration d'un état composite est composée de la configuration des états contenus. Un état composite peut contenir plusieurs régions (*region* en anglais) qui peuvent évoluer de façon indépendante les unes des autres. Ce concept est également supporté.

Transitions Les transitions entre les nœuds représentent une transition d'un état vers un autre état (cf. figure 6.1ii). Les transitions peuvent être associées :

- à des conditions de garde qui doivent être vraies pour que la transition soit tirée ;
- à des déclencheurs qui peuvent par exemple correspondre à la réception d'un signal ;
- à des effets qui sont des actions à réaliser si la transition est tirée.

Une transition a exactement un nœud source et un seul nœud cible.

Pseudo-états UML introduit six types de pseudo-états présentés à la figure 6.1iii).

Les pseudo-états initiaux et finaux représentent respectivement le début et la fin de la région englobante. Les nœuds initiaux nous permettent de construire la configuration initiale de la machine à états. L'exécution d'un état est terminée si toutes les régions othogonales qu'il contient ont atteint leur nœud final.

Les pseudo-états de décision, de jonction, de bifurcation et de synchronisation (resp. *DecisionNode, MergeNode, ForkNode* et *JoinNode* en anglais) sont utilisés pour connecter plusieurs transitions et ainsi former des chemins plus complexes entre les états.

Un nœud de décision (*decision* en anglais) permet de réaliser des branchements conditionnels dynamiques. Les gardes sont évaluées pour déterminer quelle transition sortante doit être traversée. Un nœud de jonction (*junction* en anglais) permet d'assembler plusieurs transitions entrantes et sortantes. Un nœud de bifurcation (*fork* en anglais) est tel que si l'état source de l'unique transition entrante est activé, les états cibles des transitions sortantes sont tous activés. Un nœud de synchronisation (*join* en anglais) est tel que si tous les états sources des différentes transitions entrantes sont activés, alors l'état cible de la transition sortante est activé.

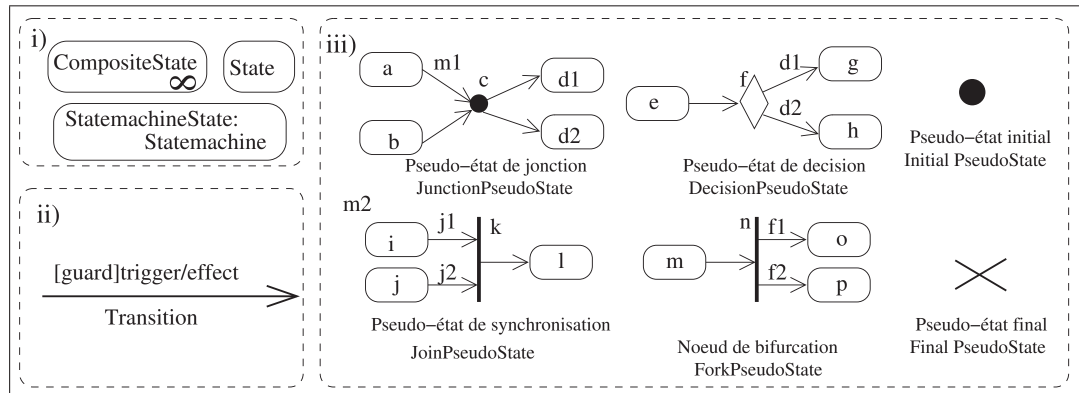


Fig. 6.1 – Représentation graphique des nœuds pouvant apparaître dans les machines à états

Le code correspondant à la sémantique opérationnelle de ces constructions est présent en annexe C.1.

Actions Les principales actions présentes dans [58] sont également prises en compte. C'est le cas de `createObjectAction`, `createLinkAction`, `destroyLinkAction`, `readLinkAction`, `addStructuralFeatureValueAction`, `readSelfAction`, `sendSignalAction` et `acceptSignalAction`. Notons que l'implantation de l'envoi et de la réception de signaux a nécessité d'associer à chaque objet une pile d'événements ce qui n'est pas clairement explicité dans la norme. La code correspondant est présent en annexe C.2.

6.2 Résultats sur le problème des philosophes

Avant d'être testé sur modèle industriel nous avons testé notre outil sur une modélisation du problème des philosophes. Ce problème a été introduit à la section 5.2.3. Le modèle sur lequel notre analyse a été appliquée y est présenté par la figure 5.12.

Remarquons que les gardes et actions ont été écrites en « pseudo-OCL ». Pour que notre model checker prenne en considération ces éléments du modèle il faut cependant les écrire avec une syntaxe compatible avec celui-ci. Par exemple, considérons la condition de garde de la transition qui va de l'état où le philosophe pense à l'état où la philosophe attend la fourchette droite. Cette transition modélise la prise de la fourchette gauche. Elle ne peut être tirée que lorsque la fourchette gauche est libre, c'est pourquoi elle est associée à la condition de garde `[Self.left.isBusy=false]`. En fait, la syntaxe attendue par notre outil pour exprimer cette condition de garde est :

```
(1)    (readSelfAction(Self),
(2)    readStructuralFeatureAction(Self,left,LeftFork),
(3)    readStructuralFeatureAction(LeftFork,isBusy,LeftForkIsBusy),
(4)    LeftForkIsBusy=false)
```

où :

- (1) est la lecture de l'objet contenant la machine à états, le résultat est donné par la valeur de `Self` ;
- (2) est la lecture de l'attribut `left` de l'objet courant, le résultat est placé dans `LeftFork` ;
- (3) permet de lire l'attribut `isBusy` de l'objet `LeftFork` et de placer le résultat dans `LeftForkIsBusy` ;
- enfin (4) permet de vérifier que `LeftForkIsBusy` est unifiable à `false` ce qui est vrai dans les cas où `LeftForkIsBusy` est indéfini ou égal à `false` ; si on avait voulu évaluer la garde à faux dans le cas où `LeftForkIsBusy` est indéfini il aurait fallu utiliser le test d'égalité `'@='` au lieu de l'opérateur d'unification `'='`.

Cette notation est plus lourde que celle exprimée en OCL. Cependant, l'expression OCL n'est pas exécutable car les actions comme `writeStructuralFeatureAction` ne sont pas implantées et car les attributs ne sont pas valués puisque les informations comportementales ne sont pas traitées. Au contraire, les actions que nous utilisons comme `readSelfAction` sont celles spécifiées dans la norme. Ces actions sont implantées dans notre outil sous forme de clauses ce qui les rend exécutables (cf. annexe C.2).

Conditions initiales Il faut spécifier la configuration initiale du modèle en début d'analyse. Pour cela notre prototype offre deux possibilités : décrire un diagramme d'objets correspondant aux conditions initiales ou exécuter des actions sur une configuration vide afin de créer les objets initiaux. La partie 5.2.3 montre un exemple de condition initiale.

Résultats expérimentaux Le tableau 6.1 présente les résultats expérimentaux de notre prototype. Ces résultats montrent des temps d'exécution convenables. Ils ont été obtenus sur un PC de bureau sous Linux équipé d'un processeur de 2GHz et de 438,54MB de mémoire vive. La première colonne exprime le nombre d'instances de la classe `Philosophe` qui correspond également au nombre d'instances de la classe `Fourchette`. La deuxième colonne fournit le temps de détection de l'inter-blocage et la dernière colonne le nombre d'états atteignables. Cette table fait ressortir le problème bien connu de l'explosion combinatoire. C'est un facteur limitant dans notre cas. En effet, alors que les temps d'exécution sont tout à fait corrects, l'explosion combinatoire conduit à un manque de mémoire. C'est pourquoi dans le cas de 10 philosophes, le nombre d'états total n'a pas pu être atteint. La détection du deadlock a quand même été possible car la détection d'une configuration redoutée ne nécessite pas forcément l'énumération complète de toutes les configurations atteignables.

Diagnostic Le diagnostic concernant la détection d'incohérence comportementale nécessite un traitement spécial. En effet, détecter qu'un modèle arrive dans une situation redoutée est intéressant mais il faut fournir à l'utilisateur les moyens de traiter cette incohérence. Ceci est réalisé en construisant la trace, c'est-à-dire les différents change-

Nb de philosophes	Temps de détection d'un deadlock	Nb d'états atteignables
5	0.227s	573
6	1.05s	2041
7	4.5s	7269
8	19.97s	25889
9	86,5s	92205
10	358s	Non disponible

Tab. 6.1 – Résultats expérimentaux pour le problème des philosophes

ments de configuration qui partent de la configuration initiale à la situation redoutée. Ceci est réalisé par le prédicat `show_trace/3` qui parcourt le graphe d'atteignabilité. Ce prédicat est présent en annexe C.4. C'est un prédicat de recherche de chemin dans un graphe comme présenté dans [76] (p.94). Une fois que la trace est construite, il est possible d'en extraire les informations utiles pour l'utilisateur. Par exemple, le prédicat `show_transition_and_conf/1` fournit dans un fichier la transition qui est tirée, l'objet à laquelle appartient cette transition et les états actifs de l'objet pour chaque changement de configuration de la trace. Plus de détail sur les informations extraites de la trace est présenté en annexe E.

6.3 Résultats sur un modèle industriel

Cette section présente les résultats d'une expérimentation sur un modèle industriel. Nous présentons rapidement le modèle en section 6.3.1. La section 6.3.2 montre les performances obtenues pour les diverses expérimentations.

6.3.1 Description du modèle

Le modèle étudié est un sous-ensemble du modèle du logiciel de gestion de vol de l'A400M. Ce modèle est développé par Thalès Avionics. Le modèle global est extrêmement complexe. Une description quantitative est disponible en [79].

Pour notre expérimentation nous nous sommes intéressés à un sous-ensemble de ce modèle. Le système modélisé est chargé de gérer la redondance entre plusieurs calculateurs ce qui constitue une fonctionnalité critique.

Le nombre de classes du modèle est plus que modeste puisque celui-ci est de quatre. Ce sous-ensemble du modèle a cependant été choisi car le comportement de ces classes est exprimé via des machines à états relativement complexes. Le comportement est spécifié au moyen d'une centaine d'états et d'une centaine de transitions. Ceux-ci sont répartis dans 33 régions différentes ce qui implique un fort parallélisme. Enfin, les machines à états sont souvent décrites au moyen d'états composites ou sous-machine. Le nombre de niveau d'états imbriqués maximum est 5.

Modifications du modèle avant l'analyse

Le modèle UML a dû être modifié en vue de l'analyse. Ces modifications concernent deux points.

D'une part, le fait qu'aucun langage d'action concret ne soit normalisé par l'OMG amène les utilisateurs d'UML à introduire des actions qui sont soit dépendantes d'un outil soit le fruit de guides internes aux entreprises. Afin d'outiller la validation des machines à états, il est cependant nécessaire de définir un tel langage d'action. Ceci permet par exemple de spécifier la syntaxe que devra respecter l'envoi de messages entre machines à états. La section 6.2 explique succinctement la définition choisie pour le langage d'action. Notons que cette lacune avait déjà été soulignée par [23] qui a également défini un langage d'action. La première modification apportée au modèle a donc consisté à traduire les actions afin de les rendre compatibles avec notre outil. Cette traduction a été faite manuellement. Nous pensons cependant qu'un langage d'action concret devrait être inclus dans la norme d'UML. Il serait alors possible de traduire automatiquement ce langage d'action indépendant de tout outil vers le langage d'action analysable par l'outil utilisé. Ceci constitue à notre sens une étape nécessaire afin qu'UML puisse être considéré comme un véritable langage de modélisation validable plutôt que comme un langage de modélisation exclusivement graphique.

D'autre part, le modèle sur lequel a été réalisée l'expérimentation n'est pas le modèle final du logiciel de vol mais est le fruit d'une itération non finale du processus. Ceci explique que le comportement de certaines classes ne soit pas spécifié par des machines à états. Ceci pose un problème lorsque ces classes interagissent avec les classes dont nous voulons valider le comportement. En effet, si les classes sans machines à états sont responsables de l'envoi de messages à destination des machines à états modélisées, ces dernières restent en attente des messages manquants puisque leur environnement n'est pas formalisé dans le modèle. Afin de tester notre outil malgré ces limitations nous avons choisi de réaliser trois analyses différentes :

1. Vérifier les machines à états sans modèle de l'environnement. Le résultat de cette analyse est alors pessimiste puisque les machines à états peuvent se retrouver bloquées dans l'attente des messages manquants. La détection des incohérences est alors due à une incomplétude du modèle plus qu'à son incohérence. Cette analyse ne nécessite aucune modification du modèle.

2. Supposer que les machines à états non spécifiées envoient les messages dès lors qu'ils sont consommés. Le résultat de cette analyse est alors quelque peu optimiste puisque qu'il est possible que ce ne soit pas le cas. Toutefois si une incohérence est trouvée il est certain que ce problème est présent quel que soit le comportement de l'environnement. La modification apportée au modèle pour réaliser cette analyse consiste à enlever les déclencheurs (*trigger* en anglais) des transitions correspondant à des messages produits par l'environnement.

3. Déduire l'envoi des messages manquants à partir des diagrammes de séquences. Ceci est réalisé en déduisant une machine à états qui modélise le comportement de l'environnement à partir des diagrammes de séquences où l'environnement interagit avec les classes analysées. Cette déduction a été réalisée manuellement mais il semble possible de l'automatiser. Remarquons qu'il ne s'agit pas d'un modèle complet de l'environnement. De plus, le degré de complétude des machines à états déduites dépend de la couverture des différents messages spécifiée dans les diagrammes de séquences.

Notons que ces modifications ont donné lieu à la réalisation d'une série de guides. Ces guides ont pour but de permettre la validation des modèles sans avoir à y apporter les modifications énoncées plus tôt. La plupart de ces guides conseillent de formaliser les informations qui apparaissent sous forme de notes non analysables. Ce travail est toutefois quelque peu annexe à ma thèse et c'est pourquoi un seul guide est présenté ici.

• Contexte : Les actions spécifiées dans UML permettent d'accéder aux attributs (lecture et écriture). Ces actions peuvent être intégrées aux machines à états. Le comportement des machines à états peut dépendre de la valeur des attributs.

• Guide « Accès à des attributs » : Il est conseillé de formaliser l'accès aux attributs (lecture écriture) lorsque ceux-ci influencent le comportement du modèle.

6.3.2 Résultats

Les résultats des analyses décrites précédemment sont fournis par les sections 6.3.2.1 à 6.3.2.3.

Le même canevas de présentation est utilisé. Nous donnons :

- le nombre de configurations atteignables et le temps de calcul nécessaire ;
- les résultats concernant la détection d'éventuels deadlocks ainsi que des informations concernant le diagnostic ;
- les incohérences concernant les états non atteignables et les transitions non tirables ; rappelons que ces règles ont été présentées en section 5.4.1 ;
- des remarques sur les résultats.

6.3.2.1 Analyse sans modification du modèle

Calcul du graphe d'atteignabilité L'analyse a montré que 152 configurations sont atteignables. Le calcul a été réalisé en 0.21s.

Propriétés de deadlock Trois deadlocks ont été détectés. La table 6.2 donne le temps de détection du deadlock, le temps d'établissement du diagnostic ainsi que le nombre de transitions d'UML tirées pour atteindre le deadlock ; nous rappelons que le but du diagnostic est de fournir des informations sur les changements de configurations qui amènent de l'état initial à la situation de deadlock (cf. annexe E) ;

Tps de détection du deadlock	Diagnostic	
	Tps de réalisation	Nbre de transitions tirées
0,006s	0,05	5
0,036s	4,14s	21
0,18s	3,5s	14

Tab. 6.2 – Informations relatives à la détection de deadlocks et à leur diagnostic

États non atteignables et transitions non tirables 19 états sont non atteignables et 60 transitions sont non tirables. Le temps de détection est respectivement de 6,2s et 20s.

Remarques Ces résultats peuvent sembler étonnants car le modèle contient un grand nombre d'incohérences. Nous rappelons cependant que cette analyse fait intervenir des modèles dont l'environnement n'est pas modélisé. Il est donc logique par exemple

que le système attende des activations de son environnement et que celui-ci se trouve en situation de deadlock. Ceci se traduit également par le non respect de la règle de cohérence structurelle qui dit que « tout message consommé doit être produit ». Cette règle est en effet enfreinte pour sept messages.

6.3.2.2 Analyse avec disponibilité systématique des messages de l'environnement

Calcul du graphe d'atteignabilité Le nombre de configurations atteignables est conséquent puisqu'il en existe 178 973. Le temps de calcul est ici de 11 minutes 13s.

Propriétés de deadlock Aucun deadlock n'a été détecté.

États non atteignables et transitions non tirables Aucune incohérence n'a encore une fois été détectée.

Remarques Cette analyse est quelque peu optimiste puisqu'elle suppose que tous les messages attendus sont envoyés par l'environnement sont présents. Elle a cependant permis de valider les différentes interactions entre les objets dont le comportement est modélisé.

Le nombre de configurations atteignables montre la complexité du comportement des modèles UML et donc la nécessité d'automatiser la détection des problèmes associés. Notons toutefois que l'hypothèse qui veut que les messages produits par l'environnement sont toujours disponibles augmente le nombre des comportements possibles. Le temps de calcul relativement modeste malgré cette complexité montre l'intérêt pratique de la méthode développée dans cette thèse.

