Environnement d'assistance au développement de transformations de graphes correctes

Amani Makhlouf

▶ To cite this version:

Amani Makhlouf. Environnement d'assistance au développement de transformations de graphes correctes. Réseaux et télécommunications [cs.NI]. Institut National Polytechnique de Toulouse - INPT, 2019. Français. NNT : 2019INPT0027 . tel-04166309

HAL Id: tel-04166309

https://theses.hal.science/tel-04166309

Submitted on 19 Jul 2023

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

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

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par :

Institut National Polytechnique de Toulouse (Toulouse INP)

Discipline ou spécialité :

Informatique et Télécommunication

Présentée et soutenue par :

Mme AMANI MAKHLOUF

le vendredi 8 février 2019

Titre :

Environnement d'assistance au développement de transformations de graphes correctes

---

Ecole doctorale :

Mathématiques, Informatique, Télécommunications de Toulouse (MITT)

Unité de recherche :

Institut de Recherche en Informatique de Toulouse (I.R.I.T.)

Directeur(s) de Thèse :

M. CHRISTIAN PERCEBOIS
MME HANH NHI TRAN

Rapporteurs :

M. CHRISTIAN ATTIOGBE, UNIVERSITE DE NANTES
Mme NICOLE LEVY, CNAM PARIS

Membre(s) du jury :

Mme REGINE LALEAU, UNIVERSITE PARIS 12, Président
M. CHRISTIAN PERCEBOIS, UNIVERSITE TOULOUSE 3, Membre
Mme HANH NHI TRAN, UNIVERSITE TOULOUSE 3, Membre
M. YAMINE AIT AMEUR, INP TOULOUSE, Membre

Remerciement

Je tiens à remercier en premier lieu mon directeur de thèse Christian Percebois qui m'a suivi tout au long de ces années. J'avoue que j'ai eu la chance d'effectuer ma thèse sous sa direction. J'apprécie extrêmement sa disponibilité, ses qualités pédagogiques et scientifiques et sa sympathie. Je lui suis très reconnaissante, cette thèse lui doit beaucoup.

Je remercie également Hanh Nhi Tran d'avoir co-encadré ces travaux de thèse. Merci pour son soutien, son aide et surtout pour ses suggestions toujours avisées.

Mes plus vifs remerciements s'adressent aux membres du jury de thèse qui ont accepté de juger ces travaux. Merci à mes deux rapporteurs, Christian Attiogbé et Nicole Levy d'avoir accepté d'évaluer ma thèse. Je remercie également Yamine Ait Ameur et Régine Laleau d'avoir accepté d'être examinateurs de ces travaux.

J'adresse aussi mes remerciements à Martin Strecker pour les échanges nourris et pour sa disponibilité par rapport à mes interrogations.

Je souhaite remercier aussi les membres de l'équipe ARGOS, ainsi que les autres membres du laboratoire IRIT, notamment avec qui j'ai partagé de bons moments.

J'exprime ma reconnaissance à tous mes amis qui m'ont encouragée et soutenue le long de ce travail. Merci d'avoir toujours été là pour moi et de m'avoir tant aidée. Je suis tellement chanceuse de les avoir dans ma vie !

Ce travail n'aurait pu être mené à bien sans le soutien de ma famille, particulièrement ma mère : la clef de ma réussite. Merci pour sa patience. Elle m'a toujours rassurée et supportée dans les bons comme les mauvais moments. Sans elle je n'en serais pas là aujourd'hui !

Ces remerciements ne peuvent s'achever sans une pensée pour toi qui n'a pas vu l'aboutissement de ces dernières années, mais je sais que tu en aurais été très fier de ta fille.

Résumé

Les travaux de cette thèse ont pour cadre la vérification formelle, et plus spécifiquement le projet ANR Blanc CLIMT (*Categorical and Logical Methods in Model Transformation*) dédié aux grammaires de graphes. Ce projet, qui a démarré en février 2012 pour une durée de 48 mois, a donné lieu à la définition du langage Small-t$\mathcal{ALC}$, bâti sur la logique de description $\mathcal{ALCQI}$. Ce langage prend la forme d'un DSL (*Domain Specific Language*) impératif à base de règles, chacune dérivant structurellement un graphe. Le langage s'accompagne d'un composant de preuve basé sur la logique de Hoare chargé d'automatiser le processus de vérification d'une règle.

Cependant, force est de constater que tous les praticiens ne sont pas nécessairement familiers avec les méthodes formelles du génie logiciel et que les transformations sont complexes à écrire. En particulier, ne disposant que du seul prouveur, il s'agit pour le développeur Small-t$\mathcal{ALC}$ d'écrire un triplet de Hoare $\{P\}S\{Q\}$ et d'attendre le verdict de sa correction sous la forme d'un graphe contre-exemple en cas d'échec. Ce contre-exemple est parfois difficile à décrypter, et ne permet pas de localiser aisément l'erreur au sein du triplet. De plus, le prouveur ne valide qu'une seule règle à la fois, sans considérer l'ensemble des règles de transformation et leur ordonnancement d'exécution.

Ce constat nous a conduits à proposer un environnement d'assistance au développeur Small-t$\mathcal{ALC}$. Cette assistance vise à l'aider à rédiger ses triplets et à prouver ses transformations, en lui offrant plus de rétroaction que le prouveur. Pour ce faire, les instructions du langage ont été revisitées selon l'angle *ABox* et *TBox* de la logique $\mathcal{ALCQI}$. Ainsi, conformément aux logiques de description, la mise à jour du graphe par la règle s'assimile à la mise à jour *ABox* des individus (les nœuds) et de leurs relations (les arcs) d'un domaine terminologique *TBox* (le type des nœuds et les étiquettes des arcs) susceptible d'évoluer.

Les contributions de cette thèse concernent : (1) un extracteur de préconditions *ABox* à partir d'un code de transformation S et de sa postcondition Q pour l'écriture d'une règle $\{P\}S\{Q\}$ correcte par construction, (2) un raisonneur *TBox* capable d'inférer des propriétés sur des ensembles de nœuds transformés par un enchaînement de règles $\{P_i\}S_i\{Q_i\}$, et (3) d'autres diagnostics *ABox* et *TBox* sous la forme de tests afin d'identifier et de localiser des problèmes dans les programmes.

Table des matières

Table des figures **11**

Liste des tableaux **15**

Introduction **17**

1 Le langage Small-t$\mathcal{ALC}$: contexte et contributions **21**

1.1 Le langage Small-t$\mathcal{ALC}$ . . . 22

1.1.1 L'approche algébrique . . . 22

1.1.2 Vers une approche basée sur une logique . . . 26

1.1.3 Le langage Small-t$\mathcal{ALC}$ . . . 28

1.1.3.1 De la logique $\mathcal{ALCQI}$ aux formules Small-t$\mathcal{ALC}$ . . . 28

1.1.3.2 Les instructions du langage . . . 32

1.1.3.3 Programme Small-t$\mathcal{ALC}$ . . . 34

1.1.3.4 Graphe Small-t$\mathcal{ALC}$ . . . 37

1.1.4 Transformation d'un diagramme de classes UML . . . 37

1.2 L'assistance au développement Small-t$\mathcal{ALC}$ . . . 41

1.2.1 Problématique . . . 42

1.2.2 Objectif . . . 44

1.2.3 Environnement d'assistance au développement Small-t$\mathcal{ALC}$ . . . 46

1.2.3.1 Assistance à la construction des règles *ABox* . . . 46

1.2.3.2 Vérification *TBox* des règles Small-t$\mathcal{ALC}$ . . . 48

1.2.3.3 Diagnostics *ABox* et *TBox* . . . 50

1.2.3.3.1 Le générateur de cas de test *ABox* . . . 50

1.2.3.3.2 Le raisonneur *TBox* . . . 51

1.3 Conclusion . . . 53

2 Extraction des préconditions *ABox* des règles Small-t$\mathcal{ALC}$ 55
2.1 Vérification d'une règle Small-t$\mathcal{ALC}$ . . . 56
2.2 Analyse statique d'une règle Small-t$\mathcal{ALC}$ . . . 58
2.2.1 Approche générale . . . 58
2.2.2 Calcul d'alias . . . 60
2.2.2.1 Non-déterminisme du langage Small-t$\mathcal{ALC}$ . . . 60
2.2.2.2 Variables possiblement équivalentes . . . 62
2.2.2.3 Calcul des variables non possiblement équivalentes . . . 64
2.2.2.4 Exemple de calcul d'alias . . . 66
2.2.3 Transformateur de prédicats Small-t$\mathcal{ALC}$ . . . 68
2.2.3.1 Réécriture des *wp* en formules $\mathcal{ALCQI}$ . . . 70
2.2.3.2 Simplification des $wp_{\mathcal{ALCQI}}$ . . . 72
2.2.3.2.1 Les instructions *add* et *delete* . . . 72
2.2.3.2.2 L'instruction *select* . . . 74
2.2.3.2.3 Les autres instructions . . . 76
2.2.3.3 Spécificités du transformateur . . . 77
2.2.4 Exemple . . . 78
2.3 Mise au point d'un triplet d'une règle Small-t$\mathcal{ALC}$ . . . 79
2.4 Travaux similaires . . . 85
2.5 Conclusion . . . 88

3 Vérification *TBox* des règles Small-t$\mathcal{ALC}$ 89
3.1 Visions *ABox* et *TBox* d'un graphe Small-t$\mathcal{ALC}$ . . . 90
3.2 Inférence des propriétés globales Small-t$\mathcal{ALC}$ . . . 92
3.2.1 Interprétation abstraite et règle Small-t$\mathcal{ALC}$ . . . 93
3.2.2 Sémantique abstraite Small-t$\mathcal{ALC}$ . . . 97
3.2.3 Contrats de transformation . . . 98
3.2.4 Formules *TBox* . . . 99
3.3 Analyse statique par interprétation abstraite . . . 100
3.3.1 Processus de l'analyseur statique *TBox* . . . 101
3.3.2 Interprétation abstraite des instructions Small-t$\mathcal{ALC}$ . . . 102
3.3.3 Exemple . . . 109
3.4 Relation entre les niveaux *ABox* et *TBox* . . . 111
3.4.1 Dépendance *ABox* et *TBox* . . . 111
3.4.2 Complémentarité des deux niveaux de vérification . . . 112
3.4.2.1 Formules *TBox* vérifiées, règles *ABox* incorrectes . . . 113
3.4.2.2 Règles *ABox* prouvées correctes, formules *TBox* invalides . . . 114
3.5 Propriétés du second ordre . . . 115