L'outil souffre cependant de certaines limites. Considérons par exemple la machine à états de la figure 6.2 qui modélise l'algorithme suivant :

```
X:=0;
Tq X<1000 do
   X:=X+1;
od
(...)
```

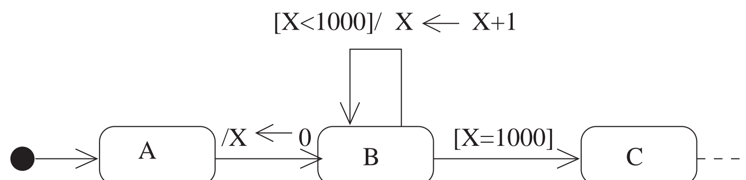


FIG. 6.2 – Représentation graphique des nœuds pouvant apparaître dans les machines à états

Cet algorithme et son modèle se comportent de la même manière quand on a $X \in [0..999]$: l'entier X est incrémenté. Or, dans notre cas, l'outil fait la distinction entre toutes ces valeurs. Lorsque l'état `B` est actif, une configuration atteignable est calculée pour $X = 0$, $X = 1$, ..., $X = 999$. Afin de rendre compte du calcul effectué, notons $post(\mathcal{C}_n)$ l'ensemble des configurations atteignables à partir de la configuration $\mathcal{C}_n$ et par le tir de la transition UML T. Nous représentons la configuration de la machine à état

$\mathcal{C}_n$ par le doublet $< S, V >$ où S est l'état actif et V est la valeur de la variable X. Le calcul de l'ensemble des configurations atteignables par l'outil tel qu'il est actuellement aboutit au résultat suivant :
$\mathcal{C}_0 =< init, _ >$ (état initial)
$post(\mathcal{C}_0) = \{< A, _ >= \mathcal{C}_1\}$
$post(\mathcal{C}_1) = \{< B, 0 >= \mathcal{C}_2\}$
$post(\mathcal{C}_2) = \{< B, 1 >= \mathcal{C}_3\}$
(...)
$post(\mathcal{C}_{1000}) = \{< B, 999 >= \mathcal{C}_{1001}\}$
$post(\mathcal{C}_{1001}) = \{< B, 1000 >= \mathcal{C}_{1002}\}$
$post(\mathcal{C}_{1002}) = \{< C, 1000 >= \mathcal{C}_{1003}\}$
1004 configurations atteignables sont ainsi calculées.

Il paraît ici intéressant d'utiliser une expression symbolique pour représenter la valeur de la variable X afin de regrouper l'ensemble des configurations $\{\mathcal{C}_2, \mathcal{C}_3, ..., \mathcal{C}_{1001}\}$ en une seule configuration. Le calcul effectué serait alors :
$\mathcal{C}_0 =< init, _ >$ (état initial)
$post(\mathcal{C}_0) = \{< A, _ >= \mathcal{C}_1\}$
$post(\mathcal{C}_1) = \{< B, [0..999] >= \mathcal{C}_2\}$
$post(\mathcal{C}_2) = \{< B, [0..999] >= \mathcal{C}_2 \ , \ < B, 1000 >= \mathcal{C}_3\}$
$post(\mathcal{C}_3) = \{< C, 1000 >= \mathcal{C}_4\}$
Ainsi, l'expression symbolique de la valeur de la variable X a permis de calculer l'ensemble des configurations atteignables par le calcul de seulement 5 configurations, d'où un gain important en terme d'espace mémoire.

Le model-checking utilisant ce type de technique est appelé model-checking symbolique [36, 31]. L'intérêt est de regrouper dans une même configuration un ensemble potentiellement infini de configurations. Dans l'exemple, on voit que la configuration $\mathcal{C}_2$ obtenue lors du calcul symbolique représente l'ensemble des configurations $\{\mathcal{C}_2, \mathcal{C}_3, ..., \mathcal{C}_{1001}\}$ obtenu lors du premier calcul. Ceci permet non seulement d'améliorer les performances mais aussi d'analyser des systèmes non finis. Certains travaux comme [53] s'intéressent d'ailleurs au couplage possible entre model-checking symbolique et CLP. Intuitivement, ceci est réalisé en utilisant les solveurs de contraintes. Par exemple, l'expression $X \in [0..999]$ peut être représentée par la conjonction de contraintes « $X \geq 0 \wedge X \leq 999$ ». Ceci fait parti des perspectives de cette thèse. Outre l'intérêt en terme de performances, ce type de manipulation permettrait par exemple de réaliser des vérifications sur des extensions temps-réel d'UML où le temps serait modélisé par des horloges. La valeur des horloges serait alors prise en compte par des contraintes. [21] a utilisé cette approche.

6.3.2.3 Analyse avec modélisation de l'environnement

Pour réaliser cette analyse, le comportement de l'environnement a été modélisé par des machines à états déduites à partir des diagrammes de séquences. Ceci n'est pas automatisé. Dans cette analyse nous considérons que l'environnement est éternel. Il est modélisé par une machine à états qui contient 5 régions différentes. Chaque région est chargée d'envoyer un ou plusieurs signaux aux objets de l'application que nous voulons valider.

Calcul du graphe d'atteignabilité Le modèle peut atteindre 659 configurations différentes qui sont calculées en 19s.

Propriétés de deadlock L'analyse a révélé 16 deadlocks en un temps total de 21s.
La méthode utilisée pour réaliser le diagnostic est parfois coûteuse en temps. Le diagnostic du deadlock qui a pris le plus de temps a été calculé en 396s. La longueur de la trace étant pourtant modeste puisqu'elle contenait 20 configurations. Le diagnostic étant une étape importante dans la vérification du comportement des systèmes, elle doit être améliorée. Des travaux se sont intéressés au problème [68, 62]. [62] décrit une voie qui semble prometteuse car elle permet de construire le diagnostic au fur et à mesure de l'analyse sans toutefois en dégrader significativement les performances. Pour le moment, le diagnostic est réalisé par un prédicat qui recherche le chemin de l'état initial à l'état redouté dans le graphe d'atteignabilité. Ce prédicat se rapproche par exemple de celui présenté par [76](p.94) et est présenté en annexe C.4.

États non atteignables et transitions non tirables Le nombre d'états non atteignables et de transitions non tirables est important. Les causes sont cependant connues. En effet, le comportement de l'environnement a été réalisé en fonction des diagrammes de séquences uniquement. Or, ceux-ci souffrent d'incomplétudes : deux messages consommés sont non produits. Aussi, nous conseillons de vérifier les règles de cohérence structurelle avant les règles de cohérence comportementale. Par exemple, considérons le modèle de la figure 6.3 et supposons que la transition de ce modèle soit non tirable. Les causes de

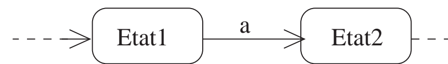


Fig. 6.3 – Exemple d'une transition activée sur un événement

cette incohérence peuvent être multiples. Par exemple, si l'état `Etat1` n'est pas atteignable, si l'objet ne se trouve jamais dans la configuration où l'état `Etat1` est actif et où le signal `a` est présent dans la pile des messages, etc. Or, l'analyse structurelle permet de s'assurer que « tout message consommé dans une classe est envoyé à destination de cette classe ». Si cette règle n'est pas vérifiée, ceci aboutit inévitablement à une transition non tirable. Dans l'exemple présenté ici, il est clair que si aucune classe ne spécifie l'envoi du signal « `a` » à la classe qui contient la machine à états, la transition activée par ce signal sera non tirable. On voit donc que des incohérences comportementales peuvent être causées par des incohérences structurelles. De plus, l'analyse structurelle ne requiert pas la construction du graphe d'atteignabilité, elle est donc plus rapide, et le traitement des incohérences structurelles est moins difficile car les configurations dynamiques du modèle n'interviennent pas. Il est donc préférable de vérifier les incohérences structurelles avant les incohérences comportementales.

6.4 Conclusion

Les expérimentations exposées dans ce chapitre montrent l'intérêt pratique de l'approche développée pendant cette thèse. Les performances obtenues sont en effet tout à fait acceptables comme le montre le temps de calcul de plus de 170 000 configurations atteignables en 11 minutes 13s (cf. section 6.3.2.2). Les limitations rencontrées sont dues plus à des limites d'espace mémoire qu'au temps de calcul.

Nous pouvons cependant regretter le manque de « benchmark » dans le milieu UML qui interdit la comparaison avec d'autres outils d'analyse.

Un aspect important est la hiérarchisation dans l'ordre d'application des règles. Nous conseillons en effet de vérifier en premier lieu la cohérence structurelle des modèles avant d'en vérifier la cohérence comportementale. Les incohérences structurelles sont en effet plus faciles à localiser et à traiter et peuvent être la cause d'incohérences comportementales.

Il paraît possible d'améliorer le temps de calcul pour établir le diagnostic des incohérences comportementales en utilisant les résultats décrits dans [62]. Ce diagnostic est important car les changements de configuration du système qui mènent de l'état initial à la situation redoutée peuvent être nombreux et durs à appréhender sans outillage adapté.

Au cours de cette expérimentation nous nous sommes limités à la détection d'incohérences qui sont des propriétés que tout modèle UML doit respecter. Notre méthode permet également la vérification de propriétés propres à une application. En l'état actuel cependant, ces propriétés sont limitées aux propriétés de sûreté qui expriment la non atteignabilité d'une configuration redoutée. La possibilité d'exprimer des propriétés autres que sur les configurations redoutées semble possible. Il serait par exemple intéressant de pouvoir vérifier des propriétés de vivacité sur des traces de longueurs quelconques. Comme expliqué en 5.4.2, c'est une possibilité mais les performances alors obtenues ne permettent pas l'analyse de modèles complexes. Afin de combattre l'explosion combinatoire, l'interprétation abstraite est une technique qui a déjà fait ses preuves et nous envisageons donc de la mettre en œuvre. Cette technique permet en effet de d'observer le comportement du système à un certain niveau d'abstraction en ignorant les détails d'implantation comme la valeur d'attributs qui n'influencent pas le comportement du modèle. Notons cependant que cela fait parti des perspectives de cette thèse et que le sujet mérite d'être approfondi.

De plus, au cours de cette expérimentation nous n'avons pas utilisé la capacité des systèmes logiques à résoudre des systèmes de contraintes. Cette capacité pourra être utilisée pour réaliser du model-checking symbolique. Ceci permet de représenter un ensemble de configurations par une seule configuration. Elle est réalisée par une expression symbolique telle que les contraintes sur les variables. Ceci augmenterait considérablement le champ d'investigation de notre outil puisque cela permettrait de prendre en compte la modélisation du temps par exemple.

Chapitre 7

Conclusion

7.1 Synthèse

La complexité croissante des logiciels développés a conduit à l'utilisation de langages de modélisation permettant de spécifier des vues abstraites du système. L'expression en multiples vues permet en effet de décrire de manière séparée les différents aspects du système et donc d'en maîtriser le complexité. UML est le langage de modélisation qui s'est imposé dans l'industrie ces dernières années et qui répond à ce besoin. D'autre part, les fonctionnalités du logiciel sont parfois critiques et un haut niveau de fiabilité doit donc être respecté. Évaluer la correction des modèles dès leur création permet d'apporter un niveau d'assurance sur le bon fonctionnement du système final. Notre approche est d'évaluer la correction du système au moyen de propriétés que doit respecter tout modèle UML qui sont appelées règles de cohérence.

Cette thèse se situe dans un contexte de gestion du risque. En effet, les incohérences liées au langage UML sont associées aux constructions de ce langage UML qui peuvent être utilisées et assemblées de manière valide ou non. La présence d'incohérences est donc potentielle. D'autre part, la présence d'une incohérence peut avoir des effets plus ou moins graves sur la suite du développement. Or, la notion de risque consiste à coupler le degré de vraisemblance d'un événement dommageable à la gravité du dommage engendré. Il nous est donc apparu adapté de gérer le risque d'incohérence. Deux étapes de la gestion du risque d'incohérence ont été abordées au cours de cette thèse, l'identification du risque d'incohérence d'une part et le traitement du risque d'autre part.

L'identification des incohérences a été réalisée par un processus rigoureux et redondant d'analyse de la norme. Le document produit [50] contient 650 règles de cohérence. Ceci représente la base de données de règles la plus complète à notre connaissance. Ces règles ont été utilisées pour réaliser des modèles incohérents afin de comparer la capacité de détection des outils industriels. Nous pensons en effet que ce type de benchmark est intéressant pour aider les chefs de projets à réaliser des choix sur les outils à utiliser. Il nous a également permis de montrer la faiblesse d'outils industriels en terme de détection.

La contribution de cette thèse à l'étape de traitement des incohérences est ensuite réalisée par la mise au point d'une méthode de détection automatique de ces incohérences. La méthode développée permet de déceler avec un seul formalisme aussi bien les incohérences structurelles que comportementales. La vérification de règles de cohérence structurelles ne tient compte que de la structure des modèles UML alors que la vérification de règles de cohérence comportementale doit également intégrer la sémantique opérationnelle d'UML, c'est-à-dire le comportement des modèles UML.

La pierre angulaire de cette méthode est basée sur un encodage d'UML en CLP (Constraint Logic Programming). L'intérêt de l'encodage proposé est que toutes les données relatives au modèle UML sont présentes dans la représentation en CLP. Ceci permet l'expression de raisonnements en CLP sur l'intégralité du modèle UML. Nous avons utilisé cette capacité pour exprimer et vérifier les règles de cohérence structurelle.

Fournir un diagnostic précis de l'incohérence détectée est essentiel car c'est à partir de ce diagnostic que pourra être traitée l'incohérence. L'emploi de notre méthode offre un retour sur le modèle aisé car les éléments manipulés par notre checker sont les éléments du modèle UML.

Un autre aspect positif de notre méthode est que le métamodèle est encodé dans sa totalité. Il est ainsi possible d'écrire des règles CLP sur les parties abstraites de celui-ci (par exemple sur les espaces de nommages). Ceci garantit l'application exhaustive des règles sur l'ensemble des éléments concernés (par exemple les classes, les paquetages, etc.).

Enfin, l'encodage développé l'a été indépendamment du métamodèle d'UML. Il pourra donc être appliqué sur les évolutions futures du langage UML, mais aussi sur tout langage décrit par un métamodèle comme par exemple AADL ou des documents XML.

La vérification des règles de cohérence comportementale nécessite l'expression de la sémantique opérationnelle du langage UML. Comme souvent dans le cas des systèmes discrets la sémantique opérationnelle est décrite au moyen de règles de changement de configurations. Nous avons donc défini dans un premier temps les valeurs qui caractérisent la configuration dynamique d'un modèle UML puis nous avons exprimé les règles de changement de configuration au moyen de CLP. La vérification de règles de cohérence comportementale est alors possible. Durant cette thèse nous nous sommes intéressés aux règles de cohérence comportementale qui contribuent au développement d'applications certifiables par la détection de transitions et d'états non atteignables qui sont la source potentielle de génération de code mort ou par la détection d'inter-blocages entre les différents objets de l'application. Une fois encore le diagnostic de l'incohérence a fait l'objet d'une attention particulière. Lorsqu'une configuration redoutée est détectée, le diagnostic permet de reconstruire la trace des configurations qui mènent de l'état initial jusqu'à la situation redoutée. Une fois cette trace obtenue différentes informations utiles à l'utilisateur peuvent en être extraites comme par exemple la succession des transitions des machines à états tirées.

Plus généralement, l'approche exposée propose le couplage de deux domaines de recherche. Le premier domaine est l'IDM (Ingénierie Dirigée par le Modèles) dont les concepts clefs sont les modèles, les métamodèles et les méta-métamodèles. Le deuxième

domaine consiste à utiliser les CLP comme langage d'analyse de propriétés. Ce domaine de recherche est actif et semble être prometteur comme le montre les travaux dédiés à cette thématique : édition d'un livre [20], organisation de workshops (eg. [63, 46, 19]), projets universitaires (eg. [13]). Notons toutefois qu'en l'état actuel de nos travaux la programmation logique est utilisée sans le solveur de contraintes.

Nous proposons en fait de coupler les techniques de métamodélisation et d'expression de la sémantique en CLP. Cette approche est nouvelle dans le domaine UML, la sémantique opérationnelle d'UML n'ayant à notre connaissance jamais été exprimée au moyen de règles (C)LP. La méthode proposée a été développée pour les modèles UML mais celle-ci se situant au niveau métamodèle, elle est applicable sur tout langage décrit par un métamodèle.

De plus, nous nous sommes limités aux propriétés génériques mais des propriétés spécifiques à une application comme l'impossibilité d'atteindre une configuration redoutée pourraient aussi être vérifiées.

Enfin, la méthode mise au point durant cette thèse a été outillée afin d'en tester l'intérêt pratique. Deux modèles ont été analysés, le premier est le modèle UML d'un problème universitaire, le deuxième modèle est issu du milieu industriel avionique. Les performances obtenues lors des différents tests sont encourageantes. De plus, l'analyse du modèle industriel a conduit à la rédaction de guides de modélisation qui permettent l'analyse automatique des modèles UML constitués de classes et de machines à états. Il s'est également avéré pratique de vérifier en premier lieu les règles de cohérence structurelle avant les règles de cohérence comportementale. L'utilisation d'un seul formalisme pour réaliser toutes les analyses prend alors tout son intérêt car elle limite les transformations des modèles UML ainsi que l'expertise nécessaire à l'écriture des règles.

7.2 Perpectives

Concernant l'identification des règles de cohérence les perspectives portent surtout sur la maintenance du document comme l'intégration de remarques éventuelles de relecteurs, la mise à jour du document vis-à-vis des nouvelles versions d'UML, etc. L'expression en français des règles est aussi une limite à laquelle il faudrait pallier. Afin de faire profiter l'ensemble de la communauté UML de cette base de données de règles il faudrait en effet les traduire en anglais. Une expression formelle de ces règles en vue d'une détection automatique est également possible.

L'étape de vérification automatique des incohérences développée au cours de cette thèse ouvre de nombreuses de perspectives.

La définition d'un langage d'action concret pour UML est tout d'abord nécessaire pour exprimer formellement les différentes actions à réaliser au sein des modèles eux-mêmes. Ce langage pourra alors être transformé dans le langage requis par l'outil d'analyse utilisé.

La formalisation de la sémantique opérationnelle d'UML a soulevé un certain nombre de points de variation sémantique. Il serait intéressant de modéliser ces points de variation sémantique comme expliqué dans [12]. Ceci permettrait aux utilisateurs de l'outil de choisir la sémantique opérationnelle voulue.

Le diagnostic des incohérences comportementales est précis mais sa réalisation est parfois longue et est établie une fois que la configuration redoutée a été trouvée. Il paraît cependant possible d'améliorer les temps de calcul en construisant la trace au fur et à mesure de l'analyse [62].

Au cours de cette thèse nous n'avons pas utilisé la capacité des systèmes logiques à résoudre des systèmes de contraintes. Cette capacité pourra être utilisée pour réaliser du model-checking symbolique. La manipulation d'expressions symboliques permet de représenter un ensemble potentiellement infini de configurations par une seule configuration. Cette approche semble tout à fait abordable et sa faisabilité en CLP a déjà été montrée pour la vérification d'automates à états temporisé par exemple. Deux motivations importantes suscitent l'intérêt pour le model checking symbolique. D'une part, les performances de la vérification sont accrues et d'autre part la classe des systèmes vérifiables est augmentée. Ceci permettrait par exemple la prise en compte de contraintes temporelles, et donc la vérification de propriétés liées à des profils temps-réels d'UML. À l'heure actuelle, aucun profil n'a été étudié. Des propriétés extra-fonctionnelles comme la précision du résultat [61] peuvent également être exprimées et vérifiées par le raisonnement symbolique permis par les CLP.

Une autre perspective serait de définir une logique temporelle applicable sur les modèles UML. Cette approche semble être faisable car les CLP ont par exemple servi d'implantation à la logique temporelle « $\mu - calculus$ » [18]. Comme expliqué en 5.4.2 cette approche n'a pas été approfondie mais les quelques tests réalisés ont montré des performances faibles qu'il faudra donc chercher à améliorer si nous voulons poursuivre dans cette voie. Comme expliqué plus tôt, le model checking symbolique semble être prometteur. Ceci risque cependant de ne pas être suffisant, nous prévoyons donc de nous servir des techniques d'interprétations abstraites. En effet, pour le moment l'ensemble des informations du modèle UML sont prises en compte même celles qui n'influencent pas ou peu le comportement de celui-ci. Il serait donc intéressant de réaliser des analyses sur des configurations faisant abstraction de ces détails. Cette technique a déjà montré de très bons résultats pour l'analyse de programmes notamment et nous pensons qu'elle peut être retranscrite à l'analyse de modèles UML.

Annexe A

Définition de l'encodage des modèles UML en XMI

Cette annexe fournit un exemple de règle EBNF qui spécifie la génération d'un document XMI afin d'en faire conprendre les mécanismes. Un point important est que ces règles EBNF ne dépendent pas d'un métamodèle. Elles permettent en fait d'encoder en XML tout modèle dont le langage est défini par un métamodèle exprimé en MOF.

Pour cela, nous considérons l'exemple du modèle UML et de son encodage présenté par la figure A.1.

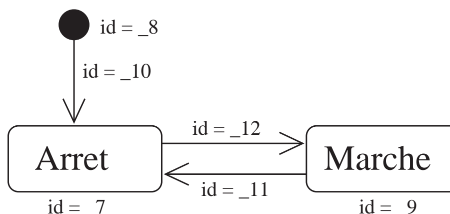
Ce modèle est une instance du métamodèle de la figure A.2. Cet encodage mérite quelques explications. La présence d'une machine à états est spécifiée par la ligne `(1)`. Celle-ci contient une région (ligne `(2)`). C'est cette région qui contient les différents états (lignes `(3)`, `(7)` et `(8)`) et les différentes transitions (lignes `(9)`,`(10)` et `(11)`). Ceci est bien conforme au métamodèle. Plus précisément, la ligne `(3)` spécifie la présence de l'état (`uml:type=State`) nommé `Arret` (`name=Arret`).

Considérons maintenant la règle EBNF ci-dessous inspirée de la spécification du XMI [60] :

```
2 :XMIElement : := 2a :XMIObjectElement
                 | 2c :XMIReferenceElement

2a :XMIObjectElement : := ( "<" xmiName 2d :TypeAttrib 2e :XMIRefAttribute (2b :FeatureAttributes)* ">"
                                       (2 :XMIElement)*
                           "</" xmiName ">" )

2b :FeatureAttributes : := ( xmiName " =' " value " ' " )

2c :XMIReferenceElement : := "<" xmiName "xmi :idref='" idValue "'/>"

2d :TypeAttrib : := "xmi :type=' " xmiName "'"

2e :XMIRefAttribute : := "xmi :id= ' " ElementId " ' "
```

Nous expliquons ces règles EBNF en se référant à l'exemple d'encodage.

- `2` spécifie qu'un élément XMI est soit un objet XMI (`2a`) soit une référence à un objet XMI (`2c`) ;
- `2a` spécifie qu'un objet XMI :
  1. a un nom (***xmiName***) comme par exemple `subvertex` de la ligne `(3)` de l'encodage ; l'explication concernant les références à ***xmiName*** est donnée plus tard ;
  2. a un type (règle `2d`), par exemple `State` de la ligne `(3)` ;




```
(...)
<nestedClassifier xmi:type="StateMachine" xmi:id="_5">................(1)
  <region xmi:type="Region" xmi:id="_6">..............................(2)
    <subvertex xmi:type="State" xmi:id="_7" name="Arret">.............(3)
        <outgoing xmi:idref="_11"/>...................................(4)
        <incoming xmi:idref="_10"/>...................................(5)
        <incoming xmi:idref="_12"/>...................................(6)
    </subvertex
    <subvertex xmi:type="Pseudostate" xmi:id="_8"  kind="initial"......(7)
        <outgoing xmi:idref="_10"/>
    </subvertex>
    <subvertex xmi:type="State" xmi:id="_9" name="Marche">............(8)
        <outgoing xmi:idref="_12"/>
        <incoming xmi:idref="_11"/>
    </subvertex>
    <transition xmi:type="Transition" xmi:id="_10">...................(9)
       <source xmi:idref="_8"/>
       <target xmi:idref="_7"/>
    </transition>
    <transition xmi:type="Transition" xmi:id="_11">...................(10)
       <source xmi:idref="_7"/>
       <target xmi:idref="_9"/>
    </transition>
    <transition  xmi:type="Transition" xmi:id="_12">..................(11)
       <source xmi:idref="_9"/>
       <target xmi:idref="_7"/>
    </transition>
 </region>
</nestedClassifier>
(...)
```

Fig. A.1 – Un modèle UML et son encodage en XMI

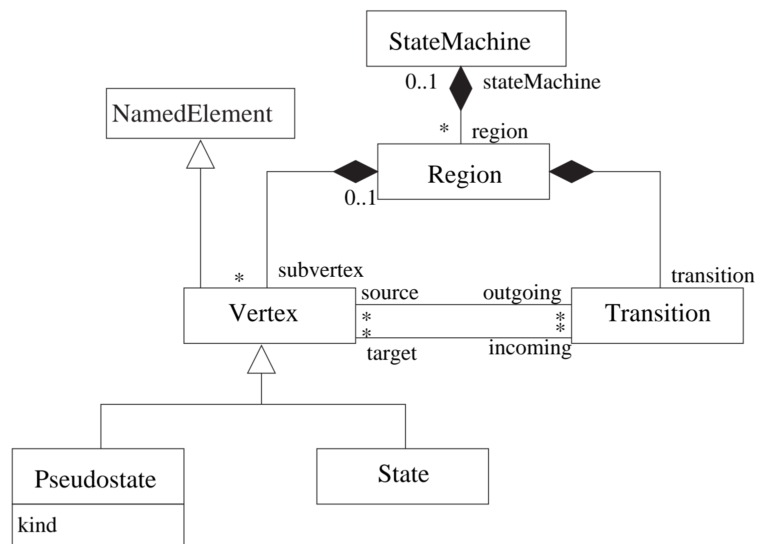


Fig. A.2 – Sous-ensemble du métamodèle pour les machines à états

3. a un identifiant (règle 2e), par exemple l'identifiant de l'élément de la ligne (3) de l'encodage est "_7" ;
4. peut être caractérisé par la valeur de ses méta-attributs (règle 2b), par exemple la valeur du méta-attribut Name de l'état de la ligne (3) de l'encodage est Arret ;
5. peut contenir d'autres éléments qui sont soit des références (par exemple les lignes (4), (5) et (6) sont des références aux transitions entrantes et sortantes contenues dans la spécification de l'état Arret présentée ligne (3)), soit la déclaration de nouveaux éléments (par exemple les éléments états sont déclarés au sein de la déclaration de la région).

Ces règles sont définies au niveau M3 car elles ne sont pas spécifiques à un métamodèle. En effet, ces règles font référence à *xmiName* qui correspond en fait à une chaîne de caractères qui dépend du métamodèle. Par exemple, la règle 2d : "xmi :type=' " *xmiName* "'" est instanciée par la ligne (3) xmi :type="State" où State est une métaclasse du métamodèle. Un autre exemple est l'instanciation de la règle 2c par la ligne (3) <subvertex (...), en effet subvertex est le nom de la fin d'association du métamodèle entre une région (ligne (2)) et un état (ligne (3)).

Le principe de génération de schémas XML suit les même principes.

Annexe B

Présentation de l'outil de traduction UML → CLP

Le but de cette annexe est donner une vue d'ensemble de l'implantation de l'outil qui permet de traduire les modèles UML encodés en XML en clauses de programmation logique. La section B.1 présente l'analyse du métamodèle et la section B.2 présente l'analyse du modèle.

B.1 Analyse du métamodèle

Comme expliqué en section 4.2, l'analyse du métamodèle a pour but d'extraire les méta-faits destinés à être instanciés et les règles qui reconstruisent les parties abstraites du métamodèle. Cette section montre l'outillage associé aux concepts introduits plus tôt. Nous limiterons nos explications aux méta-faits qui représentent les métaclasses et aux règles qui représentent les méta-généralisations. Afin de montrer les différentes étapes de l'analyse du métamodèle nous nous appuierons sur un sous-ensemble du métamodèle et sur son encodage. Ce sous-ensemble est présenté à la section B.1.1. La section B.1.2 présente ensuite une vue d'ensemble de l'analyse du métamodèle et détaille ses différentes étapes. Ces étapes sont ensuite explicitées par les sections B.1.3 à B.1.6.

B.1.1 Exemple de métamodèle et de son encodage

Cette section présente un sous-ensemble du métamodèle et son encodage. Ce sous ensemble à trait à la métaclasse `State` et est présenté par la figure B.1.

Le métamodèle analysé est encodé en XMI. Le listing ci-dessous est le sous-ensemble qui encode la classe `State`.

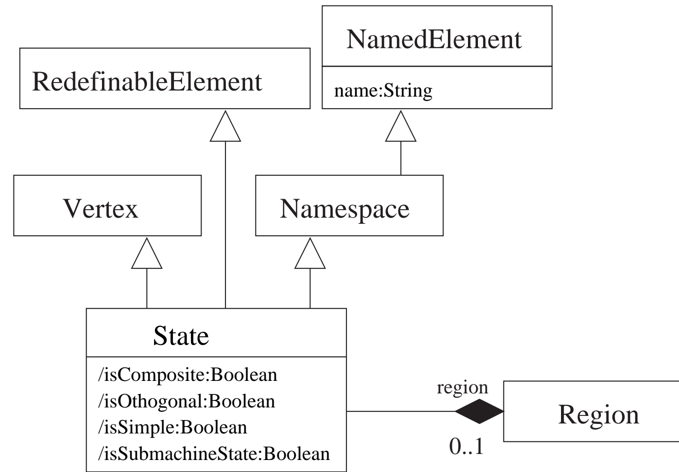


FIG. B.1 – Métamodèle pour la métaclasse `State`

```
1  <ownedMember xmi:type="uml:Class" xmi:id="_m9bP8q86EdiEh75YJ_3n8g" name="State">
2      <generalization xmi:id="idGen1" general="idNamespace"/>
3      <generalization xmi:id="idGen3" general="idVertex"/>
4      <generalization xmi:id="idGen2" general="idRedefinableElement"/>
5      <ownedAttribute xmi:id="idIsComposite" name="isComposite" type="idBooleanType"
6                      isReadOnly="true" isDerived="true">
7      </ownedAttribute>
8      <ownedAttribute xmi:id="idIsOrthogonal" name="isOrthogonal" type="idBooleanType"
9                      isReadOnly="true" isDerived="true">
10     </ownedAttribute>
11     <ownedAttribute xmi:id="idIsSimple" name="isSimple" type="idBooleanType"
12                     isReadOnly="true" isDerived="true">
13     </ownedAttribute>
14     <ownedAttribute xmi:id="idIsSubmachineState" name="isSubmachineState"
15                     type="idBooleanType" isReadOnly="true" isDerived="true">
16     </ownedAttribute>
17      (...)
18     <ownedAttribute xmi:id="idRegion" name="region" type="idClassRegion"
19                     isOrdered="true" association="_m9bQPq86EdiEh75YJ_3n8g"
20                     aggregation="composite">
21       <upperValue xmi:type="uml:LiteralUnlimitedNatural" value="-1"/>
22     </ownedAttribute>
23      (...)
24 </ownedMember>
```

Les lignes `2` à `4` expriment qu'un état est un espace de nommage, un élément redéfinissable ainsi qu'un sommet graphique. Les lignes `5` à `23` spécifient les différents attributs. Deux types d'attributs sont distingués. D'une part, les attributs qui ne sont pas le fruit d'une association. C'est par exemple le cas de l'attribut `isComposite` de la ligne `5`. D'autre part, les attributs déduits d'une fin d'association navigable, c'est le cas de l'attribut `region` de la ligne `18`. Ce type d'attribut fait référence à une association, contrairement aux attributs qui ne découlent pas d'une association.

Faisons tout de suite remarquer que l'encodage de la métaclasse `State` ne rend pas compte de tous ses méta-attributs. Par exemple, l'attribut `name` n'y est pas spécifié. En fait cet attribut est hérité des classes `namespace` et `namedElement`. Pour obtenir l'ensemble des attributs d'un méta-classe, il faudra donc tenir compte des méta-généralisations. Ainsi le méta-fait non complet qui représente la méta-classe `State` sera : `state(Id,IsComposite,IsOrthogonal,IsSimple,IsSubmachineState).`
En supposant que le seul attribut hérité par cette méta-classe et l'attribut `name`, le méta-

fait complet sera donc :
`state(Id,IsComposite,IsOrthogonal,IsSimple,IsSubmachineState,Name).`

B.1.2 Vue d'ensemble de l'analyse du métamodèle

La figure B.2 montre les classes qui interviennent dans l'analyse du métamodèle. `MetaModelAnalyser` est la classe principale. Cette classe est chargée de lire le métamodèle et d'en extraire les données voulues. `MetaModelAnalyser` contient deux ensembles de méta-faits. L'ensemble nommé `EnsembleMetaFactSSGene` contient des méta-faits incomplets car ils ne disposent pas de tous les attributs hérités par les méta-faits plus généraux. [1] Au contraire l'ensemble `EnsembleMetaFactAvecGene` contient des méta-faits complets. Cette classe contient également un ensemble de `Generalization` qui représentent les méta-généralisations entre méta-classes.

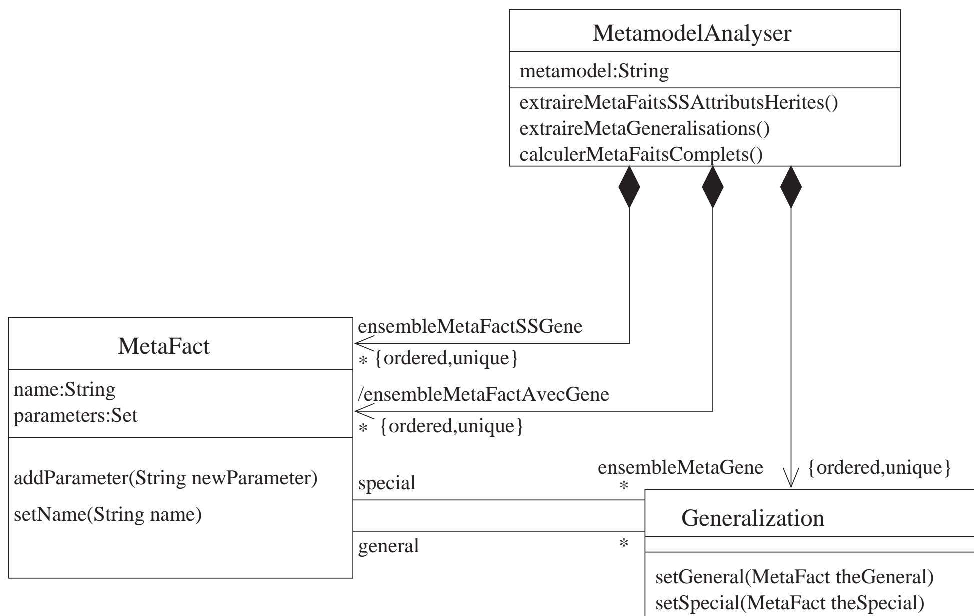


Fig. B.2 – Classes pour l'analyse du métamodèle

La figure B.3 montre l'enchaînement des actions pour obtenir l'ensemble des méta-faits qui représentent les méta-classes et l'ensemble des règles qui représentent les méta-généralisations. La première étape est l'extraction des méta-faits sans tenir compte des attributs hérités. Cette étape est détaillée en section B.1.3. La deuxième étape est l'extraction des généralisations entre les méta-classes et est approfondie en B.1.4. Le calcul des méta-faits en tenant compte des attributs hérités est ensuite réalisé. Ceci est possible car les méta-généralisation ont été extraites (cf. section B.1.5). Les méta-généralisatons sont ensuite modifiées pour pointer sur les méta-faits complets. Il est alors possible d'écrire les règles qui encodent ces méta-généralisations (cf. section B.1.6).

[1] Rappelons qu'un méta-fait représente une métaclasse.