3.6 Travaux similaires . . . 118
3.7 Conclusion . . . 120

4 Environnement de développement Small-t$\mathcal{ALC}$ 121

4.1 Assistance à l'identification d'erreurs . . . 122
4.1.1 Tests *ABox* . . . 122
4.1.1.1 Assertions de test et formules Small-t$\mathcal{ALC}$ . . . 122
4.1.1.2 Données de test et instructions Small-t$\mathcal{ALC}$ . . . 123
4.1.1.3 Génération de tests par analyse dynamique . . . 126
4.1.1.4 Génération de tests par analyse statique . . . 127
4.1.2 Diagnostics *TBox* . . . 129
4.2 Synoptique de l'environnement Small-t$\mathcal{ALC}$ . . . 131
4.3 Étude de cas : gestion des services d'un hôpital . . . 132
4.3.1 Écriture et correction de la règle *receivePatient* . . . 133
4.3.2 Nouvelle écriture de la règle *assignDoctor* . . . 135
4.3.3 Inférence d'une formule *TBox* après la règle *assignDoctor* . . . 136
4.3.4 Localisation des erreurs dans la règle *allocateRoom* . . . 137
4.3.5 Vérification dynamique des formules *TBox* . . . 139
4.4 Autres environnements de transformation de graphes . . . 143
4.4.1 Outils existants . . . 143
4.4.2 Comparatif avec Small-t$\mathcal{ALC}$ . . . 147
4.4.3 Synthèse . . . 152
4.5 Conclusion . . . 153

Conclusion et perspectives 155

Bibliographie 159

Table des figures

1.1 Un exemple d'une règle de production simple [33] 23
1.2 Deux exemples de règles de production plus complexes [33] 24
1.3 L'approche DPO 25
1.4 L'approche SPO 26
1.5 Conditions d'application négatives [107] 28
1.6 Un modèle de graphe de $x : C$ *and* $x : (\geq 2\ r\ C)$ *and* $y : C$ *and* $x\ !r\ y$ 31
1.7 Une suite d'instructions Small-t$\mathcal{ALC}$ 32
1.8 Mise à jour du graphe de la figure 1.6 après l'application d'une suite d'instructions . 32
1.9 Sémantique axiomatique des instructions Small-t$\mathcal{ALC}$ 33
1.10 Règle *update* 35
1.11 Sémantique axiomatique des appels de règles Small-t$\mathcal{ALC}$ 35
1.12 Point d'entrée du programme 36
1.13 Exemple d'une séquence d'instructions définissant un graphe 37
1.14 Méta-modèle simplifié du diagramme de classes 38
1.15 Instance d'un diagramme de classes 38
1.16 Méta-modèle du modèle relationnel 38
1.17 Instance d'un modèle relationnel 39
1.18 Programme Small-t$\mathcal{ALC}$ de transformation d'un diagramme de classes UML en un modèle relationnel 40
1.19 Graphes source et cible du programme Small-t$\mathcal{ALC}$ de la figure 1.18 41
1.20 Composant de preuve ABox 42
1.21 Triplet de Hoare incorrect 43
1.22 Contre-exemple du triplet de la figure 1.21 43
1.23 L'environnement Small-t$\mathcal{ALC}$ 45
1.24 Extracteur de préconditions *ABox* 47
1.25 Exemple d'un graphe source et de sa transformation 47
1.26 Code et postcondition de la règle *association2Table* 48

1.27 Moteur d'inférence de formules *TBox* 49
1.28 Point d'entrée du programme *uml2rdb* annoté par des formules *TBox* 49
1.29 Générateur de cas de test *ABox* 50
1.30 Génération des tests associés à la postcondition de la règle *class2Table* 51
1.31 Le raisonneur *TBox* 52
1.32 Point d'entrée du programme *uml2rdb* annoté par des formules *TBox* 52

2.1 Les plus faibles préconditions des instructions Small-t$\mathcal{ALC}$ 57
2.2 Les conditions de vérification des instructions Small-t$\mathcal{ALC}$ 57
2.3 $wp_{\mathcal{ALCQI}}\big(add(i\ r\ j), x : (\leq 3\ r\ C)\big)$ 59
2.4 $wp_{\mathcal{ALCQI}}(add(i\ r\ j), x : (\leq 3\ r\ C))$ simplifiée 59
2.5 Triplet incorrect lorsque x a pour alias y 61
2.6 Modèle de graphe satisfaisant la précondition $x : C\ and\ x\ r\ y$ 61
2.7 Triplet incorrect lorsque b a pour alias c 61
2.8 Modèle de graphe où b et c référencent le même nœud 61
2.9 Exemple de deux modèles satisfaisant la formule $x : C\ and\ y\ r\ z$ 62
2.10 Conditions garantissant que x et y sont non possiblement équivalentes 63
2.11 Algorithme du calcul d'alias à partir de la séquence d'instructions et la postcondition d'une règle 66
2.12 Calcul d'alias sur un exemple 67
2.13 Extraction des préconditions $\mathcal{ALCQI}$ en transformant la postcondition en forme normale disjonctive 69
2.14 Simplification de la plus faible précondition $\mathcal{ALCQI}$ en exploitant les variables non possiblement équivalentes 69
2.15 Exemples de codes composés d'une instruction *select* et d'une postcondition 76
2.16 Exemple d'un code de transformation et d'une postcondition 78
2.17 Première version de la règle *assignDoctor* 80
2.18 Deuxième version de la règle *assignDoctor* 83
2.19 Version finale de la règle *assignDoctor* 85
2.20 Démarche de dérivation de préconditions OCL [32] 86
2.21 Règle de restructuration *Pull up Method* [21] 87
2.22 Développement itératif d'une règle de restructuration [21] 88

3.1 Un graphe *ABox* illustrant un exemple dans le domaine de la zoologie 91
3.2 Vision *TBox* du graphe illustrant l'exemple dans le domaine de la zoologie 91
3.3 Graphe abstrait correspondant au graphe concret de la figure 3.1 92
3.4 Approximation A de l'ensemble R non-calculable des états atteignables [63] 93
3.5 Vision *ABox*/*TBox* de Small-t$\mathcal{ALC}$ 94

3.6 Deux programmes de transformation distincts vérifiant $A \subseteq (only\ r\ B)$ 95
3.7 Modèles de graphes sources des programmes $P1$ et $P2$ 96
3.8 Modèles de graphes cibles des programmes $P1$ et $P2$ 96
3.9 Le graphe *TBox* traduisant la propriété $A \subseteq (only\ r\ B)$ 96
3.10 Contrat *TBox* de la règle *delete_r* 98
3.11 Séquence d'appels de règles avec des contrats *TBox* 100
3.12 Une règle *add_r* et son appel dans le *main* 109
3.13 Évolution des graphes *ABox* et *TBox* 110
3.14 Évolution de la *TFormule* après chaque instruction de la règle *add_r* 110
3.15 Exemple de spécifications *ABox* affaiblies 112
3.16 Une règle *ABox* incorrecte 113
3.17 Contre-exemple du prouveur indiquant que *delete_r_nonB* est incorrecte 113
3.18 Affaiblissement des spécifications *ABox* 114
3.19 Programme Small-t$\mathcal{ALC}$ permettant de construire un graphe biparti 116
3.20 Une règle assurant le *TFait* $C \subseteq (some\ succ\ C)$ 117
3.21 Une occurrence interdite dans un graphe vérifiant le *TFait* $all = (\leq 1\ r\ all)$ 117
3.22 Programme Small-t$\mathcal{ALC}$ vérifiant qu'un graphe est acyclique 118

4.1 Test généré à partir de la formule $a : A\ and\ a : (\leq 2\ r\ A)\ and\ a\ !r\ a$ 123
4.2 Graphe typique de la formule $a : A\ and\ a : (<= 3\ r\ A)\ and\ b : A$ 124
4.3 Trois graphes associés à la formule $a : A\ and\ a : (<= 3\ r\ A)\ and\ b : A$ et homomorphes au graphe de la figure 4.2 125
4.4 Processus de génération des tests par analyse dynamique *ABox* 126
4.5 Triplet de Hoare incorrect 127
4.6 Test associé à la postcondition *ABox* de la règle *update* 127
4.7 Processus de génération des tests par analyse statique *ABox* 128
4.8 Exemple d'identification d'erreurs par analyse statique [1] 128
4.9 Processus de vérification d'une formule *TBox* 130
4.10 Point d'entrée annoté de formules *TBox* 130
4.11 Évolution du graphe après chaque règle 130
4.12 L'environnement expérimental Small-t$\mathcal{ALC}$ 131
4.13 Première version de la règle *receivePatient* 133
4.14 Contre-exemple de la règle *receivePatient* de la figure 4.13 133
4.15 Résultats du générateur de tests par analyse statique ABox 134
4.16 Version finale et correcte de la règle *receivePatient* 135
4.17 Version initiale de la règle *assignDoctor* 135
4.18 Version finale de la règle *assignDoctor* 136
4.19 Point d'entrée du programme *Hospital* 137

4.20 Première version de la règle *allocateRoom* . . . 138
4.21 Contre-exemple de la règle *allocateRoom* de la figure 4.20 . . . 138
4.22 Graphe cible et test associé à la postcondition de la règle *allocateRoom* . . . 139
4.23 Version finale de la règle *allocateRoom* . . . 139
4.24 Point d'entrée du programme *Hospital* . . . 140
4.25 Évolution du graphe après chaque règle . . . 142
4.26 Schéma de règle conditionnelle en GP [100] . . . 145
4.27 Programme *colouring* en GP [104] . . . 145
4.28 Erreur sémantique en PROGRES . . . 147
4.29 Deux règles AGG en conflit [96] . . . 150
4.30 Positionnement des outils existants par rapport à Small-t$\mathcal{ALC}$ . . . 153