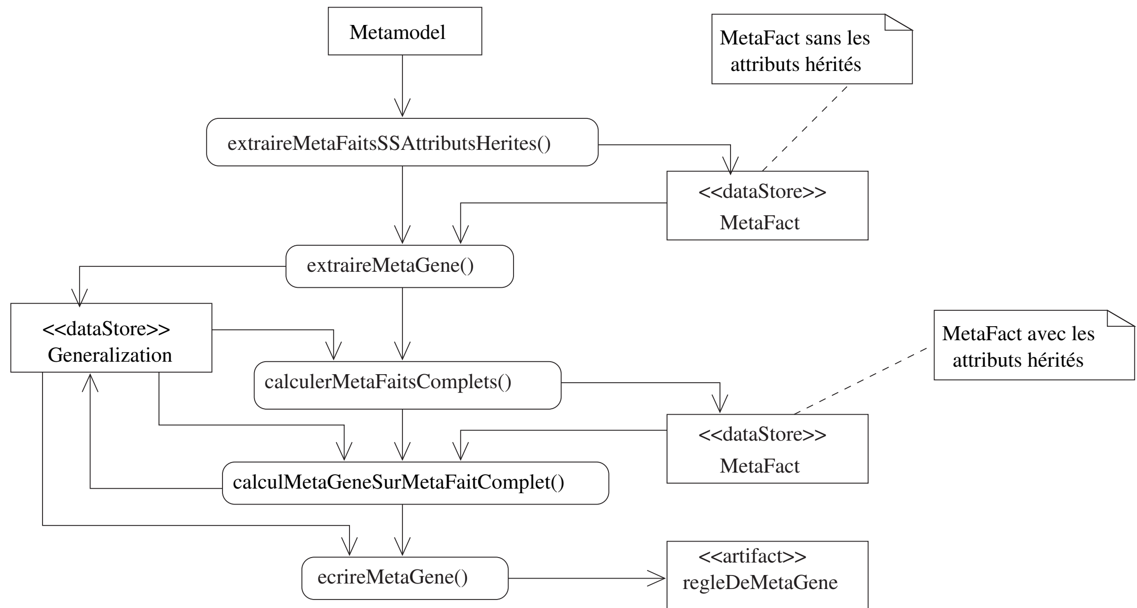


Fig. B.3 – Analyse du métamodèle

B.1.3 Extraction des méta-faits sans attributs hérités

La méthode qui permet d'extraire les meta-faits sans tenir compte des attributs hérités est présentée ci-dessous.

```
1  public void extraireMetaFaitsSansAttributHerite()
2  {
3    Document parsingDuMeta=null;
4    SAXReader reader = new SAXReader();
5
6    //Lecture du meta modele
7    try {
8      parsingDuMeta = reader.read(metaModel);
9    } catch (DocumentException e) {
10     e.printStackTrace();}
11
12   //Recuperation de l'element racine du meta modele
13   Element racineUml2 = parsingDuMeta.getRootElement();
14
15   //Recuperation des toutes les meta-classes par le parcours
16   //des elements contenus dans la paquetages principal
17   //Une metaCLasse est de type "uml:Class"
18   for(Iterator i = racineUml2.elementIterator("ownedMember"); i.hasNext(); )
19   {
20     Element ownedMember = (Element) i.next();
21     Attribute memberType = ownedMember.attribute("type");
22
23     //Pour chaque meta classe on construit son meta-fait
24     if (memberType.getValue().equals("uml:Class"))
25     {
26       //Recuperation du nom du meta fait
27       Attribute memberName = ownedMember.attribute("name");
28       //Constrution du meta fait avec son nom qui doit commencer par une minuscule
29       String nomMetaFait = memberName.getValue();
30       nomMetaFait =
31       nomMetaFait.substring(0,1).toLowerCase().concat(nomMetaFait.substring(1));
32
33       MetaFact metaFaitEnConstruction = new MetaFact(nomMetaFait);
34
35       //Recuperation des attributs de la meta-classe
36       for (Iterator itAttribut=ownedMember.elementIterator("ownedAttribute") ;
```

```
37            itAttribut.hasNext();)
38        {
39          Element ownedAttribute = (Element) itAttribut.next();
40          //On s'assure que ce n'est pas un attribut de fin de association
41          if (ownedAttribute.attribute("association")==null)
42          {
43            String nomAttribut = ownedAttribute.attribute("name").getValue();
44            //Ajout du nom du meta attribut au parametre du meta-fait
45            //On met la premiere Lettre en majuscule
46            nomAttribut =
47            nomAttribut.substring(0,1).toUpperCase().concat(nomAttribut.substring(1));
48            metaFaitEnConstruction.addParameter(nomAttribut);
49          }
50        }
51        //Ajout du meta fait a la base de meta-faits
52        metaFactsSSHeritage.add(metaFaitEnConstruction);
53      }
54    }
55  }
```

Les principales étapes sont les suivantes :

1. lecture du metamodèle (lignes 6 à 11) avec l'outil dom4j [22] (notons que dom4j utilise l'outil de lecture de fichier XML appelé SAX) et récupération de l'élément racine du métamodèle (ligne 13) ;
2. pour chaque métaclasse on construit le méta-fait correspondant :
   (a) extraction du nom du méta-fait qui correspond au nom de la classe (lignes 28 à 31) ;
   (b) extraction des attributs du méta-fait qui sont les attributs de la classe qui ne font pas référence à une association (lignes 35 à 50) ;
   (c) ajout du méta-fait dans l'ensemble des méta-faits incomplets (lignes 52).

Comme nous le verrons en section B.1.6, il est nécessaire d'afficher ces méta-faits avec un format compatible aux outils de programmation logique. Ceci est réalisé par la méthode `toString` présentée ci-dessous :

```
1  //transforme un meta fait en chaine et rajoute la fin de chaine passee en
2  //parametre a la fin
3  public String toString(String enFinDeChaine)
4  {
5    String result = new String(name + "("+"Id");
6
7    for(Iterator iterateur = parameters.iterator(); iterateur.hasNext();)
8    {
9      Object unParametre = null;
10     unParametre = iterateur.next();
11     result = result + ","+unParametre.toString();
12   }
13   result = result + ")" + enFinDeChaine;
14   return result;
15 }
```

Appliquées au sous-ensemble du métamodèle présenté à la figure B.1 ces deux routines créent le méta-fait ci-dessous :
`state(Id,IsComposite,IsOrthogonal,IsSimple,IsSubmachineState).` Il est clair que ce méta-fait est incomplet car il ne tient pas compte des attributs hérités par la métaclasse représentée. Par exemple, l'attribut `name` n'est pas formalisé.

Notons qu'un certain nombre de fonctionnalités offertes par dom4j sont utilisées, par exemple la possibilité de parcourir l'ensemble des fils d'un élément XML (ligne 18), d'accéder à un attribut d'un élément XML (ligne 21), etc.

B.1.4 Extraction des méta-généralisations

La méthode ci-dessous permet d'extraire l'ensemble des méta-généralisations du métamodèle.

```
1  //Extraction du metatmodele de l'ensemble des meta-genralisation
2  public void extraireMetaGeneralisations()
3  {
4    Document parsingDuMeta=null;
5    SAXReader reader = new SAXReader();
6
7    //Lecture du meta modele
8    try {
9      parsingDuMeta = reader.read(metamodel);
10   } catch (DocumentException e) {
11     e.printStackTrace();
12   }
13
14   //Recuperation de l'element racine du meta modele
15   Element racineUml2 = parsingDuMeta.getRootElement();
16   //Recuperation des toutes les meta-classes
17   //Une metaCLasse est de type "uml:Class"
18   for(Iterator i = racineUml2.elementIterator("ownedMember"); i.hasNext(); )
19   {
20     Element ownedMember = (Element) i.next();
21     Attribute memberType = ownedMember.attribute("type");
22     Attribute memberName = ownedMember.attribute("name");
23
24     //Pour chaque meta classe on recupere son element general
25     if (memberType.getValue().equals("uml:Class"))
26     {
27       for(Iterator itGene=ownedMember.elementIterator("generalization");
28           itGene.hasNext();)
29       {
30         Element generalizationNode = (Element) itGene.next();
31         Attribute idGeneral = generalizationNode.attribute("general");
32         if(idGeneral!=null)
33         {
34           Generalization laGeneralization= new Generalization();
35           Element general =  trouverClasseAvecId(racineUml2,idGeneral.getValue());
36           if(general!=null)
37           {
38             Attribute generalName = general.attribute("name");
39             //System.out.println("Element general : " + generalName.getValue());
40             //Recherche du meta fait general et du meta fait special
41             //Pour tous les meta-fiats connus
42             for(Iterator itMetaFacts=metaFactsSSHeritage.iterator();
43                 itMetaFacts.hasNext();)
44             {
45               MetaFact metaFaitCourrant=(MetaFact) itMetaFacts.next();
46
47               //Nom du meta fait courrant est egal au nom de la meta classe generale
48               if (metaFaitCourrant.name.compareToIgnoreCase(generalName.getValue())==0)
49               {
50                 //Faire pointer l'attribut general de la generalisation sur le meta
51                 //fait general
52                 laGeneralization.setGeneral(metaFaitCourrant);
53                 metaFaitCourrant.AfficherMetaFact();
54               }
55
56               //Si le nom du meta fait courrant est egal au nom de la classe qui
57               //contient la generalisation, ie. au nom de la classe speciale
58               if (memberName.getValue().compareToIgnoreCase(metaFaitCourrant.name)==0)
59               {
60                 //Faire pointer l'attribut special de la generalisation sur le meta
61                 //fait correspondant
62                 laGeneralization.setSpecial(metaFaitCourrant);
63               }
64             }
65           }
```

```
66            if (laGeneralization.getGeneral()!=null &&
67               laGeneralization.getSpecial()!=null){
68             metaGeneralisationsSSHeritage.add(laGeneralization);}
69          }
70        }
71      }
72    }
73  }
```

Nous ne détaillerons pas les étapes de cette méthode car les concepts sont similaires à ceux utilisés pour l'extraction des méta-faits. Notons juste que la méthode `trouverClasseAvecId` de la ligne `35` utilise l'outil XPATH qui permet de faire des requêtes sur un document XML un peu comme sur une base de donnée.

B.1.5 Calcul des méta-faits complets

Le but de l'étape présentée ici est de calculer les méta-faits complets, c'est-à-dire de calculer l'ensemble des attributs hérités du méta-fait. Ceci est réalisé en utilisant les méta-faits qui ne tiennent pas compte des attributs hérités ainsi que des méta-généralisations.

```
1  //Retourne l'ensemble des parametres d'un metaFait.
2  //Tient compte des parametres herites.
3  private Set parametreHerite(MetaFact metaFact)
4  {
5    //Declaration de l'ensemble qui contiendra l'ensemble des parametres
6    LinkedHashSet ensembleParametres = new LinkedHashSet();
7    //Ajout dans l'ensemble des parametres les parametres non herites
8    ensembleParametres.addAll(metaFact.parameters);
9
10   //Iteration l'ensemble des meta-generalisations pour obtenir tous
11   //les attributs herites
12   for(Iterator itMetaGene=metaGeneralisations.iterator();itMetaGene.hasNext();)
13   {
14     Generalization metaGene=(Generalization) itMetaGene.next();
15
16     //Cas ou la meta-classe speciale correspond à la meta-classe dont on veut
17     //connaitre tous les parametres herites
18     if (metaGene.special.name.compareTo(metaFact.name)==0)
19     {
20       //obtention de tous les parametres par un appel ércursif
21       ensembleParametres.addAll(parametreHerite(metaGene.general));
22     }
23   }
24   return ensembleParametres;
25 }
```

Cette méthode parcourt de manière récursive (ligne `21`) l'ensemble des méta-classes générales (lignes `12` à `23`) dont on veut obtenir l'ensemble des paramètres. Le méta-fait correspondant est donc complet. Par exemple, le méta-fait obtenu pour la métaclasse `State` est le suivant : `state(Id,IsComposite,IsLeaf,IsOrthogonal,IsSimple, IsSubmachineState,Name,QualifiedName,Visibility).` On voit par exemple que l'attribut `name` fait bien parti des attributs de la métaclasse `State`.

La métaclasse `Class` est représentée par le méta-fait :
`class(Id,IsAbstract,IsActive,IsLeaf,Name,QualifiedName,Visibility).`

B.1.6 Calcul des méta-généralisations entre méta-faits complets

L'étape suivante consiste à modifier l'ensemble méta-généralisations pour que celles-ci pointent sur les méta-faits complets. L'explication de cette étape ne présente aucun intérêt particulier et n'est donc pas détaillée ici.

La dernière étape de l'analyse du métamodèle consiste à générer dans un fichier texte les règles LP qui encodent ces méta-généralisations. Pour cela, il faut respecter la syntaxe particulière des systèmes logiques. La méthode ci-dessous renvoie une chaîne de caractères qui respecte cette syntaxe.

```
1  public String toCLPFormat()
2  {
3    //MetaFact leSpecial=new MetaFact(special);
4    String result = new String();
5
6    result = result + general.toString(":-")+special.name+"(Id";
7    for (Iterator itParam=special.parameters.iterator(); itParam.hasNext() ; )
8    {
9      Object param=itParam.next();
10     if (general.parameters.contains(param))
11     {
12       result = result+","+param;
13     }
14     else
15     {
16       result=result + ",_"+param;
17     }
18   }
19   result=result+").";
20   return result;
21 }
```

Par exemple, la méta-généralisation entre un état et un sommet graphique est encodé par la règle :

```
vertex(Id,Name,QualifiedName,Visibility) :-
   state(Id,_IsComposite,_IsLeaf,_IsOrthogonal,_IsSimple,
         _IsSubmachineState,Name,QualifiedName,Visibility).
```

Celle-ci exprime que l'on peut déduire la présence dans le modèle d'un sommet graphique (*Vertex* en anglais) à partir de la présence d'un état (*State* en anglais).

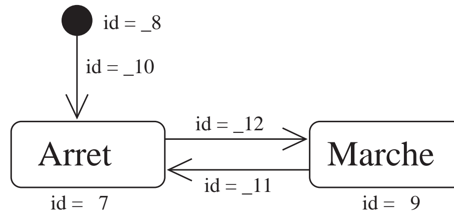
B.2 Analyse du modèle

L'encodage en (C)LP du modèle consiste à extraire des informations du modèle, et en fonction de ces informations à instancier les méta-faits en faits. Ces faits représenteront alors l'ensemble des éléments et des relations du modèle. Nous limitons notre présentation à l'instanciation des faits qui représentent les éléments du modèle. La génération des relations entre les éléments d'UML n'est pas illustrée.

La section B.2.1 présente le modèle utilisé afin d'illustrer la génération des faits. La section B.2.2 présente la méthode d'instanciation des méta-faits en faits. Enfin la section B.2.3 présente la méthode qui permet de générer les faits avec la syntaxe voulue.

B.2.1 Exemple de modèle

Afin d'expliquer le génération des faits, nous nous basons sur l'exemple du modèle déjà introduit en annexe A. Nous nous focalisons cependant sur la génération du seul fait qui représente l'état d'identifiant `_7` présenté à la ligne `(3)` de la figure B.4.




```
(...)
<nestedClassifier xmi:type="StateMachine" xmi:id="_5">................(1)
  <region xmi:type="Region" xmi:id="_6">..............................(2)
    <subvertex xmi:type="State" xmi:id="_7" name="Arret" isSimple=true>(3)
(...)
```

Fig. B.4 – Un modèle UML et son encodage en XMI

B.2.2 Instanciation des faits

La figure B.5 présente les classes impliquées dans la génération des faits. L'analyse du modèle conduit à la génération d'un ensemble de faits. Un fait est l'instance du méta-

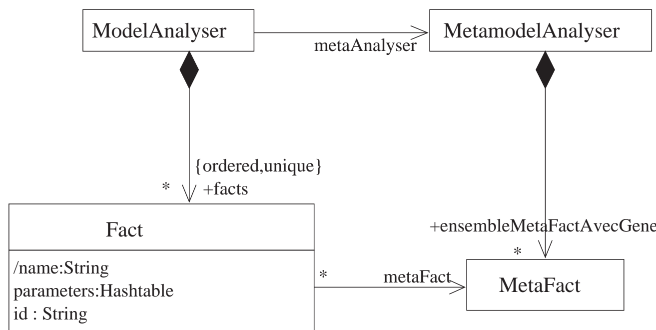


Fig. B.5 – Structure de données d'un fait

fait auquel il est associé par la fin d'association `metaFact`. Un fait contient un nom qui est le même que le nom de son méta-fait, c'est donc un attribut dérivé. Un fait a un identifiant qui correspond à l'identifiant de l'élément dans le document XMI qui encode le modèle. Chaque identifiant est unique. Enfin un fait a un ensemble de paramètres. Ces paramètres sont représentés par une table de hash car à chaque paramètre du méta-fait nous ferons correspondre la valeur instanciée pour le fait en question. Par exemple, l'état de la figure B.4 est nommé `Arret` ce qui est mémorisé en insérant le couple de valeurs (”$Name$”,”$arret$”) dans le table de hash.

La méthode ci-dessous permet d'instancier un méta-fait en fait. Elle prend en paramètre le méta-fait à instancier ainsi que l'élément d'UML encodé en XML.

```
1  //1-Fonction qui permet d'instancier un meta fait
2  public Fact instancierMetaFait(Element element, MetaFact metaFact){
3
4    //2- Construite un fait du meme nom que le meta fait
5    Fact faitResultat = new Fact(metaFact.name);
6    faitResultat.setMetaFact(metaFact);
7
8    //2- Pour chaque parametre du meta fait essayer de l'instancier
9    for (Iterator itParam = metaFact.parameters.iterator();itParam.hasNext();){
10     String parametre = (String) itParam.next();
11     //3-Regarder si la valeur du parametre est specifie dans le modele
12     String parametreMin = new
13     String(parametre.substring(0,1).toLowerCase().concat(parametre.substring(1)));
14     Attribute valeurAttribute = element.attribute(parametreMin);
15
16     if (valeurAttribute!=null){
17       //4-Cas ou la valeur est specifiee dans le modele
18       String valeurAttributMinusucule = new String(valeurAttribute.getValue());
19       valeurAttributMinusucule =
20       valeurAttributMinusucule.substring(0,1).toLowerCase().
21                                 concat(valeurAttributMinusucule.substring(1));
22       //4-Ajout du parametre dans la table de hash
23       faitResultat.parameters.put(parametre,valeurAttributMinusucule);
24     }
25     else{
26       //4-Sinon le parametre est ajoute mais il prend la valeur defaut
27       faitResultat.parameters.put(parametre,"default");
28     }
29   }
30   return faitResultat;
31 }
```

Notons que lorsque la valeur d'un attribut n'est pas spécifiée dans le modèle, on lui attribut de manière arbitraire la valeur `default`.

L'application de cette méthode sur l'état `Arret` du modèle de la figure B.4 crée l'objet représenté à la figure B.6.