Liste des tableaux

1.1 Variation de l'instanciation $x : C$ 44
1.2 Les objectifs et les outils Small-t$\mathcal{ALC}$ proposés 45
1.3 Les outils Small-t$\mathcal{ALC}$ proposés 53

2.1 Élimination des substitutions des *wp* de *add(i : C)* et *delete(i : C)* 70
2.2 Élimination des substitutions des *wp* de *add(i r j)* et *delete (i r j)* 71
2.3 Élimination du quantificateur de $wp(select\ v\ with\ F, Q)$ 72
2.4 Simplification de la plus faible précondition $\mathcal{ALCQI}$ de $add(i : C)$ 73
2.5 Simplification de la plus faible précondition $\mathcal{ALCQI}$ de $add(i\ r\ j)$ 73
2.6 Simplification de la plus faible précondition $\mathcal{ALCQI}$ de $delete(i : C)$ 74
2.7 Simplification de la plus faible précondition $\mathcal{ALCQI}$ de $delete(i\ r\ j)$ 74
2.8 Condition de vérification de l'instruction *select* 75
2.9 Condition de vérification de l'instruction *while* 76

3.1 Interprétation de l'instruction $add(i : C)$ 105
3.2 Interprétation de l'instruction $delete(i : C)$ 106
3.3 Interprétation de l'instruction $add(i\ r\ j)$ 107
3.4 Interprétation de l'instruction $delete(i\ r\ j)$ 108

4.1 Assertions associées aux faits Small-t$\mathcal{ALC}$ 123
4.2 Composants Small-t$\mathcal{ALC}$ développés durant cette thèse 132
4.3 Nombre de lignes de code des composants Small-t$\mathcal{ALC}$ 132
4.4 Comparaison entre Small-t$\mathcal{ALC}$ et les autres outils présentés 148

Introduction

Objets mathématiques, les graphes permettent de modéliser des structures de données complexes de manière expressive et intuitive. On les retrouve dans le développement des transformations en ingénierie dirigée par les modèles, la représentation des bases de données NoSQL, ou encore l'analyse de la qualité de service d'un réseau de stations de travail reconfiguré. Ils sont non seulement utilisés en informatique, mais aussi dans d'autres domaines comme la chimie, la biologie, la gestion, etc. D'aucuns estiment que les transformations de graphes constituent un formalisme flexible, adapté à la modélisation des systèmes dynamiques en décrivant leur évolution par l'application de règles. Toutefois, transformer un graphe demeure un point critique en ingénierie logicielle.

Le projet ANR Blanc CLIMT (*Categorical and Logical Methods in Model Transformation*) s'est intéressé à la réécriture de graphes pour transformer des modèles. Il vise trois objectifs : les fondements de la réécriture au travers de la définition de nouvelles classes de système intégrant de nouveaux traits de la réécriture comme le clonage de nœuds ou la manipulation d'attributs inductifs, les techniques de preuve avec le choix de nouvelles logiques pour exprimer la transformation et la prouver, et l'automatisation des preuves afin de garantir un niveau de confiance dans la correction des transformations. Le projet a démarré en février 2012 pour une durée de 48 mois ; il s'est notamment soldé par la définition d'un langage de réécriture de graphes, bâti sur la logique de description $\mathcal{ALCQI}$. Ce langage, appelé Small-t$\mathcal{ALC}$ et dédié à la dérivation structurelle de graphes, prend la forme d'un DSL (*Domain Specific Language*) impératif à base de règles.

La précondition P, les instructions S et la postcondition Q d'une règle de transformation Small-t$\mathcal{ALC}$ forment un triplet de Hoare $\{P\}S\{Q\}$. Conformément au troisième objectif du projet, le langage s'accompagne d'un composant de preuve chargé d'automatiser le processus de vérification d'une règle. Le prouveur détermine la plus faible précondition de la règle, à partir de ses instructions et de sa postcondition, puis s'assure que la précondition du triplet renforce, au sens de l'implication, la plus faible précondition. Si tel est le cas, les instructions de transformation respectent la spécification donnée par la pré- et la postcondition, et le triplet est correct. Dans le cas contraire, un graphe contre-exemple est fourni en tant que diagnostic d'échec de la preuve.

Cette thèse complète le composant de preuve CLIMT en offrant un environnement d'assistance au développeur lui permettant de concevoir des transformations correctes. Elle prolonge les objectifs

du projet, en établissant le constat que tous les praticiens ne sont pas nécessairement familiers avec les méthodes formelles du génie logiciel et que les transformations sont complexes à écrire. En particulier, ne disposant dans l'environnement CLIMT que du seul prouveur, il s'agit pour le développeur d'écrire un triplet complet et d'attendre le verdict de sa correction sous la forme d'un graphe contre-exemple en cas d'échec. Ce contre-exemple est parfois difficile à décrypter, et ne permet pas de localiser aisément l'erreur au sein du triplet de Hoare. De plus, le prouveur ne valide qu'une seule règle à la fois, sans considérer l'ensemble des règles de transformation et leur ordonnancement d'exécution. Dans ce contexte, il s'agit d'offrir plus de rétroaction au développeur afin de l'aider à rédiger et à prouver ses triplets.

Par rapport aux logiques de description, un nœud est codé par un individu *ABox* et un arc par une relation *ABox*. La mise à jour du graphe par la règle s'assimile dès lors à la mise à jour *ABox* des individus et de leurs relations d'un domaine terminologique *TBox*, représenté par le type des nœuds et les étiquettes des arcs, et susceptible d'évoluer. Cette vision ontologique du langage Small-t$\mathcal{ALC}$ ouvre de nouvelles perspectives par rapport à CLIMT pour la vérification d'un programme de transformation. Le travail de cette thèse s'est donc attaché à revisiter les instructions du langage selon l'angle *ABox* et *TBox* de la logique $\mathcal{ALCQI}$. Il a donné lieu au développement de deux outils d'analyse statique complétant le prouveur : (1) un extracteur de préconditions *ABox* à partir d'un code de transformation S et de sa postcondition Q pour l'écriture d'une règle $\{P\}S\{Q\}$ correcte par construction, (2) et un raisonneur *TBox* capable d'inférer des propriétés sur des ensembles de nœuds transformés par un enchaînement de règles $\{P_1\}S_1\{Q_1\}, \{P_2\}S_2\{Q_2\}, ..., \{P_n\}S_n\{Q_n\}$.

L'analyse statique du code de transformation d'une règle, combinée à un calcul d'alias des variables désignant les nœuds du graphe, permet d'extraire un ensemble de préconditions *ABox* validant la règle. Les inférences *TBox* pour un enchaînement de règles résultent d'une analyse statique par interprétation abstraite des règles *ABox* afin de vérifier formellement des états du graphe avant et après les appels des règles. A côté de ces outils formels, les développeurs Small-t$\mathcal{ALC}$ peuvent aussi identifier et localiser des problèmes dans leurs programmes grâce à des diagnostics d'exécution sous la forme de tests.

Le premier chapitre a pour vocation de présenter le contexte général, la problématique et l'objectif de cette thèse. Il introduit d'abord les deux approches les plus utilisées pour les transformations de graphes : l'approche algébrique, historiquement la plus ancienne et la plus répandue, et l'approche logique adoptée par CLIMT. Ce chapitre présente ensuite la logique $\mathcal{ALCQI}$ exploitée pour définir le langage de programmation impératif Small-t$\mathcal{ALC}$. La syntaxe d'un programme Small-t$\mathcal{ALC}$ consistant en plusieurs règles de transformation sous la forme de triplets Hoare $\{P\}S\{Q\}$, est également fournie. Le chapitre discute ensuite de la difficulté à construire de tels triplets et à analyser des règles erronées, ce qui nous conduit à proposer un environnement d'assistance Small-t$\mathcal{ALC}$ dont les fonctionnalités sont résumées dans ce même chapitre.

Le chapitre 2 fait l'objet de la première contribution de cette thèse. Développer une transformation avec une spécification correcte formée d'une précondition et d'une postcondition est souvent

difficile. Dans ce cadre, cette contribution vise à aider les développeurs à construire des triplets $\{P\}S\{Q\}$ corrects par construction : à partir d'un code de transformation S et d'une postcondition Q, une précondition est extraite par analyse statique de S à l'égard de Q en exploitant les plus faibles préconditions des instructions Small-t$\mathcal{ALC}$. L'analyse statique met en œuvre un calcul d'alias identifiant des variables du code pouvant désigner un même nœud du graphe. La formule résultante, sous forme normale disjonctive, permet au développeur de sélectionner une précondition exprimant au mieux son intention tout en garantissant un triplet correct.

Le chapitre 3 concerne la deuxième contribution. Une règle de transformation Small-t$\mathcal{ALC}$, exécutée au niveau *ABox*, affecte les concepts de la *TBox*. Ce chapitre étudie l'effet de l'enchaînement des règles sur la *TBox* en exploitant ses axiomes pour exprimer des propriétés sur les ensembles des sommets et arcs. Cela permet de vérifier le graphe à un niveau abstrait en inférant des propriétés globales *TBox* à partir des propriétés locales *ABox* explicites d'une règle. Des propriétés monadiques du second ordre sont ainsi exprimables sur le graphe transformé. Cette inférence est effectuée à travers une analyse statique par interprétation abstraite en se basant sur une approximation de la sémantique des instructions *ABox*, naturellement présente notre langage. Ce chapitre montre également la dépendance et la complémentarité entre les deux niveaux de vérification *ABox* et *TBox*.