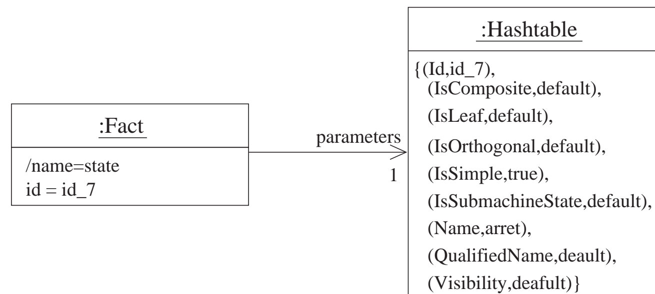


Fig. B.6 – Objets qui sauvegardent les informations du fait correspondant à l'état `Arret`

Notons que tous les éléments du modèle ont un identificateur unique dans le fichiers XMI. Pour obtenir une liste qui contient l'ensemble des éléments du modèle, nous pouvons donc utiliser un requête XPATH qui va sélectionner les éléments qui ont un identificateur. Ceci est réalisé par la ligne de code :

```
1  List ensembleElementAInstancier = racineDuModele.selectNodes("//*[@xmi:id]");
```

B.2.3 Génération des faits avec la syntaxe voulue

De la même manière que pour les règles, les faits doivent ensuite être générés dans un format compatible avec les outils de programmation logique. De plus, ce format doit suivre celui définit par le méta-fait, par exemple, la position des paramètres doit respecter la position des paramètres du méta-fait. L'obtention du fait au format voulu est réalisé par la méthode de la classe `Fact` dont voici le listing :

```
1  public String toCLPFormat()
2  {
3    String result = new String(name + "("+id);
4
5    for (Iterator itParam = metaFact.parameters.iterator();itParam.hasNext();){
6      String param = (String) itParam.next();
7      result +=","+(String)parameters.get(param);
8    }
9    result += ").";
10   return result;
11 }
```

Le fait généré qui va représenter la présence de l'état `Arret` dans le modèle est obtenu en passant en paramètre de cette méthode l'objet représenté à la figure B.6. Le résultat produit est :
`state(id_7,default,default,default,true,default,arret,default,default).`
Ce fait suit bien la syntaxe définie par le méta-fait qui représente la méta-classe `State` :
`state(Id,IsComposite,IsLeaf,IsOrthogonal,IsSimple,IsSubmachineState,`
`Name,QualifiedName,Visibility).`

Annexe C

Code du checker

C.1 Sémantique des constructions d'une machine à états

Le code LP ci-dessous est l'expression de la sémantique opérationnelle des constructions des machines à états prises en compte par le prototype.

```
:-import member/2 from basics.
:-import append/3 from basics.

%TRANSITION À PARTIR D'UN PSEUDO ÉTAT
stateMachineTrans(0,ObjectConf1,ObjectConf2,[Object|TransitionPath]):-
        member([Object,_Classifier,_SFC,ConfigStateMachine1,
                _InputEvents|_ObjectConfTail],ObjectConf1),
        isActive(SourceStatePath,ConfigStateMachine1),
        last(SourceStatePath,State),
        isPseudostate(State),
        getKind(State,StateKind),
        %Semantics depending the kind of the pseudo state
        %branch is present to support raphsody models (equivalent to choice)
        (StateKind=initial;StateKind=choice;StateKind=branch),
        sourcePath(TransitionPath,SourceStatePath),
        stateMachineTrans(ObjectConf1,ObjectConf2,[Object|TransitionPath]).

%TRANSITION SI UN EVENEMENT SYNCHRONE À ÉTÉ ENVOYÉ ON PRIVILÉGIE CET OBJET
stateMachineTrans(1,ObjectConf1,ObjectConf2,[Object|TransitionPath]):-
        not(stateMachineTrans(0,ObjectConf1,_,_)),
        member([Object,_Classifier,_SFC,_ConfigStateMachine1,
                InputEvents|_ObjectConfTail],ObjectConf1),
        member([_OpName,1],InputEvents),
        stateMachineTrans(ObjectConf1,ObjectConf2,[Object|TransitionPath]).

%TRANSITION À PARTIR D'UN ÉTAT CLASSIQUE
stateMachineTrans(2,ObjectConf1,ObjectConf2,[Object|TransitionPath]):-
        not(stateMachineTrans(0,ObjectConf1,_,_)),
        not(stateMachineTrans(1,ObjectConf1,_,_)),
        %Object\=environnement,
        stateMachineTrans(ObjectConf1,ObjectConf2,
                        [Object|TransitionPath]).


stateMachineTrans(ObjectConf1,ObjectConf2,[Object|TransitionPath]):-
        % TransitionPath contains the identifiers of the differents composites states,
        % and the transition id.
        member([Object,Classifier,SFC,ConfigStateMachine1,
                InputEvents|ObjectConfTail],ObjectConf1),
        isActive(SourceStatePath,ConfigStateMachine1),
        sourcePath(TransitionPath,SourceStatePath),
        targetPath(TransitionPath,TargetStatePath),
```

```
        guardIsOK(TransitionPath,ObjectConf1,Object),
        deactivate(SourceStatePath,ConfigStateMachine1,
                   ConfigStateMachineInter),
        activate(TargetStatePath,ConfigStateMachineInter,
                 ConfigStateMachine2),
                 %ConfigStateMachine2),
        replaceFirst([Object,Classifier,SFC,ConfigStateMachine1,
                      InputEvents|ObjectConfTail],
                     [Object,Classifier,SFC,ConfigStateMachine2,
                      InputEvents|ObjectConfTail],
                     ObjectConf1,ObjectConfInter1_1),
        executeExitActions(SourceStatePath,Object,ObjectConfInter1_1,
                           ObjectConfInter1),
        consumeEvent(TransitionPath,Object,ObjectConfInter1,ObjectConfInter2),
        executeActions(TransitionPath,Object,ObjectConfInter2,
                       ObjectConfInter3),
        executeEntryActions(TargetStatePath,Object,ObjectConfInter3,ObjectConf2).

%Activate a State depending on the type of the state in a configuration
activate([PathHead|PathTail],StateMachineConf1,StateMachineConf2):-
        PathTail\=[],
        member([PathHead,SelectedSubConfTmp],StateMachineConf1),
        (SelectedSubConfTmp=0 ->
            activateStateMachineOrCompositeStateWithoutDefault(PathHead,SelectedSubConf),
            replaceFirst([PathHead,0],[PathHead,SelectedSubConf],
                         StateMachineConf1,StateMachineConfInter)
        ;
            SelectedSubConf=SelectedSubConfTmp),
        activate(PathTail,SelectedSubConf,SubConf2),
        replaceFirst([PathHead,SelectedSubConf],[PathHead,SubConf2],
                     StateMachineConfInter,StateMachineConf2).

activate([StateId],Conf1,Conf2):-
        activateWithDefault(StateId,StateConf),
        replaceFirst([StateId,_],[StateId,StateConf],Conf1,Conf2).

%Gives the activated configuration of a state depending on its type
activateWithDefault(StateId,1):-
        (state(StateId,_IsComposite,_IsLeaf,_IsOrthogonal,true,
              _IsSubmachineState,_Name,_QualifiedName,_Visibility);
         getKind(StateId,branch)).

activateWithDefault(SM_ComposteState,StateConf):-
        (isStateMachine(SM_ComposteState);
         getIsComposite(SM_ComposteState,true)),
        activateStateMachineOrCompositeStateWithDefault(SM_ComposteState,StateConf).

activateStateMachineOrCompositeStateWithDefault(CompositeState,StateConf):-
        % create the intial state list contained by the region of the
        % state machine
        findall(InitialStateConf,(region(CompositeState,Region),
                                  subvertex(Region,InitialState),
                                  pseudostate(InitialState,initial,
                                              _Name,_QualifiedName,
                                              _Visibility),
                                  InitialStateConf=[InitialState,1]),
                InitialStateList),
        findall(OtherStateConf,(region(CompositeState,Region),
                                subvertex(Region,State),
                                not(pseudostate(State,initial,
                                                _Name,_QualifiedName,
                                                _Visibility)),
                                OtherStateConf=[State,0]),
                OtherStateConfList),
        append(InitialStateList,OtherStateConfList,StateConf).

activateStateMachineOrCompositeStateWithoutDefault(CompositeState,
                                                   StateConfList):-
        findall(StateConf,(region(CompositeState,Region),
                           subvertex(Region,State),
```

```
                                StateConf=[State ,0]) ,
                StateConfList ).

%Deactivate a State depending on the type of the state
deactivate([PathHead|PathTail], StateMachineConf1, StateMachineConf2):-
        PathTail\=[],
        member([PathHead, SelectedSubConf], StateMachineConf1),
        deactivate(PathTail, SelectedSubConf, SubConf2Tmp),
        (containJustPassiveState(SubConf2Tmp)->
                SubConf2=0
        ;
                SubConf2=SubConf2Tmp),
        replaceFirst([PathHead, SelectedSubConf], [PathHead, SubConf2],
                StateMachineConf1, StateMachineConf2).

deactivate([StateId], _, [StateId, 0]).

containJustPassiveState(Conf):-
        not((member(SubConf, Conf), SubConf=[_Id, ConfId], ConfId\=0)).

%isActive(StatePath, StateMachineConf).
isActive([PathHead|PathTail], StateMachineConf):-
        member([PathHead, SelectedSubConf], StateMachineConf),
        isActive(PathTail, SelectedSubConf).

isActive([StateId], StateMachineConf):-
        member([StateId, StateConf], StateMachineConf),
        StateConf\=0.

%source(TransitionPath, SourceStatePath).
sourcePath([StatePathHead|TransitionPathTail],
           [StatePathHead|StatePathTail]):-
        sourcePath(TransitionPathTail, StatePathTail).

sourcePath([TransitionId], [StateId]):-
        source(TransitionId, StateId).

targetPath(TransitionPath, TargetStatePath):-
        last(TransitionPath, Transition),
        target(Transition, State),
        statePathInStateMachine(State, TargetStatePath).


%guardAreOK check if the guards of a transition are all true.
%If the transition has no guard, it is OK.
guardIsOK(TransitionPath, Configuration, Object):-
        %Get the id of the transition
        ((last(TransitionPath, Transition),
        %Get the body of the guard
        guard(Transition, Guard),
        specification(Guard, GuardSpecification))->
            (getBody(GuardSpecification, Body),
            %The guard is true if and only if it can be executed.
            actionAExecuter(Body, Configuration, _, Object))
        ;
            true).

%Execution of actions of a transition
executeActions(TransitionPath, Object, Configuration1,
            Configuration2):-
        ((last(TransitionPath, Transition), isTransition(Transition),
        effect(Transition, OpaqueBehavior),
        getBody(OpaqueBehavior, Body))->
            actionAExecuter(Body, Configuration1, Configuration2, Object)
        ;
            Configuration1=Configuration2).

%Execution of entry actions of a state
executeEntryActions(TargetStatePath, Object, Configuration1, Configuration2):-
        ((last(TargetStatePath, State), isState(State),
```

```
        entry(State,OpaqueBehavior),
        getBody(OpaqueBehavior,Body))->
            actionAExecuter(Body,Configuration1,Configuration2,Object)
        ;
            Configuration1=Configuration2).

%Execution of entry actions of a state
executeExitActions(TargetStatePath,Object,Configuration1,Configuration2):-
        ((last(TargetStatePath,State),isState(State),
        exit(State,OpaqueBehavior),
        getBody(OpaqueBehavior,Body))->
            actionAExecuter(Body,Configuration1,Configuration2,Object)
        ;
            Configuration1=Configuration2).

/**************************************************/
%Consume an event included in the stack on the object
consumeEvent(TransitionPath,Object,ObjectConfiguration1,ObjectConfiguration2):-
        ((last(TransitionPath,Transition),
        trigger(Transition,SignalEvent))->
            ((occurrence(Signal,SignalEvent),
              signal(Reception,Signal),
              isReception(Reception))->
                acceptEventAction(SignalEvent,ObjectConfiguration1,
                                  ObjectConfiguration2,Object)
            ;
                acceptCallAction(SignalEvent,ObjectConfiguration1,
                                 ObjectConfiguration2,Object))
        ;
            ObjectConfiguration1=ObjectConfiguration2).

/**************************************************/
% Compute the path of a state (or Vertex) in a stateMachine
statePathInStateMachine(State,Path):-
        isVertex(State),
        subvertex(Region,State),
        region(Container,Region),
        statePathInStateMachine(Container,Path2),
        append(Path2,[State],Path).

statePathInStateMachine(StateMachine,[StateMachine]):-
        isStateMachine(StateMachine).
```

C.2 Sémantique des actions

Le code LP ci-dessous est l'expression de la sémantique opérationnelle des différentes actions prises en compte par notre outil.

```
:-import member/2 from basics.
:-import comma_to_list/2 from basics.
:-import eval/2 from eval.
:-import append/3 from basics.
/************************************************************/
/*CREATE AN OBJECT*/
/*If the name 'Name' of the object is not an atom, an objectId is
 autaomatically generated*/
createObjectAction(Classifier,Name,Objectconf1,Objectconf2,_):-
        isClass(Classifier),
        % Create the structural feature configuration
        %  of the object
        findall(Property,ownedAttribute(Classifier,Property),
                AtributeList),
        createOwnedSFC(AtributeList,OwnedSFC),
        % Create the classifier behavioral feature configuration
        createOwnedCBFC(Classifier,OwnedClassifierBFC),
        %Manage the object name
```

```
        (atom(Name)->
            IdObject=Name
        ;
            generateObjectId(IdObject)
            ),
        %The new configuration contains the Object Conf.
        Objectconf2=[[IdObject, Classifier ,OwnedSFC,OwnedClassifierBFC,
                    []]
                  |Objectconf1].

%Create structural feature configurations
createOwnedSFC([],[]).
createOwnedSFC([PropertyId|PropertyTail],
               [[PropertyId,TheDefaultValue]|PropertyConfigurationTail]):-
        getDefault(PropertyId,Default),
        ((Default=default)->
            TheDefaultValue=[]
        ;
            TheDefaultValue=[Default]),
        createOwnedSFC(PropertyTail,PropertyConfigurationTail).

% Create a classifier behavioral feature configuration when
% it is a state machine.
createOwnedCBFC(Classifier, InitialConf):-
        (context(CBehavior, Classifier)->
            isStateMachine(CBehavior),
            activateStateMachineOrCompositeStateWithDefault(CBehavior,
                                                            SMConf),
            InitialConf=[[CBehavior,SMConf]]
        ;
            InitialConf=[]).


/*********************************************************************/
%addStructuralFeatureValueAction([ContainingObject, Property, Value,
%InsertAt, IsReplaceall], ConfObjet1, ConfObjet2)

addStructuralFeatureValueAction([ContainingObject, Property, Value,
                                 InsertAt, IsReplaceall],
                                Conf1,Conf2,_):-
        (addStructuralFeatureValueAction_aux([ContainingObject, Property, Value,
                                              InsertAt, IsReplaceall],
                                             Conf1,Conf2)->
            createLinkAction([[ContainingObject, Property, Value, InsertAt,
                               IsReplaceall]], Conf1,Conf2,_);
            nl,write('ERROR␣:␣L␣attribut␣ID␣ou␣NAME='),
            write(Property),write('␣ne␣fait␣pas␣partie␣de␣l␣objet␣ID='),
            writeln(ContainingObject),fail)
            .

addStructuralFeatureValueAction_aux([ContainingObject, Property, Value,
                                     InsertAt, IsReplaceall],
                                    Conf1,Conf2):-
        createLinkAction([[ContainingObject, Property, Value, InsertAt,
                           IsReplaceall]], Conf1,Conf2,_).

/*******************************************************************/
/*CREATION D'UN LIEN*/
%linkEndCreationData(Property1, Value, InsertAt, IsReplaceAll)->
% [ContainingObject, Property, Value, InsertAt, IsReplaceall]
%There is a containing object that is not present in the UML
% specification but which is necessary.
createLinkAction([[ContainingObject, PropertyIdOrName, Value, InsertAt,
                IsReplaceall]|ResteLinkEnd],
            ConfObject1, Result,_):-
        %Get the structural feature of the object (OwnedSFC)
        member([ContainingObject,_,OwnedSFC|ResteConfObject], ConfObject1),
        %Get the value of the property (Value1)
        (getName(Property, PropertyIdOrName); Property=PropertyIdOrName),
        member([Property, Value1],OwnedSFC),
```

```
        %Compute the new value
        %3 case : the set of value is ordered (InsertAt is specified),
        % replaceAll=true, others
        ((integer(InsertAt))->
            addIth(InsertAt,Value,Value1,Value2)
        ;
            ((IsReplaceall=true)->
                Value2=[Value]
            ;
                Value2=[Value|Value1])),
        %Computation of the new configuration of the Property (replace
        % Value1 by Value2)
        replaceFirst([Property,Value1],[Property,Value2],OwnedSFC,
                     OwnedSFC2),
        %%Computation of the new configuration of the Object (replace
        %% the old config "OwnedSFC" of the Property by the new one
        %% "OwnedSFC2")
        replaceFirst([ContainingObject,Classifier,OwnedSFC|ResteConfObject],
                     [ContainingObject,Classifier,OwnedSFC2|ResteConfObject],
                     ConfObject1,ConfObject2),
        createLinkAction(ResteLinkEnd,ConfObject2,Result,_).

createLinkAction([],Conf,Conf,_).

/*********************************************************************/
%readLinkAction([L'Objet contenant, l'attribut à lire],
%                LaConfGlobaleDuSystème,LaValeurDeL'attribut)

%linkEndData(ContainingObject,Property)
readLinkAction([ContainingObject,Property],Value,ObjectConfs,_,_):-
        %write('ID de la property : '),write(ContainingObject),
        (readLinkAction_aux([ContainingObject,Property],Value,ObjectConfs,_,_)->
            true;
            nl,write('ERROR2: L attribut ID='),
            write(Property),write(' ne fait pas partie de l objet ID='),
            writeln(ContainingObject),fail).

readLinkAction_aux([ContainingObject,PropertyNameOrID],ObjectConfs,ValueFinal):-
        (isProperty(PropertyNameOrID)->
            Property=PropertyNameOrID
        ;
            getName(Property,PropertyNameOrID)),
        %Récupération de l'objet et de ses caractéristiques structurelles
        member([ContainingObject,_,OwnedSFC|_],ObjectConfs),
        %Récupération de la property de l'objet
        member([Property,Value],OwnedSFC),
        (getUpper(Property,1)->
            %write('CAS1 : je renvoie la valeur : '),
            Value=[ValueFinal]%,writeln(ValueFinal)
        ;
            %write('coucou1'),
            ValueFinal=Value).

/*********************************************************************/
acceptEventAction(Trigger,ObjectConfiguration1,
                  ObjectConfiguration2,Object):-
        %Get the name of the signal
        signal(Trigger,Signal),
        getName(Signal,SignalName),
        %Consume the signal
        member([Object,_Classifier,_SFC,_ConfigStateMachine1,
                InputEvents|ObjectConfTail],ObjectConfiguration1),
        removeFirst([SignalName,0],InputEvents,InputEvents2),
        %write(acceptEventAction_Object),write(Object),writeln(SignalName),
        replaceFirst([Object,_Classifier,_SFC,_ConfigStateMachine1,
                      InputEvents|ObjectConfTail],
                     [Object,_Classifier,_SFC,_ConfigStateMachine1,
                      InputEvents2|ObjectConfTail],
                     ObjectConfiguration1,ObjectConfiguration2).
```

```
acceptCallAction(Trigger,ObjectConfiguration1,
                 ObjectConfiguration2,Object):-
        %Get the name of the signal
        signal(Trigger,Signal),
        getName(Signal,SignalName),
        %write(acceptCallAction_Object),write(Object),writeln(SignalName),
        %Consume the signal
        member([Object,_Classifier,_SFC,_ConfigStateMachine1,
                InputEvents|ObjectConfTail],ObjectConfiguration1),
        removeFirst([SignalName,1],InputEvents,InputEvents2),
        replaceFirst([Object,_Classifier,_SFC,_ConfigStateMachine1,
                      InputEvents|ObjectConfTail],
                     [Object,_Classifier,_SFC,_ConfigStateMachine1,
                      InputEvents2|ObjectConfTail],
                     ObjectConfiguration1,ObjectConfiguration2).

% The sendMessageAction can be a callOperationAction or a
% sendSignalAction depending on the nature of the signal
sendMessageAction(Message,TargetObject,ObjectConfiguration1,
                  ObjectConfiguration2,_):-
        (getName(IdMessage,Message);IdMessage=Message),
        (isOperation(IdMessage)->
            callOperationAction(Message,TargetObject,ObjectConfiguration1,
                    ObjectConfiguration2,_)
        ;
            sendSignalAction(Message,TargetObject,ObjectConfiguration1,
                    ObjectConfiguration2,_)).

%The management of the execution thread can be done here.
callOperationAction(Operation,TargetObject,ObjectConfiguration1,
                    ObjectConfiguration2,_):-
        %Get the name of the operation
        (getName(Operation,OperationName);OperationName=Operation),
        %write('sendSignalAction Vers'),write(TargetObject),writeln(OperationName),
        %Add the trigger in the stack of the target object
        member([TargetObject,Classifier,SFC,ConfigStateMachine1,
                InputEvents|ObjectConfTail],ObjectConfiguration1),
        replaceFirst([TargetObject,Classifier,SFC,ConfigStateMachine1,
                      InputEvents|ObjectConfTail],
                     [TargetObject,Classifier,SFC,ConfigStateMachine1,
                     [[OperationName,1]|InputEvents]|ObjectConfTail],
                     ObjectConfiguration1,ObjectConfiguration2).


sendSignalAction(Operation,TargetObject,ObjectConfiguration1,
                 ObjectConfiguration2,_):-
        %Get the name of the operation
        (getName(Operation,OperationName);OperationName=Operation),
        %write('sendSignalAction Vers'),write(TargetObject),writeln(OperationName),
        %Add the trigger in the stack of the target object
        member([TargetObject,Classifier,SFC,ConfigStateMachine1,
                InputEvents|ObjectConfTail],ObjectConfiguration1),
        replaceFirst([TargetObject,Classifier,SFC,ConfigStateMachine1,
                      InputEvents|ObjectConfTail],
                     [TargetObject,Classifier,SFC,ConfigStateMachine1,
                     [[OperationName,0]|InputEvents]|ObjectConfTail],
                     ObjectConfiguration1,ObjectConfiguration2).


readSelfAction(0,_,_,0).

/******************************************************************/
% branch(Cond,Behavior1,Behavior2,C1,C2,Object)
% If Cond can be respected then Behavior1 is evaluated else Behavior1 is evaluated
branch(Cond,Behavior1,Behavior2,C1,C2,Object):-
        (actionAExecuter(Cond,C1,C2,Object)->
                comma_to_list(Behavior1,BehaviorInList)
        ;
                comma_to_list(Behavior2,BehaviorInList)),
```

```
        executerLesActions(BehaviorInList ,C1,C2,Object).

or(Behavior1 ,Behavior2 ,C1,C2,Object):-
        (comma_to_list(Behavior1 ,BehaviorInList);
        comma_to_list(Behavior2 ,BehaviorInList)),
        executerLesActions(BehaviorInList ,C1,C2,Object).

/*******************************************************************/
%Execute Actions contained in Body
actionAExecuter(Body,C1,C2,Object):-
        comma_to_list(Body,BodyEnListe),
        executerLesActions(BodyEnListe,C1,C2,Object).

executerLesActions([],C2,C2,_).

executerLesActions([Action|ResteAction],C1,C2,Object):-
        executerUneAction(Action,C1,CInter,Object),
        executerLesActions(ResteAction,CInter,C2,Object).

executerUneAction(Action,C1,C2,Object):-
        functor(Action,Functor,_),
        ((Functor= ==;
         Functor= \==;
         Functor= @=>;
         Functor= @=<;
         Functor= @>;
         Functor= @<;
         Functor= @=;
         Functor= =;
         Functor= \=)->
            once(Action)
        ;
            Action =.. [PredAction|Arg],
            append(Arg,[C1,C2,Object],ArgAvecConf),
            Action2 =.. [PredAction|ArgAvecConf],
            call(Action2)).
```