Le dernier chapitre complète les contributions précédentes basées sur des techniques formelles permettant de vérifier statiquement les transformations Small-t$\mathcal{ALC}$ aux niveaux *ABox* et *TBox*. Il introduit ainsi deux autres outils semi-formels capables d'offrir des diagnostics riches à un développeur. Le premier est un générateur de tests *ABox* qui aide à identifier des erreurs dans le code d'une règle et sa spécification *ABox*. Il peut procéder par une analyse statique en générant une spécification à comparer à celle du développeur, ou par analyse dynamique pour confronter la postcondition à l'égard du code donné en exécutant la règle sur un graphe source arbitraire ou généré à partir de la précondition. Le deuxième outil est un raisonneur *TBox* qui vérifie dynamiquement des propriétés *TBox* données par le développeur sur un graphe concret ; il s'agit d'appliquer les règles appelées sur ce graphe, et de vérifier les propriétés *TBox* vis-à-vis des graphes mis à jour. Une étude de cas exploitant les composants de l'environnement Small-t$\mathcal{ALC}$ est aussi décrite dans ce chapitre.

Enfin, cette thèse s'achève par une conclusion sur le travail présenté et laisse entrevoir des perspectives à court, ainsi qu'à long terme.

Chapitre 1

Le langage Small-t$\mathcal{ALC}$: contexte et contributions

Sommaire

1.1 Le langage Small-t$\mathcal{ALC}$ 22
1.1.1 L'approche algébrique 22
1.1.2 Vers une approche basée sur une logique 26
1.1.3 Le langage Small-t$\mathcal{ALC}$ 28
1.1.3.1 De la logique $\mathcal{ALCQI}$ aux formules Small-t$\mathcal{ALC}$ 28
1.1.3.2 Les instructions du langage 32
1.1.3.3 Programme Small-t$\mathcal{ALC}$ 34
1.1.3.4 Graphe Small-t$\mathcal{ALC}$ 37
1.1.4 Transformation d'un diagramme de classes UML 37
1.2 L'assistance au développement Small-t$\mathcal{ALC}$ 41
1.2.1 Problématique 42
1.2.2 Objectif 44
1.2.3 Environnement d'assistance au développement Small-t$\mathcal{ALC}$ 46
1.2.3.1 Assistance à la construction des règles *ABox* 46
1.2.3.2 Vérification *TBox* des règles Small-t$\mathcal{ALC}$ 48
1.2.3.3 Diagnostics *ABox* et *TBox* 50
1.2.3.3.1 Le générateur de cas de test *ABox* 50
1.2.3.3.2 Le raisonneur *TBox* 51
1.3 Conclusion 53

Dans la plupart des systèmes de réécriture de graphes, l'utilisateur contrôle l'application d'une règle par des patrons de graphe visuels exprimant des contraintes positives et négatives sur le graphe à filtrer. Ces patrons servent à identifier un morphisme de graphes entre le membre gauche décoré par les conditions d'application et le graphe source. Les patrons de graphe peuvent s'avérer complexes, notamment dans le cas de contraintes négatives. Ils ne sont pas suffisamment expressifs. Le projet ANR Blanc CLIMT (*Categorical and Logical Methods in Model Transformation*) leur a préféré une logique visant à coder par des prédicats les conditions d'identification d'un graphe source. De façon similaire, le graphe cible est décrit par une formule logique vue comme un ensemble de contraintes à satisfaire. Ce choix s'est traduit par la définition d'un langage impératif expérimental Small-t$\mathcal{ALC}$ basé sur les logiques de description, plus spécifiquement sur la logique $\mathcal{ALCQI}$. Les formules ainsi établies jouent le rôle de précondition et de postcondition d'une règle et permettent un style de vérification à la Hoare. La section 1.1 précise le contexte d'étude et présente le langage Small-t$\mathcal{ALC}$. La problématique, l'objectif et les principales contributions de cette thèse font l'objectif de la section 1.2.

1.1 Le langage Small-t$\mathcal{ALC}$

Cette section définit le contexte de cette thèse en introduisant deux des approches les plus utilisées pour les transformations de graphes : l'approche algébrique, historiquement la plus ancienne et la plus répandue, est détaillée à la section 1.1.1, et l'approche logique adoptée par CLIMT explicitée à la section 1.1.2. La section 1.1.3 introduit le langage de transformation de graphes Small-t$\mathcal{ALC}$. Un exemple de programme Small-t$\mathcal{ALC}$ dans le domaine de la transformation de modèles est donné à la section 1.1.4.

1.1.1 L'approche algébrique

En préambule à l'approche Small-t$\mathcal{ALC}$, nous introduisons les concepts fondamentaux de la réécriture de graphes selon l'approche algébrique, bâtie sur un formalisme mathématique décrivant formellement comment transformer un graphe. La notion de morphisme de graphes est centrale aux approches algébriques.

Un graphe G orienté ayant arcs et nœuds étiquetés est constitué d'un ensemble fini de nœuds V, d'un ensemble fini d'arcs E, d'une fonction s qui associe un nœud source de V à chaque arc de E, d'une fonction t qui associe un nœud cible de V à chaque arc de E, d'une fonction l_V qui attribue une étiquette à chaque nœud de V et d'une fonction l_E qui attribue une étiquette à chaque arc de E.

Un morphisme de graphes entre deux graphes G et G' est défini par une application $m : G \to G'$ qui plonge le graphe G dans le graphe G', telle que l'image de G dans G' respecte les relations

d'adjacence présentes dans G et les étiquettes des nœuds et arcs de G. Intuitivement, G est contenu dans G'.

Dans ce contexte, un pas de réécriture consiste à appliquer la production $p : L \rightsquigarrow R$ à un graphe source G, où L et R sont des motifs de graphe exprimant respectivement la précondition et la postcondition de p. Si L correspond à un sous-graphe du graphe source G, alors la production p appliquée à G résulte en un graphe cible H dans lequel il existe un sous-graphe correspondant à R. Intuitivement, H est le résultat de la substitution de R à L dans G. Dans l'approche algébrique, ces patrons sont visuels, et la correspondance de L dans G est établie par un morphisme de graphes.

L'application d'une règle de production $p : L \rightsquigarrow R$ revient à coller les parties communes à deux graphes par une opération catégorique appelée somme amalgamée (*pushout*). Cette construction nécessite la définition d'un morphisme de graphes entre le graphe filtré par le membre gauche L de p et le membre droit R de p, en déterminant les nœuds et les arcs à préserver, ceux à supprimer et ceux à créer. Considérons la production p_1 inspirée de l'exemple donné par Corradini et al. [33] à la figure 1.1.

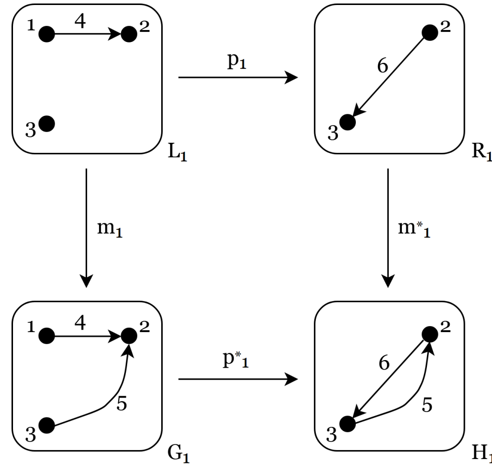


FIGURE 1.1 – Un exemple d'une règle de production simple [33]

Cette règle de production comprend trois nœuds 1, 2 et 3 dans le membre gauche L_1 et un arc 4 reliant 1 et 2, et connecte les nœuds 2 et 3 par un nouvel arc 6 comme indiqué par le membre droit R_1. Afin d'appliquer p_1 à un graphe source G_1, L_1 doit être associé à un sous-graphe de G_1 par un morphisme m_1, appelé correspondance, qui sert à repérer un sous-graphe de G_1 correspondant à L_1 : le graphe G_1 en excluant l'arc 5. Il reste alors à calculer le *pushout* des morphismes m_1 et p_1 pour obtenir le transformé H_1 de G_1 suivant la production p_1. La co-production p_1^* associe le graphe source G_1 au graphe transformé H_1 et la co-correspondance m_1^* relie la partie droite R_1 de la production à son occurrence dans H_1. Dans ce cadre, une transformation de G_1 en H_1 résultante

d'une application d'une règle de production p_1 et d'un morphisme de correspondance m_1 est notée $G_1 \xRightarrow{p_1, m_1} H_1$. Une grammaire de graphes est la donnée d'un ensemble de productions et d'un graphe initial.

La transformation n'est pas si simple dans d'autres cas. D'une part, L peut ne pas être toujours isomorphe à son image dans un graphe source G, et d'autre part, la suppression d'un nœud ayant des arcs entrants et sortants est problématique du fait qu'elle résulte en une structure différente d'un graphe. Considérons les deux exemples des règles de production p_2 et p_3 à la figure 1.2, inspirées de Corradini et al. [33].

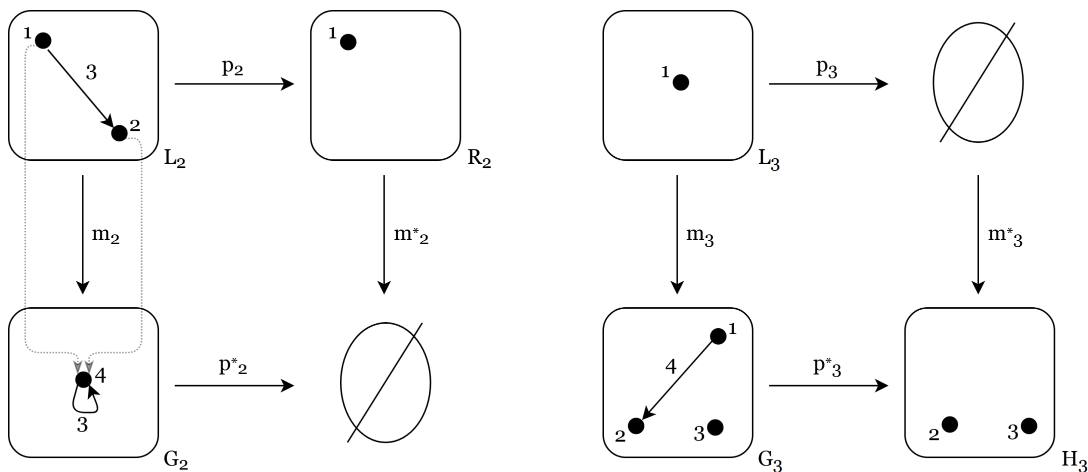


FIGURE 1.2 – Deux exemples de règles de production plus complexes [33]

La production p_2 supprime l'arc 3 entre les deux nœuds 1 et 2 et s'applique à un graphe G_2 composé de l'unique nœud 4 et d'un arc bouclant sur ce nœud. Les deux nœuds 1 et 2 de L_2 pointent ainsi vers le même nœud 4 de G_2. Dans ce cas, p_2 spécifie à la fois une suppression et une préservation du nœud 4, ce qui se traduit par un conflit pouvant être résolu de trois manières : (1) le nœud 4 peut être supprimé, (2) il peut être préservé, (3) la production peut être interdite. Dans la figure 1.2, la première alternative est considérée.