C.3 Expression des propriétés

Le code LP ci-dessous est l'expression des différentes incohérences comportementales. Ces incohérences utilisent le prédicat qui détermine le graphe des configurations atteignables.

```
:-table reachable/1.
:-table next/2.
:-import length/2 from basics.
:-import member/2 from basics.

reachable(InitialConf):-
        initialConfiguration(InitialConf).

reachable(C2):-
        reachable(C1),
        stateMachineTrans(_,C1,C2,_TransitionPath)
        ,not(integrityConstraintsAreNotSatisfied(C2)).

deadlock(Configuration):-
        reachable(Configuration),
        not((stateMachineTrans(_,Configuration,C2,_),reachable(C2))).

numberOfReachableConf(N):-
        cputime(T1),
        findall(Conf,
                reachable(Conf),
                ConfList),
        cputime(T2),
```

```
        T3 is T2-T1,
        write(temps_de_calcul___),
        writeln(T3),
        length(ConfList,N).

nonFirableTransition(T):-
        findall(TOK, firableTransition(TOK),
                TOKList),
        isTransition(T),
        not(member(T,TOKList)).

firableTransition(T):-
        once(firableTransition_aux(T)).

firableTransition_aux(T):-
        reachable(C),
        stateMachineTrans(_,C,_,Info),
        last(Info,T).

reachableState(S):-
        isVertex(S),
        reachable(C),
        member([_Object,_Classifier,_SFC,ConfigStateMachine,
                _InputEvents|_ObjectConfTail],C),
        isActive(SourceStatePath,ConfigStateMachine),
        last(SourceStatePath,S).

reachableState(S,C):-
        isVertex(S),
        reachable(C),
        member([_Object,_Classifier,_SFC,ConfigStateMachine,
                _InputEvents|_ObjectConfTail],C),
        isActive(SourceStatePath,ConfigStateMachine),
        last(SourceStatePath,S).
nonReachableState(S):-
        isVertex(S),
        not(reachableState(S)).

integrityConstraintsAreNotSatisfied(C):-
        member([_Object,_Classifier,_SFC,_ConfigStateMachine1,
                InputEvents|_ObjectConfTail],C),
        length(InputEvents,N),
        N@>5.
```

C.4 Diagnostic d'incohérence comportementale

Le code LP ci-dessous permet de construire la trace qui mène de l'état initial à la configuration redoutée. Il permet également de fournir certaines informations sur cette trace (se référer à l'annexe E pour plus de détails).

```
:-import length/2 from basics.
:-import member/2 from basics.
:-import append/3 from basics.
:-import storage_insert_keypair_bt/4,
   storage_delete_keypair_bt/3 from storage.
:-import storage_find_keypair/3 from storage.

show_trace(C,Trace,Length):-
        cputime(T1),
        initialConfiguration(ConfigInit),
        storage_delete_keypair_bt(diag, the_trace,_),
        storage_insert_keypair_bt(diag, the_trace,[ConfigInit],_),
        show_trace_aux(ConfigInit,C,Trace),!,
        cputime(T2),
        T3 is T2-T1,
        length(Trace,Length),
```

```
        write(temps_de_calcul___),
        writeln(T3).

show_trace_aux(C2,C2,T):-
        storage_find_keypair(diag, the_trace,T).

show_trace_aux(SourceConf,TargetConf,Trace):-
        stateMachineTrans(_,SourceConf,InterConf,_),
        reachable(InterConf),
        storage_find_keypair(diag, the_trace,T),
        ((member(InterConf,T))->
            fail
        ;
            storage_delete_keypair_bt(diag,the_trace,_),
            append(T,[InterConf],NewTrace),
            storage_insert_keypair_bt(diag,the_trace,NewTrace,_),
            show_trace_aux(InterConf,TargetConf,Trace)).

/*******************************************************************/
/* Write the trace and information about this trace on the file "traceFile"*/
show_transition_and_conf(C):-
        show_trace(C,T,_Length),
        open(file(traceFile),write,File),
        write(File,'TRACE_DE_C='),
        objectsInStates(C,SS),
        writeln(File,SS),
        writeln(File,''),
        writeln(File,C),
        close(File),
        show_transition_and_conf_aux(T).

show_transition_and_conf_aux([_]).

show_transition_and_conf_aux([C1|[C2|Trace]]):-
        stateMachineTrans(_,C1,C2,T),
        last(T,Transition),
        infoTransition(Transition,SourceStateInfo,TargetStateInfo),
        open(file(traceFile),append,File),
        writeln(File,''),
        writeln(File,'NEW_TRANSITION'),
        write(File,Transition),write(File,'objet_:_'),
        T=[O|_],
        write(File,O),writeln(File,'_:_'),
        write(File,SourceStateInfo),write(File,'_->_'),
        writeln(File,TargetStateInfo),
        write(File,'CONFIGURATION1_=_'),
        objectsInStates(C1,SS1),
        writeln(File,SS1),
        write(File,'CONFIGURATION2_=_'),
        objectsInStates(C2,SS2),
        writeln(File,SS2),
        close(File),
        show_transition_and_conf_aux([C2|Trace]).
```

C.5 Prédicats annexes

Le code LP ci-dessous est un ensemble de prédicats annexes utiles dans la définition des prédicats précédents notamment pour l'implantation de la sémantique et du diagnostic.

```
:-import member/2 from basics.

%replaceFirste(A,B,List1,List2). List2 is List1 where the first
% occurence of A has been replaced by B.
replaceFirst(A,B,[A|ListeTail],[B|ListeTail]).
replaceFirst(A,B,[Elem|ListeTail1],[Elem|ListeTail2]):-
```

```
        A\=Elem,
        replaceFirst(A,B,ListeTail1,
                     ListeTail2).

/*****************************************************/
%Get the last element of a list last(List,Element).
last([Element|[]],Element).
last([_AnElement|ListTail],Element):-
        last(ListTail,Element).



/*****************************************************/
% objectsInStates gives the name of the active states of the objects in
% a particular global configuration.
objectsInStates(Conf,ActiveStates):-
             findall([Object,ActiveStatesPath],
                confToState(Conf,Object,ActiveStatesPath),
                ActiveStates).

confToState(Conf,Object,StateListPlusEvent):-
        member([Object,_Class,_SFC,StateMachineConfiguration,
                InputEvents|_ResteConfObject],Conf),
        findall(NamePath,
                (isDirectlyActive(StatePath,StateMachineConfiguration),
                 idToName(StatePath,NamePath)),
                StateList),
        StateListPlusEvent=[StateList,InputEvents].


%removeFirst(Elem,List1,List2)
removeFirst(Elem,[Elem|List1Tail],List1Tail).
removeFirst(Elem,[Elem2|List1Tail],[Elem2|List2Tail]):-
        Elem\=Elem2,
        removeFirst(Elem,List1Tail,List2Tail).

%idToName permit to replace a list of ids by a string of name.

idToName([],[]).

idToName([Id|IdList],[Name|NameList]):-
        getName(Id,Name),!,
        idToName(IdList,NameList),!.

%%Gives Information about a transition
infoTransition(T,SourceStateInfo,TargetStateInfo):-
        isTransition(T),
        source(T,SS),statePathInStateMachine(SS,SourceStatePath),
        idToName(SourceStatePath,SourceStateInfo),
        target(T,TS),statePathInStateMachine(TS,TargetStatePath),
        idToName(TargetStatePath,TargetStateInfo),!.

stateInfo(S,Info):-
        not(is_list(S)),
        statePathInStateMachine(S,StatePath),
        idToName(StatePath,Info),!.

stateInfo(S,Info):-
        is_list(S),
        idToName(S,Info),!.

activeStatesInConf(ListState,Conf):-
        findall(SourceStateName,
                (member([_Object,_Classifier,_SFC,ConfigStateMachine,
                         _InputEvents|_ObjectConfTail],Conf),
                 isDirectlyActive(SourceStatePath,ConfigStateMachine),
                 idToName(SourceStatePath,SourceStateName)),
                ListState).

%isActive(StatePath,StateMachineConf).
```

```
isDirectlyActive([PathHead|PathTail],StateMachineConf):-
        member([PathHead,SelectedSubConf],StateMachineConf),
        isDirectlyActive(PathTail,SelectedSubConf).

isDirectlyActive([StateId],StateMachineConf):-
        member([StateId,StateConf],StateMachineConf),
        StateConf=1.

/*****************************************************/
% Gives information about Configurations

descriptionConf(Conf,Object):-
        member([Object,_Classifier,_SFC,_ConfigStateMachine1,
                _InputEvents|_ObjectConfTail],Conf),
        write(Object),writeln(':'),
        write('Machine A Etats : '),
        confToState(Conf,Object,ActiveStatesPath),
        writeln(ActiveStatesPath),
        write('Attributs :  '),
        confToAttribute(Conf,Object,AttributeList),
        writeln(AttributeList).

confToAttribute(Conf,Object,AttributeList):-
        findall([AttributeName,Value],
                (member([Object,_Classifier,SFC,_ConfigStateMachine1,
                _InputEvents|_ObjectConfTail],Conf),
                member([AttributeId,Value],SFC),
                once(getName(AttributeId,AttributeName))),
                AttributeList).

%confToStateTest(Conf,O,State,StateList):-
confToStateTest(Conf,O,StateMachineConfiguration,StateList):-
        member([O,_,_,StateMachineConfiguration|_],Conf),
        findall(NamePath,
               (isDirectlyActive(StatePath,StateMachineConfiguration),
                idToName(StatePath,NamePath)),
                StateList).

activeState(S,Conf):-
        member([_Object,_Classifier,_SFC,ConfigStateMachine,
                _InputEvents|_ObjectConfTail],Conf),
        isActive(StatePath,ConfigStateMachine),
        last(StatePath,S).

attValueInConf(Att,Conf,Value):-
        member([_Object,_Classifier,SFC,_ConfigStateMachine,
                        _InputEvents|_ObjectConfTail],Conf),
        getName(I,Att),
        member([I,ValueTmp],SFC),
        (ValueTmp=[SingleV]->
                SingleV=Value
        ;
                ValueTmp=Value).

transitionPath(T,Path):-
        source(T,S),
        statePathInStateMachine(S,StatePath),
        replaceFirst(S,T,StatePath,Path).
```