La production p_3 de la figure 1.2 montre que la suppression du nœud 1 du graphe G_3 entraîne la suppression du nœud source de l'arc 4. Dans ce cas, la cible H_3 ayant un arc suspendu ne respectera plus la définition d'un graphe. Cela peut être résolu de deux manières : (1) la suppression du nœud identifié avec ses arcs entrants et sortants, (2) l'interdiction d'appliquer p_3. Ici, la première alternative est prise en compte.

Deux approches traitent ces situations problématiques de deux manières différentes : l'approche *double pushout* (DPO) et l'approche *simple pushout* (SPO). Dans l'approche DPO, la transformation de ces situations n'est pas autorisée, et dès lors les productions p_2 et p_3 ne sont pas applicables. L'approche SPO, plus générale, les autorise tout en donnant la priorité à la suppression sur la préservation et en supprimant les arcs suspendus ; c'est ainsi que les productions p_2 et p_3 sont bien appliquées en SPO.

L'approche DPO [49], illustrée à la figure 1.3, consiste en deux *pushouts* $L \xleftarrow{l} K \xrightarrow{r} R$ ayant une interface commune K entre L et R appelée invariant. Chaque *pushout* exprime un morphisme total. L'application DPO consiste à : (1) identifier une correspondance de L dans le graphe source G, (2) supprimer de G, lors d'un premier *pushout*, le sous-graphe $L - K$ qui résulte en un graphe dit de contexte D où figurent toujours les éléments de K, (3) lors d'un deuxième *pushout*, ajouter à D tous les éléments de R ne figurant pas dans K. Cette dernière étape produit ainsi le graphe cible H.

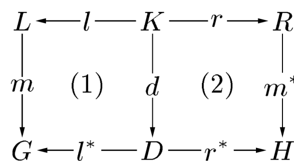


FIGURE 1.3 – L'approche DPO

Afin d'éviter les situations problématiques présentées, la correspondance m dans cette approche doit satisfaire une condition appelée condition de collage (*gluing condition*) qui consiste en deux conditions : (1) la première condition appelée condition de suspension exige que la suppression d'un nœud de G ne causera pas l'existence dans H d'arêtes suspendues, (2) la deuxième appelée condition d'identification s'assure que chaque nœud du graphe G pouvant être supprimé n'a qu'une seule occurrence dans L.

Dans certaines situations, il convient d'imposer des contraintes sur le morphisme de correspondance, notamment l'injectivité. Par exemple, sans l'injectivité, un système de transformation de graphes fini et convergent ne peut pas être obtenu. A noter qu'un système de transformation de graphes est dit convergent si les applications de règles répétées à un graphe d'entrée se terminent toujours et produisent un graphe de sortie unique. Le morphisme injectif rend ainsi l'approche DPO plus expressive car il offre un contrôle plus fin sur les transformations [60].

Dans une approche SPO, une production p consiste en un seul *pushout* $L \xrightarrow{p} R$ désignant un morphisme partiel de graphes comme le montre la figure 1.4. L'application de p consiste à : (1) identifier le sous-graphe de G qui correspond à L selon le morphisme partiel $m : L \to G$, (2) supprimer de G le sous-graphe identifié en plus des arcs suspendus, (3) ajouter à G le sous-graphe qui correspond à R.

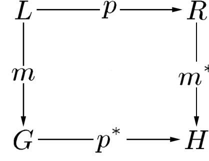


FIGURE 1.4 – L'approche SPO

L'approche SPO ignore l'application de la condition de collage. Ceci provoque éventuellement des effets indésirables sur un graphe comme illustré à la figure 1.2 ; d'une part, la suppression d'un nœud entraîne la suppression implicite de tous les arcs y entrant et sortant, d'autre part, le conflit entre la suppression et la préservation d'un nœud est résolu en faveur de la suppression. Dans ce cas, la co-correspondance m^* sera un morphisme partiel du fait que des éléments du membre droit R devant être préservés dans le graphe cible H sont supprimés. Par contre, les productions dans l'approche SPO sont plus expressives que dans DPO, car elles sont capables d'appliquer des transformations non autorisées de l'approche DPO.

Le filtrage d'un sous-graphe s'appuie sur des conditions d'application positives. Les règles de production peuvent aussi avoir des conditions négatives appelées NAC (*Negative Application Conditions*) [59]. Ce sont des patrons de graphe qui servent à définir des conditions que le graphe source ne doit pas satisfaire. Les conditions d'application négatives ont fait l'objet d'investigations dès lors qu'elles contribuent, au même titre que les conditions positives, au filtrage du graphe source [107].

1.1.2 Vers une approche basée sur une logique

La théorie des catégories [50] est à la base des approches algébriques. Il s'agit d'un cadre générique des mathématiques mettant l'accent sur les relations entre des objets (dans notre cas des graphes) par le biais de morphismes (dans notre cas des morphismes de graphes). Initialement, les travaux en réécriture de graphes ont prolongé ceux en réécriture de termes [31] et se sont concentrés sur des propriétés spécifiques à la réécriture comme la confluence et la terminaison [99], et non sur des propriétés relatives à la structure du graphe cible après transformation. Or raisonner sur la structure d'un graphe à partir de morphismes n'est pas simple [116]. Il y a donc nécessité à proposer de nouvelles méthodes qui garantissent la correction d'une transformation pour un passage à l'échelle. Deux voies sont envisageables, chacune visant à transférer le formalisme des graphes vers celui de la logique : l'une consiste à exploiter une logique dans laquelle les graphes sont citoyens de première classe, l'autre cherche à identifier une logique permettant d'encoder les graphes et leurs propriétés. Dans les deux cas, le gain concerne le raisonnement et les outils que l'on peut déployer pour la logique retenue, sur la base de techniques largement répandues.

Habel et Penneman [58] ont montré que les conditions imbriquées de graphes (*nested graph conditions*) qu'ils proposent peuvent être exploitées dans un cadre logique pour d'une part produire

des transformations correctes et d'autre part établir un calcul de plus faible précondition et de plus forte postcondition au sens de Dijkstra. Leurs travaux appartiennent à la première catégorie ; ils ont été étendus et confirmés par Lambers et Orejas [72] par la preuve de propriétés sur les graphes en utilisant la méthode des tableaux sémantiques. Les *E-conditions* du langage GP (*Graph Programs*) [104] enrichissent les conditions d'Habel et Penneman par la manipulation de labels et de contraintes.

La représentation des graphes par des structures relationnelles permet d'écrire directement et formellement leurs propriétés au moyen de formules logiques. C'est l'approche largement étudiée par Courcelle [35, 37, 38, 57], archétype de la deuxième classification, qui a proposé plusieurs logiques, dont la logique du premier ordre et la logique du second ordre monadique qui permet d'exprimer des problèmes comme la k-colorabilité, des propriétés de connexité ou encore l'existence de chemins de telle ou telle forme d'un nœud vers un autre. Ces logiques sont décidables ; elles sont descriptives et ne concernent qu'un nombre limité de graphes. De plus elles ne se dérivent pas aisément en algorithmes de transformation.

Dans le cadre de la deuxième classification, nous exploitons les logiques de description [14] capables naturellement de décrire des graphes afin d'appliquer et de raisonner sur des transformations de graphes. Plus précisément, l'un des sujets CLIMT concerne l'écriture de règles Small-t$\mathcal{ALC}$ correctes par construction. Ce langage est basé sur la logique de description $\mathcal{ALCQI}$ permettant à la fois de représenter les graphes, d'écrire les règles de transformation et de raisonner sur ces transformations.

Notre représentation des graphes est ensembliste : un graphe est composé d'un ensemble de nœuds et d'arcs typés [95, 111, 30] sur la base de catégories générales d'individus et de relations logiques que les individus ou catégories entretiennent entre eux. Par rapport aux approches algébriques qui mettent évidence des motifs de graphe et des morphismes de graphes pour définir l'applicabilité des règles, nous décrivons les prédicats que le graphe source doit satisfaire.

Ce changement de paradigme a été suggéré par Rensink [107] dans un objectif de généralisation des patrons de graphe. Il est illustré par la figure 1.5 qui met en évidence deux NAC : le premier pour indiquer que les deux nœuds connectés par a et b ne doivent pas être reliés par c et d à un troisième nœud, et le second qui interdit de fusionner les deux nœuds en un seul par le biais d'un morphisme injectif.

Ce patron de graphe peut s'exprimer formellement par la contrainte logique $\exists x, y \ / \ a(x, y) \wedge b(y, x) \wedge (\nexists z \ / \ c(x, z) \wedge d(y, z)) \wedge x \neq y$. Ainsi, la notation graphique, largement exploitée dans les systèmes de réécriture algébriques [114, 12, 117], ne permet pas par exemple de spécifier simplement que les nœuds x et y doivent être distincts. C'est ainsi que certains systèmes identifient par un numéro de nœud l'action des morphismes au regard des éléments du graphe concerné [117].