Annexe D

Représentation en LP du modèle des philosophes

Cette annexe fournit le code LP qui représente la structure du modèle des philosophes présenté en section 5.2.3 par la figure 5.12. Notons que la représentation du métamodèle n'est pas exposée.

```
1  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2  %%%% FAITS REPRESENTANT LES ELEMENTS DU MODELE %%%%
3  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
4  association(idRpy_13,default,default,default,default,default,default,public).
5  association(idRpy_8,default,default,default,default,default,default,public).
6  class(idRpy_47,default,default,default,topLevel,default,default,public).
7  class(idRpy_12,default,true,default,philosopher,default,default,public).
8  class(idRpy_3,default,default,default,fork,default,default,public).
9  component(idRpy_67,default,default,default,default,defaultComponent,default,default,
10           public).
11 constraint(idRpy_34,default,default,default,default).
12 constraint(idRpy_30,default,default,default,default).
13 dataType(idRpy_5,default,default,default,default,default,public).
14 expression(idRpy_7,true,default,default,default,default,default).
15 opaqueBehavior(idRpy_36,
16                (readSelfAction(Self),
17                 readStructuralFeatureAction([Self,right],RightFork),
18                 addStructuralFeatureValueAction([RightFork,isFree,false,_,true])),
19                default,default,default,default,default,
20                (readSelfAction(Self),
21                 readStructuralFeatureAction([Self,right],RightFork),
22                 addStructuralFeatureValueAction([RightFork,isFree,false,_,true])),
23                default,default,public).
24 opaqueBehavior(idRpy_32,
25                (readSelfAction(Self),
26                 readStructuralFeatureAction([Self,left],LeftFork),
27                 addStructuralFeatureValueAction([LeftFork,isFree,false,_,true]),
28                 default,default,default,default,default,
29                 (readSelfAction(Self),
30                  readStructuralFeatureAction([Self,left],LeftFork),
31                  addStructuralFeatureValueAction([LeftFork,isFree,false,_,true])),
32                 default,default,public).
33 opaqueBehavior(idRpy_28,
34                ((readSelfAction(Self),
35                  readStructuralFeatureAction([Self,right],RightFork),
36                  addStructuralFeatureValueAction([RightFork,isFree,true,_,true]),
37                  readStructuralFeatureAction([Self,left],LeftFork),
38                  addStructuralFeatureValueAction([LeftFork,isFree,true,_,true]))),
39                default,default,default,default,default,
40                ((readSelfAction(Self),
41                  readStructuralFeatureAction([Self,right],RightFork),
42                  addStructuralFeatureValueAction([RightFork,isFree,true,_,true]),
43                  readStructuralFeatureAction([Self,left],LeftFork),
```

```
44                     addStructuralFeatureValueAction([LeftFork,isFree,true,_,true]))),
45                   default,default,public).
46  opaqueExpression(newId2,
47                     (readSelfAction(Self),
48                      readStructuralFeatureAction([Self,right],RightFork),
49                      readStructuralFeatureAction([RightFork,isFree],IsFree),
50                      IsFree=true
51                     ),umlCLPAnalyser,default,default,default).
52  opaqueExpression(newId1,( readSelfAction(Self),
53                            readStructuralFeatureAction([Self,left],LeftFork),
54                            readStructuralFeatureAction([LeftFork,isFree],IsFree),
55                            IsFree=true
56                           ),umlCLPAnalyser,default,default,default).
57  package(idRpy_66,adaCodeGeneration,default,default,public).
58  package(idRpy_65,testingProfile_ADA,default,default,public).
59  package(idRpy_1,default,default,default,public).
60  property(idRpy_16,none,default,default,default,default,default,default,
61           default,default,default,1,default,default,1,public).
62  property(idRpy_14,none,default,default,default,default,default,default,
63           default,default,default,1,right,default,1,public).
64  property(idRpy_11,none,default,default,default,default,default,default,
65           default,default,default,1,default,default,1,public).
66  property(idRpy_9,none,default,default,default,default,default,default,
67           default,default,default,1,left,default,1,public).
68  property(idRpy_6,default,true,default,default,default,default,default,
69           default,default,default,1,isFree,default,1,public).
70  pseudostate(idRpy_27,initial,default,default,public).
71  region(idRpy_19,default,rOOT,default,public).
72  state(idRpy_26,false,default,false,true,false,eat,default,public).
73  state(idRpy_24,false,default,false,true,false,waitForRightFork,default,public).
74  state(idRpy_20,false,default,false,true,false,wait,default,public).
75  stateMachine(idRpy_18,default,default,default,default,
76               statechartOfPhilosopher,default,default,public).
77  transition(idRpy_25,default,default,1,default,public).
78  transition(idRpy_21,default,default,0,default,public).
79  transition(idRpy_23,default,default,3,default,public).
80  transition(idRpy_22,default,default,2,default,public).
81
82  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
83  %%%% FAITS REPRESENTANT LES RELATIONS DU MODELE %%%%
84  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
85  class(idRpy_11,idRpy_3).
86  class(idRpy_14,idRpy_12).
87  class(idRpy_16,idRpy_3).
88  class(idRpy_9,idRpy_12).
89  context(idRpy_18,idRpy_12).
90  effect(idRpy_21,idRpy_32).
91  effect(idRpy_22,idRpy_28).
92  effect(idRpy_25,idRpy_36).
93  feature(idRpy_3,idRpy_6).
94  guard(idRpy_21,idRpy_30).
95  guard(idRpy_25,idRpy_34).
96  incoming(idRpy_20,idRpy_22).
97  incoming(idRpy_20,idRpy_23).
98  incoming(idRpy_24,idRpy_21).
99  incoming(idRpy_26,idRpy_25).
100 memberEnd(idRpy_13,idRpy_14).
101 memberEnd(idRpy_13,idRpy_16).
102 memberEnd(idRpy_8,idRpy_11).
103 memberEnd(idRpy_8,idRpy_9).
104 outgoing(idRpy_20,idRpy_21).
105 outgoing(idRpy_24,idRpy_25).
106 outgoing(idRpy_26,idRpy_22).
107 outgoing(idRpy_27,idRpy_23).
108 ownedAttribute(idRpy_12,idRpy_14).
109 ownedAttribute(idRpy_12,idRpy_9).
110 ownedAttribute(idRpy_3,idRpy_11).
111 ownedAttribute(idRpy_3,idRpy_16).
112 ownedAttribute(idRpy_3,idRpy_6).
113 ownedAttribute(idRpy_6,idRpy_3).
```

```
114 ownedElement(idRpy_1,idRpy_12).
115 ownedElement(idRpy_1,idRpy_13).
116 ownedElement(idRpy_1,idRpy_3).
117 ownedElement(idRpy_1,idRpy_47).
118 ownedElement(idRpy_1,idRpy_8).
119 ownedElement(idRpy_12,idRpy_18).
120 ownedElement(idRpy_3,idRpy_5).
121 region(idRpy_18,idRpy_19).
122 source(idRpy_21,idRpy_20).
123 source(idRpy_22,idRpy_26).
124 source(idRpy_23,idRpy_27).
125 source(idRpy_25,idRpy_24).
126 specification(idRpy_30,newId1).
127 specification(idRpy_34,newId2).
128 stateMachine(idRpy_19,idRpy_18).
129 subvertex(idRpy_19,idRpy_20).
130 subvertex(idRpy_19,idRpy_24).
131 subvertex(idRpy_19,idRpy_26).
132 subvertex(idRpy_19,idRpy_27).
133 target(idRpy_21,idRpy_24).
134 target(idRpy_22,idRpy_20).
135 target(idRpy_23,idRpy_20).
136 target(idRpy_25,idRpy_26).
137 type(idRpy_11,idRpy_12).
138 type(idRpy_14,idRpy_3).
139 type(idRpy_16,idRpy_12).
140 type(idRpy_6,idRpy_5).
141 type(idRpy_9,idRpy_3).
```

Annexe E

Exemple de trace

Cette annexe présente un exemple de diagnostic pour la détection d'une incohérence comportementale. L'incohérence détectée est un deadlock sur le problème des philosophes. Le nombre de philosophes et de fourchettes est de 3. Les objets qui représentent les philosophes sont appelés `p1`, `p2` et `p3`. Les objets qui représentent les fourchettes sont appelés `f1`, `f2` et `f3`.

Les informations fournies sont présentées par le listing ci-dessous. Dans un premier temps, la configuration pour laquelle nous voulons des informations sur la manière dont elle a été atteinte (lignes 2 à 10) est fournie. Puis pour chaque changement de configuration de la trace nous fournissons :

- la transition responsable du changement de configuration (ligne 12 par exemple) ;
- l'objet qui contient cette transition (ligne 12 par exemple) ;
- le changement d'état pour l'objet (ligne 13 par exemple) ;
- les différents états actifs pour la configuration avant le changement de configuration (ligne 15 à 20 par exemple) ;
- les différents états actifs pour la configuration après le changement de configuration (ligne 23 à 29 par exemple).

Remarquons que nous préférons donner le nom des états actifs plutôt que leur identifiant pour plus de lisibilité. Il est cependant clair que ce type d'information n'est pas destinée à être exploité par l'utilisateur directement mais par un outil plus convivial qui les met en forme.

Notons que la dernière configuration est bien une situation de deadlock car les trois philosophes sont dans l'état `waitForRightFork` (lignes 162 à 164).

```
1   TRACE DE C =
2   [[f3,idRpy_3,[[idRpy_11,[]],[idRpy_16,[]],[idRpy_6,[false]]],[],[]],
3    [f2,idRpy_3,[[idRpy_11,[]],[idRpy_16,[]],[idRpy_6,[false]]],[],[]],
4    [f1,idRpy_3,[[idRpy_11,[]],[idRpy_16,[]],[idRpy_6,[false]]],[],[]],
5    [p3,idRpy_12,[[idRpy_14,[f2]],[idRpy_9,[f3]]],
6       [[idRpy_18,[[idRpy_20,0],[idRpy_24,1],[idRpy_26,0],[idRpy_27,0]]]],[]],
7    [p2,idRpy_12,[[idRpy_14,[f1]],[idRpy_9,[f2]]],
8       [[idRpy_18,[[idRpy_20,0],[idRpy_24,1],[idRpy_26,0],[idRpy_27,0]]]],[]],
9    [p1,idRpy_12,[[idRpy_14,[f3]],[idRpy_9,[f1]]],
10      [[idRpy_18,[[idRpy_20,0],[idRpy_24,1],[idRpy_26,0],[idRpy_27,0]]]],[]]]
11
12  NEW TRANSITION : idRpy_23   , objet : p3 :
13  [statechartOfPhilosopher,default] -> [statechartOfPhilosopher,wait]
14  CONFIGURATION1 =
15              [[f3,[[],[]]],
16               [f2,[[],[]]],
```

```
17              [f1 ,[[] ,[]]] ,
18              [p3 ,[[[ statechartOfPhilosopher , default ]] ,[]]] ,
19              [p2 ,[[[ statechartOfPhilosopher , default ]] ,[]]] ,
20              [p1 ,[[[ statechartOfPhilosopher , default ]] ,[]]]]
21
22  CONFIGURATION2 =
23             [[ f3 ,[[] ,[]]] ,
24              [f2 ,[[] ,[]]] ,
25              [f1 ,[[] ,[]]] ,
26              [p3 ,[[[ statechartOfPhilosopher , wait ]] ,[]]] ,
27              [p2 ,[[[ statechartOfPhilosopher , default ]] ,[]]] ,
28              [p1 ,[[[ statechartOfPhilosopher , default ]] ,[]]]]
29
30  NEW TRANSITION : idRpy_23   , objet : p2 :
31  [ statechartOfPhilosopher , default ] -> [ statechartOfPhilosopher , wait ]
32  CONFIGURATION1 =
33            [[ f3 ,[[] ,[]]] ,
34             [f2 ,[[] ,[]]] ,
35             [f1 ,[[] ,[]]] ,
36             [p3 ,[[[ statechartOfPhilosopher , wait ]] ,[]]] ,
37             [p2 ,[[[ statechartOfPhilosopher , default ]] ,[]]] ,
38             [p1 ,[[[ statechartOfPhilosopher , default ]] ,[]]]]
39
40  CONFIGURATION2 =
41            [[ f3 ,[[] ,[]]] ,
42             [f2 ,[[] ,[]]] ,
43             [f1 ,[[] ,[]]] ,
44             [p3 ,[[[ statechartOfPhilosopher , wait ]] ,[]]] ,
45             [p2 ,[[[ statechartOfPhilosopher , wait ]] ,[]]] ,
46             [p1 ,[[[ statechartOfPhilosopher , default ]] ,[]]]]
47
48  NEW TRANSITION : idRpy_23   , objet : p1 :
49  [ statechartOfPhilosopher , default ] -> [ statechartOfPhilosopher , wait ]
50  CONFIGURATION1 = [[ f3 ,[[] ,[]]] ,
51             [f2 ,[[] ,[]]] ,
52             [f1 ,[[] ,[]]] ,
53             [p3 ,[[[ statechartOfPhilosopher , wait ]] ,[]]] ,
54             [p2 ,[[[ statechartOfPhilosopher , wait ]] ,[]]] ,
55             [p1 ,[[[ statechartOfPhilosopher , default ]] ,[]]]]
56  CONFIGURATION2  =
57            [[ f3 ,[[] ,[]]] ,
58             [f2 ,[[] ,[]]] ,
59             [f1 ,[[] ,[]]] ,
60             [p3 ,[[[ statechartOfPhilosopher , wait ]] ,[]]] ,
61             [p2 ,[[[ statechartOfPhilosopher , wait ]] ,[]]] ,
62             [p1 ,[[[ statechartOfPhilosopher , wait ]] ,[]]]]
63
64  NEW TRANSITION : idRpy_21   , objet : p3 :
65  [ statechartOfPhilosopher , wait ] -> [ statechartOfPhilosopher , waitForRightFork ]
66  CONFIGURATION1  =
67            [[ f3 ,[[] ,[]]] ,
68             [f2 ,[[] ,[]]] ,
69             [f1 ,[[] ,[]]] ,
70             [p3 ,[[[ statechartOfPhilosopher , wait ]] ,[]]] ,
71             [p2 ,[[[ statechartOfPhilosopher , wait ]] ,[]]] ,
72             [p1 ,[[[ statechartOfPhilosopher , wait ]] ,[]]]]
73  CONFIGURATION2  =
74            [[ f3 ,[[] ,[]]] ,
75             [f2 ,[[] ,[]]] ,
76             [f1 ,[[] ,[]]] ,
77             [p3 ,[[[ statechartOfPhilosopher , waitForRightFork ]] ,[]]] ,
78             [p2 ,[[[ statechartOfPhilosopher , wait ]] ,[]]] ,
79             [p1 ,[[[ statechartOfPhilosopher , wait ]] ,[]]]]
80
81  NEW TRANSITION : idRpy_25   , objet : p3 :
82  [ statechartOfPhilosopher , waitForRightFork ] -> [ statechartOfPhilosopher , eat ]
83  CONFIGURATION1  =
84            [[ f3 ,[[] ,[]]] ,
85             [f2 ,[[] ,[]]] ,
86             [f1 ,[[] ,[]]] ,
```

```
87             [p3 , [[[ statechartOfPhilosopher , waitForRightFork ]] , []]] ,
88             [p2 , [[[ statechartOfPhilosopher , wait ]] , []]] ,
89             [p1 , [[[ statechartOfPhilosopher , wait ]] , []]]]
90  CONFIGURATION2 =
91            [[ f3 , [[] , []]] ,
92             [ f2 , [[] , []]] ,
93             [ f1 , [[] , []]] ,
94             [p3 , [[[ statechartOfPhilosopher , eat ]] , []]] ,
95             [p2 , [[[ statechartOfPhilosopher , wait ]] , []]] ,
96             [p1 , [[[ statechartOfPhilosopher , wait ]] , []]]]
97
98  NEW TRANSITION : idRpy_21 , objet : p1 :
99  [ statechartOfPhilosopher , wait ] -> [ statechartOfPhilosopher , waitForRightFork ]
100 CONFIGURATION1 =
101           [[ f3 , [[] , []]] ,
102            [ f2 , [[] , []]] ,
103            [ f1 , [[] , []]] ,
104            [p3 , [[[ statechartOfPhilosopher , eat ]] , []]] ,
105            [p2 , [[[ statechartOfPhilosopher , wait ]] , []]] ,
106            [p1 , [[[ statechartOfPhilosopher , wait ]] , []]]]
107 CONFIGURATION2 =
108           [[ f3 , [[] , []]] ,
109            [ f2 , [[] , []]] ,
110            [ f1 , [[] , []]] ,
111            [p3 , [[[ statechartOfPhilosopher , eat ]] , []]] ,
112            [p2 , [[[ statechartOfPhilosopher , wait ]] , []]] ,
113            [p1 , [[[ statechartOfPhilosopher , waitForRightFork ]] , []]]]
114
115 NEW TRANSITION : idRpy_22 , objet : p3 :
116 [ statechartOfPhilosopher , eat ] -> [ statechartOfPhilosopher , wait ]
117 CONFIGURATION1 =
118           [[ f3 , [[] , []]] ,
119            [ f2 , [[] , []]] ,
120            [ f1 , [[] , []]] ,
121            [p3 , [[[ statechartOfPhilosopher , eat ]] , []]] ,
122            [p2 , [[[ statechartOfPhilosopher , wait ]] , []]] ,
123            [p1 , [[[ statechartOfPhilosopher , waitForRightFork ]] , []]]]
124 CONFIGURATION2 =
125           [[ f3 , [[] , []]] ,
126            [ f2 , [[] , []]] ,
127            [ f1 , [[] , []]] ,
128            [p3 , [[[ statechartOfPhilosopher , wait ]] , []]] ,
129            [p2 , [[[ statechartOfPhilosopher , wait ]] , []]] ,
130            [p1 , [[[ statechartOfPhilosopher , waitForRightFork ]] , []]]]
131
132 NEW TRANSITION : idRpy_21 , objet : p3 :
133 [ statechartOfPhilosopher , wait ] -> [ statechartOfPhilosopher , waitForRightFork ]
134 CONFIGURATION1 =
135           [[ f3 , [[] , []]] ,
136            [ f2 , [[] , []]] ,
137            [ f1 , [[] , []]] ,
138            [p3 , [[[ statechartOfPhilosopher , wait ]] , []]] ,
139            [p2 , [[[ statechartOfPhilosopher , wait ]] , []]] ,
140            [p1 , [[[ statechartOfPhilosopher , waitForRightFork ]] , []]]]
141 CONFIGURATION2 =
142           [[ f3 , [[] , []]] ,
143            [ f2 , [[] , []]] ,
144            [ f1 , [[] , []]] ,
145            [p3 , [[[ statechartOfPhilosopher , waitForRightFork ]] , []]] ,
146            [p2 , [[[ statechartOfPhilosopher , wait ]] , []]] ,
147            [p1 , [[[ statechartOfPhilosopher , waitForRightFork ]] , []]]]
148
149 NEW TRANSITION : idRpy_21 , objet : p2 :
150 [ statechartOfPhilosopher , wait ] -> [ statechartOfPhilosopher , waitForRightFork ]
151 CONFIGURATION1 =
152           [[ f3 , [[] , []]] ,
153            [ f2 , [[] , []]] ,
154            [ f1 , [[] , []]] ,
155            [p3 , [[[ statechartOfPhilosopher , waitForRightFork ]] , []]] ,
156            [p2 , [[[ statechartOfPhilosopher , wait ]] , []]] ,
```

```
157              [p1 ,[[[ statechartOfPhilosopher ,waitForRightFork ]] ,[]]]]
158 CONFIGURATION2  =
159            [[ f3 ,[[] ,[]]] ,
160              [ f2 ,[[] ,[]]] ,
161              [ f1 ,[[] ,[]]] ,
162              [p3 ,[[[ statechartOfPhilosopher ,waitForRightFork ]] ,[]]] ,
163              [p2 ,[[[ statechartOfPhilosopher ,waitForRightFork ]] ,[]]] ,
164              [p1 ,[[[ statechartOfPhilosopher ,waitForRightFork ]] ,[]]]]
```

Bibliographie

[1] ISO/IEC Guide 73. *Risk Management - Vocabulary - Guidelines for use in standards.* International Organization for Standardization, September 2002.

[2] Action Semantics Consortium. Action Semantics for the UML, 2001. URL : http ://www.kabira.com/as/home.html.

[3] L. Apvrille, J.-P. Courtiat, C. Lohr, and P. de Saqui-Sannes. TURTLE : A Real-Time UML Profile Supported by a Formal Validation Toolkit. *Transactions on Software Engineering, IEEE Computer Society*, pages 473–487, 2004.

[4] Brian Berenbach. The Evaluation of Large, Complex UML Analysis and Design Models. In *International Conference on Software Engineering (ICSE)*, volume 0, pages 232–241, Los Alamitos, CA, USA, 2004. IEEE Computer Society.

[5] Purandar Bhaduri and R. Venkatesh. Formal Consistency of Models in Multi-View Modelling. In Kuzniarz et al. [44], pages 149–159.

[6] Jean-Paul Bodeveix, Thierry Millan, Christian Percebois, Pierre Bazex, and Louis Féraud. *NEPTUNE : Method, Checking and Documentation Generation for UML Application* . NEPTUNE Consortium, 2003.

[7] Jean-Paul Bodeveix, Thierry Millan, Christian Percebois, Christophe Le Camus, Pierre Bazex, and Louis Feraud. Extending OCL for verifying UML models consistency. In Kuzniarz et al. [44], pages 75–90.

[8] Anita Walkowiak Bogumila Hnatkowska. Consistency Checking of USDP Models. In Bogumila Hnatkowska and Frylewicz [9], pages 59–70.

[9] Lech Tuzinkiewicz Bogumila Hnatkowska, Miroslaw Staro and Zbigniew Frylewicz, editors. *Third International Workshop, Consistency Problems in UML-based Software Development III Understanding and Usage of Dependency Relationships*, Lisbon, Portugal, October 11, 2004.

[10] Marius Bozga, Jean-Claude Fernandez, Lucian Ghirvu, Susanne Graf, Jean-Pierre Krimm, and Laurent Mounier. IF : An Intermediate Representation and Validation Environment for Timed Asynchronous Systems. In J.M. Wing, J. Woodcock, and J. Davies, editors, *Proceedings of FM'99 (Toulouse, France)*, volume 1708 of *LNCS*, pages 307–327. Springer-Verlag, September 1999.

[11] Erwan Breton and Jean Bézivin. Towards an understanding of model executability. In *FOIS '01 : Proceedings of the international conference on Formal Ontology in Information Systems*, pages 70–80, New York, NY, USA, 2001. ACM Press.

[12] Franck Chauvel and Jean-Marc Jézéquel. Code Generation from UML Models with Semantic Variation Points. In Lionel C. Briand and Clay Williams, editors, *MoDELS*, volume 3713 of *Lecture Notes in Computer Science*, pages 54–68. Springer, 2005.

[13] LMC : **L**ogic Programming-Based **M**odel **C**hecking. URL : http ://www.cs.sunysb.edu/ lmc/.

[14] Weidong Chen, Michael Kifer, and David Scott Warren. HILOG : A foundation for higher-order logic programming. *Journal of Logic Programming*, 15(3) :187–230, 1993.

[15] Weidong Chen and David S. Warren. Tabled evaluation with delaying for general logic programs. *Journal of the ACM*, 43(1) :20–74, 1996.

[16] Yunja Choi and Christian Bunse. Behavioral Consistency Checking for Component-based Software Development Using the KobrA Approach. In Bogumila Hnatkowska and Frylewicz [9], pages 83–98.

[17] P. Cousot. Interprétation abstraite. *Technique et science informatique*, 19(1-2-3) :155–164, January 2000.

[18] Baoqiu Cui, Yifei Dong, Xiaoqun Du, K. Narayan Kumar, C. R. Ramakrishnan, I. V. Ramakrishnan, Abhik Roychoudhury, Scott A. Smolka, and David Scott Warren. Logic Programming and Model Checking. In Catuscia Palamidessi, Hugh Glaser, and Karl Meinke, editors, *PLILP/ALP*, volume 1490 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 1998.

[19] Giorgio Delzanno, Sandro Etalle, and Maurizio Gabbrielli, editors. *Specification, Analysis and Validation for Emerging Technologies in Computational Logic*, volume 94 of *Datalogiske Skrifter*, Roskilde, Denmark, July 27 2002.

[20] Giorgio Delzanno, Sandro Etalle, and Maurizio Gabbrielli. Special issue on specification analysis and verification of reactive systems. In *Theory and Practice of Logic Programming (TPLP).* Cambridge University Press, 2005.

[21] Giorgio Delzanno and Andreas Podelski. Model Checking in CLP. In *TACAS '99 : Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 223–239, London, UK, 1999. Springer-Verlag.

[22] dom4j. Disponible sur Internet. URL :http ://www.dom4j.org/.

[23] Hubert Dubois, Sébastien Gérard, and Chokri Mraidha. Un langage d'lction pour le développement UML de systèmes embarqués temps-réel. In *Ingénierie Dirigée par les Modèles , Paris, 30/06/05-01/07/05*, pages 125–140. CEA List - ISBN 2-7261-1284-6, juin 2005.

[24] The Eclipse Platform. Disponible sur Internet. URL :http ://www.eclipse.org/.

[25] R. Eshuis and R. Wieringua. Tool support for verifying UML activity diagrams. *IEEE Transactions on Software Engineering, Vol.3, n° 7, IEEE Computing Society* , 2004.

[26] Andy Evans, Robert France, Kevin Lano, and Bernhard Rumpe. Developing the UML as a formal modelling notation. In Jean Bézivin and Pierre-Alain Muller, editors, *The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop, Mulhouse, France, June 1998*, pages 297–307, 1998.

[27] F. Fages. *Constraint logic programming (in French).* Cours de l'Ecole Polytechnique. Ellipses, Paris, 1996.

[28] Yosee Feldman and Ehud Y. Shapiro. Temporal debugging and its visual animation. In *Internationel Symposium on Logic Programming (ISLP)*, pages 3–17, 1991.

[29] Stephan Flake and Wolfgang Müller. Specification of real-time properties for UML models. In *Hawai'i International Conference on System Sciences (HICSS-35)*. IEEE Computer Society, 2002.

[30] Thom Frühwirth and Slim Abdennadher. *Essentials of Constraint Programming*. Springer, 2003.

[31] Stéphanie Gaudan. Abstraction par Prédicat de Systèmes de Transitions Symboliques. Master's thesis, Université Bordeaux 1, Laboratoire LABRI, 2004.

[32] Jochen Malte Küster Gregor Engels, Reiko Heckel. Rule-Based Specification of Behavioral Consistency Based on the UML Meta-model. In *Lecture Notes in Computer Science, Volume 2185, Springer, 2001*, 2001.

[33] Clare Gryce, Anthony Finkelstein, and Christian Nentwich. Lightweight Checking for UML Based Software Development. In Kuzniarz et al. [44], pages 124–132.

[34] Allain Le Guennec. *Software Engineering and Formal Methods with UML Specification. Validation and Test Generation (in French)*. PhD thesis, Université De Rennes 1, 2001.

[35] D. Harel and B. Rumpe. Modeling languages : Syntax, semantics and all that stuff, 2000. David Harel, Bernhard Rumpe, Modeling Languages : Syntax, Semantics and All That Stuff, Technical paper number MCS00-16, The Weizmann Institute of Science. available on http ://www.cs.york.ac.uk/puml/.

[36] Thomas A. Henzinger and Rupak Majumdar. A classification of symbolic transition systems. In *STACS '00 : Proceedings of the 17th Annual Symposium on Theoretical Aspects of Computer Science*, pages 13–34, London, UK, 2000. Springer-Verlag.

[37] Bogumila Hnatkowska, Zbigniew Huzar, Ludwik Kuzniarz, and Lech Tuzinkiewicz. A systematic approach to consistency within UML based software development process. In Kuzniarz et al. [44], pages 16–29.

[38] Zbigniew Huzar, Ludwik Kuzniarz, Gianna Reggio, and Jean-Louis Sourrouille. Consistency problems in uml-based software development. In Nuno Jardim Nunes, Bran Selic, Alberto Rodrigues da Silva, and José Ambrosio Toval Álvarez, editors, *UML Satellite Activities*, volume 3297 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2004.

[39] Joxan Jaffar and Michael J. Maher. Constraint logic programming : A survey. *Journal of Logic Programming*, 19/20 :503–581, 1994.

[40] Joxan Jaffar, Andrew E. Santosa, and Razvan Voicu. A CLP Proof Method for Timed Automata. In *Real-Time Systems Symposium (RTSS 2004)*, pages 175–186. IEEE Computer Society, 2004.

[41] Soon-Kyeong Kim and David Carrington. A Formal Object-Oriented Approach to defining Consistency Constraints for UML Models. In *Software Engineering Conference. Australian*. IEEE Computing Society, 2004.

[42] Philippe Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6) :42–50, 1995.

[43] L. Kuzniarz, Z. Huzar, G. Reggio, J.-L. Sourrouille, and M. Staron, editors. *Proceedings of the IEEE Workshop on Consistency Problems in UML-Based Software Development II.* IEEE and Department of Software Engineering and Computer Science of Blekinge Institute of Technology, 2003.

[44] L. Kuzniarz, G. Reggio, J.-L. Sourrouille, and Z. Huzar, editors. *Proceedings of the Workshop on Consistency Problems in UML-based Software Development.* Department of Software Engineering and Computer Science of Blekinge Institute of Technology, 2002.

[45] Hervé Leblanc, Thierry Millan, and Ileana Ober. Démarche de développement orienté modèles : de la vérification de modèles à l'outillage de la démarche . In Sébastien Gérard, Jean-Marie Favre, Pierre-Alain Müller, and Xavier Blanc, editors, *Ingénierie Dirigée par les Modèles , Paris, 30/06/05-01/07/05*, pages 125–140. CEA List - ISBN 2-7261-1284-6, 2005.

[46] Michael Leuschel, Andreas Podelski, C.R. Ramakrishnan, and Ulrich Ultes-Nitsche, editors. *Proceedings of the ACM Sigplan Workshop on Verification and Computational Logic VCL'2001*, September 2001.

[47] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation.* Prentice-Hall, 1981.

[48] Zhiming Liu, He Jifeng, Xiaoshan Li, and Yifeng Chen. Consistency and Refinement of UML Models. In Bogumila Hnatkowska and Frylewicz [9], pages 23–39.

[49] H. Malgouyres and G. Motet. A UML model consistency verification approach based on meta-modeling formalization. In *SAC '06 : Proceedings of the 2006 ACM symposium on Applied computing*, pages 1804–1809, New York, NY, USA, 2006. ACM Press.

[50] H. Malgouyres, J.-P. Seuma Vidal, and G. Motet. UML 2.0 Consistency Rules, March 2005. URL : http ://www.lesia.insa-toulouse.fr/UML.

[51] R. Marcano and N. Levy. Using B formal specifications for analysis and verification of UML/OCL models. In Kuzniarz et al. [44], pages 91–105.

[52] Julio L. Medina, Michael Gonzalez Harbour, and Jose M. Drake. The « UML Profile for Schedulability, Performance and Time » in the Schedulability Analysis and Modeling of Real-Time Distributed Systems, 2004.

[53] Ulf Nilsson and Johan Lübcke. Constraint logic programming for local and symbolic model-checking. *Lecture Notes in Computer Science, Springer*, 1861 :384+, 2000.

[54] Bashar Nuseibeh, Steve Easterbrook, and Alessandra Russo. Leveraging inconsistency in software development. *IEEE Computer*, 33(4) :24–29, 2000.

[55] Iulian Ober, Susanne Graf, and Ileana Ober. Validation of UML models via a mapping to communicating extended timed automata. In *11th International SPIN Workshop on Model Checking of Software, 2004*, volume LNCS 2989, Springer, 2004.

[56] Object Management Group. Meta Object Facility (MOF), 2003. URL : http ://www.omg.org/technology/documents/formal/mof.htm.

[57] Object Management Group. UML 2.0 OCL Final Adopted specification, 2003. URL : <http ://www.omg.org/cgi-bin/doc ?ptc/2003-10-14>.

[58] Object Management Group. Unified Modeling Language 2.0 Superstructure Specification, August 2004. URL : http ://www.omg.org/uml/.

[59] Object Management Group. Unified Modeling Language 2.0 Superstructure Specification, August 2005. URL : http ://www.omg.org/uml/.

[60] Object Management Group. MOF 2.0/XMI Mapping Specification, v2.1, September 2005. URL : http ://www.omg.org/cgi-bin/doc ?formal/2005-09-01.

[61] Noël Plouzeau Olivier Defour, Jean-Marc Jézéquel. Extra-Functional Contract Support in Components. In *Lecture Notes in Computer Science, Volume 3054*, pages 217–232. Springer, May 2004.

[62] Giridhar Pemmasani, Haifeng Guo, Yifei Dong, C. R. Ramakrishnan, and I. V. Ramakrishnan. Online justification for tabled logic programs. In *International Functional and Logic Programming (FLOPS)*, volume 2998 of *Lecture Notes in Computer Science*, pages 24–38, Nara, Japan, April 2004. Springer.

[63] Andreas Podelski, C.R. Ramakrishnan, and Ulrich Ultes-Nitsche, editors. *Proceedings of the Workshop on Verification and Computational Logic VCL'2000*, July 2000.

[64] Claudia Pons. Heuristics on the definition of uml refinement patterns. In Jirí Wiedermann, Gerard Tel, Jaroslav Pokorný, Mária Bieliková, and Julius Stuller, editors, *SOFSEM*, volume 3831 of *Lecture Notes in Computer Science*, pages 461–470. Springer, 2006.

[65] Michaël Périn. *Spécifications graphiques multi-vues : formalisation et vérification de cohérence*. PhD thesis, Institut de Formation Supérieure en Informatique et Communication (IFSIC), October 2000.

[66] Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, Scott A. Smolka, Terrance Swift, and David Scott Warren. Efficient model checking using tabled resolution. In *CAV '97 : Proceedings of the 9th International Conference on Computer Aided Verification*, pages 143–154, London, UK, 1997. Springer-Verlag.

[67] Holger Rasch and Heike Wehrheim. Consistency between UML Classes and Associated State Machines. In Kuzniarz et al. [44], pages 46–60.

[68] Abhik Roychoudhury, C. R. Ramakrishnan, and I. V. Ramakrishnan. Justifying proofs using memo tables. In *ACM Conference on Principles and Practice of Declarative Programming*, pages 178–189, 2000.

[69] Konstantinos Sagonas, Terrance Swift, and David S. Warren. Xsb as an efficient deductive database engine. In *SIGMOD '94 : Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, pages 442–453, New York, NY, USA, 1994. ACM Press.

[70] Konstantinos Sagonas, Terrance Swift, David S. Warren, Juliana Freire, and Prasad Rao. The XSB System Version 2.7.1 Volume 1 : Programmer's Manual.

[71] Konstantinos Sagonas, Terrance Swift, David S. Warren, Juliana Freire, and Prasad Rao. The XSB System Version 2.7.1 Volume 2 : Libraries, Interfaces and Packages.

[72] Philippe Schnoebelen, Béatrice Bérard, Michel Bidoit, François Laroussinie, and Antoine Petit. *Vérification de logiciels : techniques et outils du model-checking*. Vuibert, April 1999.

[73] J.P. Seuma Vidal, H. Malgouyres, and G. Motet. UML 2.0 consistency rules identification. In *SERP'05 - The International Conference on Software Engineering Research and Practice*. CSREA Press, 2005.

[74] J.-L. Sourrouille and G. Caplat. Checking UML Model Consistency. In Kuzniarz et al. [44], pages 1–15.

[75] Jean Louis Sourrouille and Guy Caplat. A Pragmatic View on Consistency Checking of UML Models. In Kuzniarz et al. [43], pages 43–50.

[76] Michael Spivey. *An introduction to logic programming through Prolog*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[77] Chouki Tibermacine, Régis Fleurquin, and Salah Sadou. Préservation de choix architecturaux lors de l'évolution d'un composant. In *Proceedings of OCM-SI'04 workshop (Objets, Composants et Modèles pour les Systèmes d'Information), held in conjunction with INFORSID'04*, Biarritz, France, May 2004.

[78] R. L. Toro, J.-P. Seuma Vidal, H. Malgouyres, and G. Motet. UML Inconsistencies Assessment. In *Proceedings of the 3rd European Conference on Embedded Real-Time Software*, Toulouse, France, January 2006.

[79] R. L. Toro, J.-P. Seuma Vidal, L.-M. Motet, H. Malgouyres, T. Le Saux, and G. Motet. Estimation of UML Inconsistencies in Avionics Domain. In L. Kuzniarz, G. Reggio, J.-L. Sourrouille, and M. Staron, editors, *Workshop Comode*. Department of Software Engineering and Computer Science of Blekinge Institute of Technology, 2005.

[80] UML2 an EMF-based implementation of the UML 2.x metamodel for the Eclipse platform, 2006. URL : http ://www.eclipse.org/uml2/.

[81] D. Varro. A formal semantics of UML Statecharts by model transition systems. In *International Conference on Graph Transformation (ICGT)*. Springer-Verlag - Lecture Notes in Computer Science, 2002.

[82] XPath, the XML Path Language. Disponible sur Internet. URL :http ://www.w3.org/TR/xpath.

[83] XSB a tabled logic programming system. URL : http ://xsb.sourceforge.net/.

Index

action, 76

certification, 85
changement de configuration, 66
code mort, 85
cohérence, 5
    définition de la, 13
    vérification
        état de l'art, 30
        cadre générique, 27
configuration, 63, 66, 68
    atteignable, 84
    attribut, 69
    globale d'un modèle UML, 71
    machine à états, 69
    objet, 70

diagramme
    relations entre, 8

encodage de modèle, 28, 32, 43, 45

formalisation des règles de cohérence, 43

guides de modélisation, 15

historique d'UML, 2

incohérence, 5
    d'un modèle UML, 6
    diagnostic, 56
    formalisation des, 7
    gestion des, 6
instanciation, 5
interprétation abstraite, 84
introspection, 60

langage formel, 29–31
logique temporelle, 39

métalangage, 4
métamodélisation, 4
métamodèle, 4, 20
    respect du, 58
model-checking, 39
    symbolique, 96
MOF, 4, 50

OCL, 7, 32

point de variation sémantique, 82
programmation logique, 36
    avec contraintes, 34, 40
    fait, 36
    interprétation déclarative, 36
    interprétation procédurale, 37
    résolution SLD, 37
    résolution SLG, 38
    règle, 37
    tabulée, 38

règle de changement de configuration, 63, 77
règle de cohérence, 6, 13, 30
    comportementale, 34, 67
    formalisation, 55, 83
    identification, 17
    invariants de configuration, 84
    structurelle, 34
risque, 9
    étapes de gestion du, 9
    évaluation du, 9
    acceptation du, 10
    estimation du, 9
    gestion du, 9
    identification du, 9
    traitement du, 9, 10

sémantique
    de vérification, 8
    opérationnelle, 8, 34
sémantique opérationnelle, 77

trace, 63

transformation de modèle, 29, 30
transformation de modèles, 15

vérification comportementale, 34
vérification structurelle, 34

XMI, 28

zone ALARP, 10