En lieu et place des patrons de graphe, nous proposons l'écriture de contraintes en logique du premier ordre sur la structure du graphe. Ces formules d'accès au graphe incluent des variables libres qui se comportent comme des références vers des nœuds du graphe. Le sous-graphe sélec-

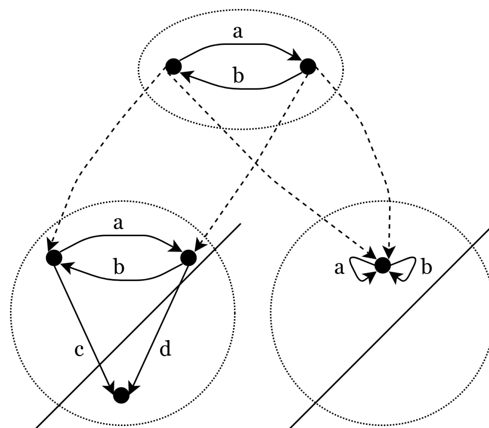


FIGURE 1.5 – Conditions d'application négatives [107]

tionné est ainsi manipulé par l'intermédiaire des instructions atomiques produisant un graphe cible attendu décrit aussi par une formule, vue comme un ensemble de contraintes à satisfaire. Dès lors, le même formalisme $\mathcal{ALCQI}$ code à la fois les conditions d'application positives et négatives, et les instructions du langage. Les prédicats permettent d'affirmer ou d'infirmer l'existence d'un nœud ou d'un arc, de spécifier des contraintes de multiplicité d'arcs entrants par rapport à un concept donné, et d'imposer l'égalité ou la non-égalité de références de nœuds.

1.1.3 Le langage Small-t$\mathcal{ALC}$

Le langage de transformation de graphes Small-t$\mathcal{ALC}$, proposé initialement en [29], est un langage de programmation impératif basé sur les logiques de description [14], une famille de langages qui permet de représenter formellement des connaissances. Plus spécifiquement, Small-t$\mathcal{ALC}$ est basé sur la logique $\mathcal{ALCQI}$ faisant l'objet de la sous-section 1.1.3.1. La syntaxe du langage est présentée aux sous-sections 1.1.3.2, 1.1.3.3 et 1.1.3.4.

1.1.3.1 De la logique $\mathcal{ALCQI}$ aux formules Small-t$\mathcal{ALC}$

Les logiques de description permettent de représenter formellement les connaissances d'un domaine spécifique en définissant les concepts ou les classes du domaine, et les rôles des relations binaires établies entre les éléments de ces classes. Par exemple, si *Claire* est une *Femme*, *Paul* est un *Homme*, et *Paul* est *mariéAvec Claire*, alors *Claire* et *Paul* sont des éléments du domaine, *Femme* et *Homme* sont les concepts, et *mariéAvec* est un rôle dans ce domaine.

Les différentes logiques de description se distinguent par les constructeurs qu'elles proposent. Ces constructeurs permettent la combinaison de concepts et rôles pour former des entités composées. La logique $\mathcal{AL}$ (*Attributive Language*), introduite par Schmidt-Schauß et Smolka [113], est minimale,

dans le sens où elle est la moins expressive des logiques de description. Les constructeurs de concepts associés sont les suivants :

C	(Concept atomique)
$\top$	(Concept universel)
$\bot$	(Concept impossible)
$\neg C$	(Négation atomique)
$C \cap C$	(Intersection)
$\exists\ r\ \top$	(Quantification existentielle limitée)
$\forall\ r\ C$	(Quantification universelle complète)

La quantification existentielle limitée exprime l'ensemble des éléments sources des relations étiquetées par le rôle r. La quantification universelle complète exprime l'ensemble des éléments sources des relations r ayant comme cible des éléments du concept C uniquement.

Notre langage Small-t$\mathcal{ALC}$ prend pour modèle la logique $\mathcal{ALCQI}$ (*Attributive Language with Complement, Qualifying number restrictions and Inverse roles*). Ainsi, en plus des descriptions déjà présentées, $\mathcal{ALCQI}$ offre des constructeurs sur les cardinalités des rôles et l'inverse d'un rôle :

$\geq n\ r\ C$	(Restriction de cardinalité qualifiée (au moins))
$\leq n\ r\ C$	(Restriction de cardinalité qualifiée (au plus))
r^{-1}	(Inverse d'un rôle)

Par rapport à un graphe, un concept atomique C exprime le type d'un nœud et un rôle r exprime le type d'un arc reliant une paire de nœuds. La sémantique de cette logique fait appel ainsi à la théorie des ensembles : un concept atomique C représente l'ensemble des nœuds appartenant à ce concept et un rôle r représente l'ensemble des paires de nœuds reliés par des arcs étiquetés r. Ainsi, chaque concept fait référence à l'ensemble des nœuds y appartenant.

Plus formellement, cette sémantique est associée à une interprétation $\mathcal{I}$, où le domaine $\Delta^{\mathcal{I}}$ définit les éléments du graphe et la fonction d'interprétation $.^{\mathcal{I}}$ associe à chaque concept un sous-ensemble de $\Delta^{\mathcal{I}}$ comportant les nœuds appartenant à ce concept, et à chaque rôle un sous-ensemble de $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$ comportant les paires de nœuds reliés par des arcs étiquetés par ce rôle. La fonction d'interprétation des constructeurs $\mathcal{ALCQI}$ [14] est définie comme suit :

$$
\begin{aligned}
&(\top)^{\mathcal{I}} = \Delta^{\mathcal{I}} \\
&(\bot)^{\mathcal{I}} = \emptyset \\
&(\neg C)^{\mathcal{I}} = \Delta^{\mathcal{I}} \backslash C^{\mathcal{I}} \\
&(C1 \cap C2)^{\mathcal{I}} = (C1)^{\mathcal{I}} \cap (C2)^{\mathcal{I}} \\
&(\exists\ r\ \top)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} |\ \exists y.(x,y) \in r^{\mathcal{I}}\} \\
&(\forall\ r\ C)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} |\ \forall y.(x,y) \in r^{\mathcal{I}} \Rightarrow y \in C^{\mathcal{I}}\} \\
&(\geq n\ r\ C)^{\mathcal{I}} = \{x, y \in \Delta^{\mathcal{I}} |\ |(x,y) \in r^{\mathcal{I}}| \geq n\}
\end{aligned}
$$

$(\leq n\ r\ C)^{\mathcal{I}} = \{x, y \in \Delta^{\mathcal{I}} |\ |(x, y) \in r^{\mathcal{I}}| \leq n\}$
$(r^{-1})^{\mathcal{I}} = \{(y, x) |\ (x, y) \in r^{\mathcal{I}}\}$

L'ensemble $\top$ est l'ensemble de tous les éléments du domaine, et $\bot$ se réduit à l'ensemble vide. $(\neg C)^{\mathcal{I}}$ présente l'ensemble des éléments du domaine n'appartenant pas à $C^{\mathcal{I}}$. L'intersection de deux concepts se ramène à l'intersection des éléments des deux ensembles. $(\exists\ r\ C)^{\mathcal{I}}$ est l'ensemble des nœuds qui sont reliés par r à des nœuds cibles de concept C. $(\forall\ r\ C)^{\mathcal{I}}$ dénote l'ensemble des nœuds sources des arcs r qui n'ont comme cibles que des nœuds de concept C. $(\geq n\ r\ C)^{\mathcal{I}}$ (respectivement $(\leq n\ r\ C)^{\mathcal{I}}$) est l'ensemble des nœuds qui sont connectés au plus (respectivement au moins) par n arcs étiquetés r ayant comme cibles des nœuds du concept C. $(r^{-1})^{\mathcal{I}}$ représente l'inverse du rôle $r^{\mathcal{I}}$.

Le langage Small-t$\mathcal{ALC}$ exploite les constructeurs de la logique $\mathcal{ALCQI}$ pour définir les formes d'un concept et d'un rôle Small-t$\mathcal{ALC}$ dont les syntaxes sont respectivement les suivantes :

Concept := *Ident*
| "all"
| "empty"
| '!' *Concept*
| *Concept* '&' *Concept*
| *Concept* '|' *Concept*
| '(' "some" *Rôle Concept* ')'
| '(' "only" *Rôle Concept* ')'
| '(' ">=" *Num Rôle Concept* ')'
| '(' "<=" *Num Rôle Concept* ')'
| '(' "=" *Num Rôle Concept* ')'

Rôle := *Ident*
| *Ident*-

Ident représente une chaîne de caractères qui désigne un nom d'un concept ou un nom d'un rôle. A noter que les mots clés *all* et *empty* dénotent respectivement les concepts *Top* ($\top$) et *Bottom* ($\bot$) des logiques de description, que !*Concept*, *Concept* & *Concept* et *Concept* | *Concept* expriment respectivement la négation, l'intersection et l'union des concepts, et que *some* et *only* signifient $\exists$ et $\forall$ respectivement. Small-t$\mathcal{ALC}$ propose aussi le constructeur d'union de deux concepts, d'interprétation $(C1 \cup C2)^{\mathcal{I}} = (C1)^{\mathcal{I}} \cup (C2)^{\mathcal{I}}$, que l'on dérive simplement de l'intersection et de la négation. *Num* est un entier désignant la cardinalité. Par exemple, $(\geq\ 2\ r\ C)$ représente l'ensemble des nœuds connectés à au moins 2 arcs r se dirigeant vers des nœuds de concept C. *Ident-* de *Rôle* s'assimile à l'inverse d'un rôle des logiques de description. Par exemple, $r-$ est l'inverse du rôle r.

La modélisation des connaissances en logique de description fait intervenir deux niveaux : la *TBox* et l'*ABox*. La *TBox* représente la terminologie du domaine et l'*ABox* déclare factuellement des

individus du domaine et leurs relations. Les instructions Small-t$\mathcal{ALC}$ manipulent impérativement et explicitement le niveau *ABox*, où les éléments du domaine correspondent aux nœuds d'un graphe éventuellement typés par des concepts, et les relations binaires entre les nœuds sont étiquetées par des rôles. Ces instructions se dérivent aisément en assertions, principalement des prédicats unaires pour les étiquettes des nœuds et des prédicats binaires pour celles des arcs. Cette forme déclarative permet de refléter la mise à jour des faits *ABox* lors de la transformation, elle est complétée par des tests d'égalité et d'inégalité des variables du code désignant des références de nœud du graphe. Les faits Small-t$\mathcal{ALC}$ sont définis comme suit :

$$\begin{aligned} AFait := & \ Ident \text{ ` :' } Concept \\ & \mid Ident \ R\hat{o}le \ Ident \\ & \mid Ident \text{ `!' } R\hat{o}le \ Ident \\ & \mid Ident \text{ `=' } Ident \\ & \mid Ident \text{ ``!='' } Ident \end{aligned}$$

Le premier *AFait* exprime le typage d'un nœud par un concept. Le deuxième et le troisième expriment respectivement l'existence et la non-existence d'un arc entre deux nœuds. Les deux derniers *AFaits* expriment l'égalité et l'inégalité de références vers des nœuds du graphe. Par exemple, le fait $x \; !r \; y$ signifie la non-existence d'un arcs r entre les deux nœuds x et y.

Les *AFormules* utilisées pour décrire des contraintes d'un graphe sont des combinaisons booléennes de *AFaits* :

$$\begin{aligned} AFormule := & \ AFait \\ & \mid \text{`(' `!' } AFormule \text{ `)'} \\ & \mid AFormule \text{ ``and'' } AFormule \\ & \mid AFormule \text{ ``or'' } AFormule \end{aligned}$$

La combinaison de *AFaits* désignant des assertions sur des nœuds et des arcs traduit un graphe. Par exemple, l'*AFormule* $x : C \ and \ x : (\geq \ 2 \ r \ C) \ and \ y : C \ and \ x \; !r \; y$ décrit un modèle de graphe où x est de concept C et relié par r à au moins deux nœuds de concept C, y est de concept C et x n'est pas connecté par r à y. Un tel graphe est schématisé à la figure 1.6.

Ces formules possèdent des variables libres dont chacune fait référence à un nœud du graphe d'une manière non-déterministe. Le nœud référencé doit ainsi satisfaire toutes les assertions concer-

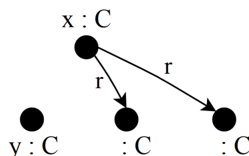


FIGURE 1.6 – Un modèle de graphe de $x : C \ and \ x : (\geq \ 2 \ r \ C) \ and \ y : C \ and \ x \; !r \; y$

nant la variable en question dans la formule. En raison de ce non-déterminisme, plusieurs variables de la formule peuvent éventuellement faire référence à un même nœud du graphe. Les assertions exprimant des relations d'inégalité entre les nœuds permettent d'interdire ces cas et de forcer ainsi un morphisme injectif.

1.1.3.2 Les instructions du langage

Pour manipuler un graphe, le langage dispose de cinq instructions atomiques. Les instructions $add(i : C)$ et $delete(i : C)$ permettent d'ajouter le nœud i au concept C et de le supprimer de C. A noter que ces instructions n'ajoutent, ni suppriment les nœuds physiquement du graphe. Autrement dit, étant dans une approche ensembliste, l'ajout d'une instance à un concept s'apparente à un typage de cette instance par ce concept. La suppression exclut également le nœud du concept impliqué. Les instructions $add(i\ r\ j)$ et $delete(i\ r\ j)$ ajoutent et suppriment l'arc étiqueté r reliant i et j. L'instruction *select* v *with* F où v est un ensemble de variables et F est une *AFormule*, permet de sélectionner des nœuds du graphe satisfaisant les assertions de F et de les affecter aux variables de v. Par exemple, *select* i *with* $i : C$ permet de sélectionner la référence d'un nœud de concept C dans le graphe et l'affecter à i. Dans un programme, si aucun nœud du graphe ne satisfait F, alors une erreur se déclenche et l'exécution s'arrête.

Considérons le graphe source représentant la formule $x : C$ *and* $x : (\geq\ 2\ r\ C)$ *and* $y : C$ *and* $x\ !r\ y$ à la figure 1.6, et la suite d'instructions Small-t$\mathcal{ALC}$ définie à la figure 1.7. Le graphe de la figure 1.8 est un résultat possible de l'application de ces instructions au graphe source. En effet, l'instruction non-déterministe *select* sélectionne un nœud arbitraire dans le graphe relié par r à x. Dans le graphe source donné, il existe deux nœuds ayant cette propriété, d'où la sélection non-déterministe

```
select z with x r z;
delete(x r z);
add(x r y);
delete(y : C);
add(x : D);
```

FIGURE 1.7 – Une suite d'instructions Small-t$\mathcal{ALC}$

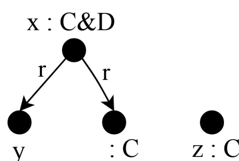


FIGURE 1.8 – Mise à jour du graphe de la figure 1.6 après l'application d'une suite d'instructions

de l'un des deux. Remarquons aussi qu'après la suppression du nœud y du concept C, ce nœud existe toujours dans le graphe, mais sans être typé d'un concept. Également, l'ajout du nœud x au concept D, autre que C, lui confère le type $C \cap D$.

Small-t$\mathcal{ALC}$ dispose aussi de structures de contrôle conventionnelles comme la séquence, la sélection et la répétition. Pour des raisons de vérification, la répétition exige en plus un invariant de boucle, c'est-à-dire une formule qui doit être vraie avant et après chaque itération. La syntaxe des instructions Small-t$\mathcal{ALC}$ est définie comme suit :

$$
\begin{aligned}
Instruction := \ & \text{“add” ‘(’ } Ident \text{ ‘ :’ } Ident \text{ ‘)’ ‘;’} \\
& \mid \text{“delete” ‘(’ } Ident \text{ ‘ :’ } Ident \text{ ‘)’ ‘;’} \\
& \mid \text{“add” ‘(’ } Ident\ Ident\ Ident \text{ ‘)’ ‘;’} \\
& \mid \text{“delete” ‘(’ } Ident\ Ident\ Ident \text{ ‘)’ ‘;’} \\
& \mid \text{“select” } Ident \ \{\text{‘,’ } Ident\}^* \text{ “with” } AFormule \text{ ‘;’} \\
& \mid Instruction\ Instruction \\
& \mid \text{“if” ‘(’ } AFormule \text{ ‘)’ “then” ‘\{’ } Instruction \text{ ‘\}’ [“else” ‘\{’ } Instruction \text{ ‘\}’]} \\
& \mid \text{“inv” ‘ : ’ } AFormule \text{ “while” ‘(’ } AFormule \text{ ‘)’ “do” ‘\{’ } Instruction \text{ ‘\}’}
\end{aligned}
$$

Plus formellement, les règles d'inférence à la figure 1.9 définissent la sémantique axiomatique des instructions Small-t$\mathcal{ALC}$. Chaque règle est vue comme une relation liant une prémisse et une

$$\frac{}{\{P[C+i \setminus C]\}\ \texttt{add}\ (i:C)\ \{P\}}\ (\textsc{addC}) \qquad \frac{}{\{P[C-i \setminus C]\}\ \texttt{delete}\ (i:C)\ \{P\}}\ (\textsc{delC})$$

$$\frac{}{\{P[r+(i,j) \setminus r]\}\ \texttt{add}\ (i\ r\ j)\ \{P\}}\ (\textsc{addR}) \qquad \frac{}{\{P[r-(i,j) \setminus r]\}\ \texttt{delete}\ (i\ r\ j)\ \{P\}}\ (\textsc{delR})$$

$$\frac{P\ \wedge \forall v(F \Rightarrow Q)}{\{P\}\ \texttt{select}\ v\ \texttt{with}\ F\ \{Q\}}\ (\textsc{select}) \qquad \frac{\{P\}\ s1\ \{Q\} \quad \{Q\}\ s2\ \{R\}}{\{P\}\ s1; s2\ \{R\}}\ (\textsc{seq})$$

$$\frac{\{P \wedge c\}\ s\ \{Q\} \quad \{P \wedge \neg c\} \Rightarrow \{Q\}}{\{P\}\ \texttt{if}\ c\ \texttt{then}\ s\ \{Q\}}\ (\textsc{if}) \qquad \frac{\{P \wedge c\}\ s1\ \{Q\} \quad \{P \wedge \neg c\}\ s2\ \{Q\}}{\{P\}\ \texttt{if}\ c\ \texttt{then}\ s1\ \texttt{else}\ s2\ \{Q\}}\ (\textsc{if-else})$$

$$\frac{\{P \wedge c\}\ s\ \{P\}}{\{P\}\ \texttt{while}\ c\ \texttt{do}\ s\ \{P \wedge \neg c\}}\ (\textsc{while})$$

Figure 1.9 – Sémantique axiomatique des instructions Small-t$\mathcal{ALC}$

conclusion, où la conclusion est dite déductible de la prémisse. Si la prémisse est vide, alors la conclusion est un axiome. Les quatre premières règles mettent en évidence des substitutions notées [valeur\variable] indiquant que la variable est remplacée par la valeur. Par exemple, pour la règle ADDC, si P est valide en substituant $C + i$ au concept C, alors P sera valide après application de l'instruction $add(i : C)$. Les axiomes DELC, ADDR et DELR s'interprètent de la même façon.

Étant donné que *select* est une instruction d'affectation, la règle SELECT s'écrit différemment. Elle est telle que, quelles que soient les variables à sélectionner, la formule F du *select* doit impliquer une postcondition Q en supposant la précondition P valide. Les règles SEQ, IF, IF-ELSE et WHILE correspondent aux structures de contrôle conventionnelles. Les conditions des constructions *if* et *while* sont considérées comme des requêtes booléennes sur le graphe.

1.1.3.3 Programme Small-t$\mathcal{ALC}$

Un programme Small-t$\mathcal{ALC}$ est constitué de plusieurs règles et d'un point d'entrée spécifiant leur enchaînement. Une règle consiste en une précondition P, une séquence d'instructions S et une postcondition Q formant ainsi un triplet de Hoare $\{P\}S\{Q\}$ qui signifie que si le prédicat P est vrai et si S termine, alors Q est vrai après l'exécution de S [64]. La précondition P d'une règle est exploitée à l'exécution et joue le rôle d'un morphisme de graphes afin de filtrer le sous-graphe d'un graphe sur lequel la transformation s'applique. Les assertions P et Q portent ainsi sur les variables manipulées dans S. Ce formalisme permet de vérifier les règles de transformation avec le calcul de E. W. Dijkstra [47] qui détermine, pour une postcondition Q et une instruction donnée S, quelle est la plus faible précondition qui assure, lorsque S termine, que Q est vrai après S. La plus faible précondition est notée $wp_S(Q)$ (pour *Weakest Precondition*) ou encore $wp(S, Q)$. Un triplet de Hoare est ainsi prouvé correct dans le cas où $P \Rightarrow wp(S, Q)$. La syntaxe d'une règle Small-t$\mathcal{ALC}$ est la suivante :

Règle := "rule" $Ident$ '{' "pre" ' :' $AFormule$ ';' *Instruction* "post" ' :' $AFormule$ ';' '}'

La figure 1.10 présente la règle *update* ayant comme précondition la formule représentée par le graphe de la figure 1.6 et la suite d'instructions de la figure 1.7. Après l'application de ces instructions, tout graphe cible doit respecter la postcondition de la règle. C'est le cas du graphe de la figure 1.8.

Les règles Small-t$\mathcal{ALC}$ traduisent des transformations sur des instances sélectionnées par les préconditions des règles. Or, définir les règles séparément ne permet pas d'obtenir un programme de transformation. Un enchaînement de ces règles doit être défini. Dans ce cadre, on considère qu'un programme Small-t$\mathcal{ALC}$ est une application modulaire de règles de transformation indépendantes portant sur des instances particulières. Nous proposons d'appeler ces règles depuis un point d'entrée du programme. Il est à noter que le projet CLIMT ne prévoyait pas cette fonctionnalité, se limitant uniquement à l'examen d'une règle par le prouveur.

```
rule update {
  pre: x : C and x : (≥ 2 r C) and y : C and x !r y;
  select z with x r z;
  delete(x r z);
  add(x r y);
  delete(y : C);
  add(x : D);
  post: x : C & D and x : (≥ 1 r C) and y : !C and x r y;
}
```

FIGURE 1.10 – Règle *update*

Les règles manipulant des instances spécifiées dans la précondition ne permettent pas de traiter aisément un ensemble d'instances appartenant à un concept donné ou ayant des propriétés spécifiques. Dès lors, outre l'appel simple d'une règle, nous proposons une instruction d'appel d'une règle dit itératif qui s'applique tant que des instances satisfont la précondition. Par exemple, en supposant qu'une précondition d'une règle donnée est $a : A$, cet appel itératif permet de sélectionner, à chaque itération de la règle, une nouvelle instance a de concept A dans le graphe.

Le point d'entrée d'un programme Small-t$\mathcal{ALC}$ se compose ainsi en séquence d'appels simples et itératifs de règles et met en œuvre implicitement des structures de contrôle : un appel simple d'une règle ne respectant pas la précondition s'assimile à une sélection, et l'appel itératif qui répète la séquence d'instructions d'une règle aussi longtemps que possible s'assimile à une répétition.

Dès lors, le point d'entrée *main* identifie deux formats d'appel : l'appel classique d'une règle en évoquant son nom et l'appel itératif d'une règle spécifié par un point d'exclamation précédé du nom de la règle concernée. La sémantique axiomatique des appels de règles Small-t$\mathcal{ALC}$ est fournie à la figure 1.11.

$$\frac{\{P\}\ S\ \{Q\} \quad corps(r) = S}{\{P\}\ r\ \{Q\}}\ (\textsc{appel}) \qquad \frac{\{P\}\ r\ \{Q\} \vdash \{P\}\ S\ \{Q\} \quad corps(r) = S}{\{P\}\ r\ \{Q\}}\ (\textsc{appel}\,!)$$

$$\frac{\{P\}S1\{R\}\ \{R\}S2\{Q\} \quad corps(r1) = S1 \quad corps(r2) = S2}{\{P\}r1; r2\{Q\}}\ (\textsc{appel1}\,;\textsc{appel2})$$

FIGURE 1.11 – Sémantique axiomatique des appels de règles Small-t$\mathcal{ALC}$

APPEL exécute le corps de la règle r s'il existe dans le graphe des instances ayant les propriétés précisées par la précondition P avant l'exécution. De ce fait, si le corps S de la règle r transforme un état P en Q, l'appel r transforme P en Q. Le deuxième format d'appel APPEL ! désigne une itération de l'application de r. Il s'exprime sémantiquement par induction : on vérifie que l'appel courant (n) de la règle est correct si les appels antérieurs le sont. Le symbole $\vdash$ indique que le fait $\{P\}S\{Q\}$ est prouvable à partir de $\{P\}r\{Q\}$ désignant l'appel $(n-1)$. Si c'est le cas et que le corps de r est S, on prouve par conséquent l'appel n de la règle. L'hypothèse d'induction nous garantit la correction d'un appel itératif tant que le programme satisfait la précondition. Par contre, cette sémantique ne donne pas une correction totale mais une correction partielle du fait qu'elle ne prouve pas la terminaison du programme.

La séquence d'appels APPEL1 ;APPEL2 indique que si une règle APPEL1 transforme un état de calcul P en un état de calcul R et si une règle APPEL2 transforme ce même état R en un autre état Q, alors la séquence APPEL1 ;APPEL2 transforme l'état P en Q.

On définit dès lors la syntaxe d'un point d'entrée *main* Small-t$\mathcal{ALC}$ par une seule règle de production comme suit :

$$Main := \text{“main” ‘\{’ \{ } \textit{Ident} \text{ [!] ‘;’\}* ‘\}’}$$

Remarquons que les règles appelées sont sans paramètres ; les valeurs des variables sont initialisées lors d'une sélection non-déterministe de nœuds dans le graphe source satisfaisant les assertions de la précondition ou par une instruction *select*.

La figure 1.12 illustre le point d'entrée du programme faisant appel à la règle *update* de la figure 1.10.

```
main {
  update;
}
```

FIGURE 1.12 – Point d'entrée du programme

L'application d'un programme Small-t$\mathcal{ALC}$ commence ainsi par l'appel de son *main* qui, à son tour, appelle les règles définies par le programme. La syntaxe d'un programme Small-t$\mathcal{ALC}$ est comme suit :

$$Programme := \text{“program” } \textit{Ident} \text{ ‘\{’ \{}\textit{Règle}\text{\}* } Main \text{ ‘\}’}$$

1.1.3.4 Graphe Small-t$\mathcal{ALC}$

L'application d'une transformation Small-t$\mathcal{ALC}$ nécessite en entrée un graphe source défini textuellement sous la forme d'une séquence d'instructions *add* qui y ajoutent physiquement des nœuds et des arcs :

$$\begin{aligned} Ajout := & \text{"add" '(' } Ident \text{ ' :' } Ident \text{ ')' ';'} \\ & | \text{ "add" '(' } Ident\ Ident\ Ident \text{ ')' ';'} \end{aligned}$$

La syntaxe d'un programme définissant un graphe est comme suit :

$$Graphe := \text{"graph" } Ident \text{ '\{' } \{Ajout\}^* \text{ '\}'}$$

Par exemple, le code Small-t$\mathcal{ALC}$ de la figure 1.13 définit une structure de graphe g équivalente à celle du graphe de la figure 1.6.

```
graph g {
 add(x1 : C);
 add(x2 : C);
 add(x3 : C);
 add(x4 : C);
 add(x1 r x2);
 add(x1 r x3);
}
```

FIGURE 1.13 – Exemple d'une séquence d'instructions définissant un graphe

La fonctionnalité de l'environnement Small-t$\mathcal{ALC}$ permettant d'exécuter un programme sera précisée au chapitre 4 de la thèse.

1.1.4 Transformation d'un diagramme de classes UML

Étant donné que les graphes sont capables naturellement de décrire des modèles, l'exemple dans cette section concerne le domaine de l'ingénierie des modèles, où un diagramme de classes UML est transformé en un modèle relationnel.

Un diagramme de classes est utilisé en génie logiciel pour montrer la structure interne d'un système en présentant les différentes classes et interfaces du système et leurs relations. Le méta-modèle simplifié correspondant à cette spécification est donné à la figure 1.14.

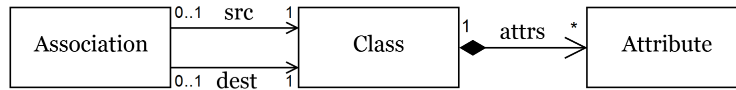


Figure 1.14 – Méta-modèle simplifié du diagramme de classes

La figure 1.15 instancie ce méta-modèle par deux objets de type anonyme *Class* ayant chacune un attribut primaire et une association anonyme entre les deux.

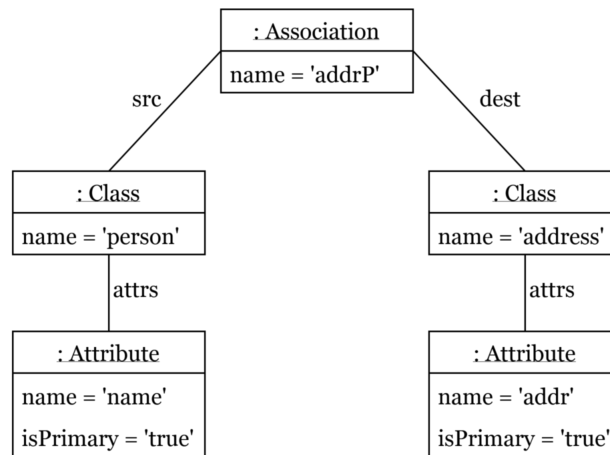


Figure 1.15 – Instance d'un diagramme de classes

Un modèle relationnel permet d'organiser des données sous forme de tables. Le méta-modèle correspondant est donné à la figure 1.16

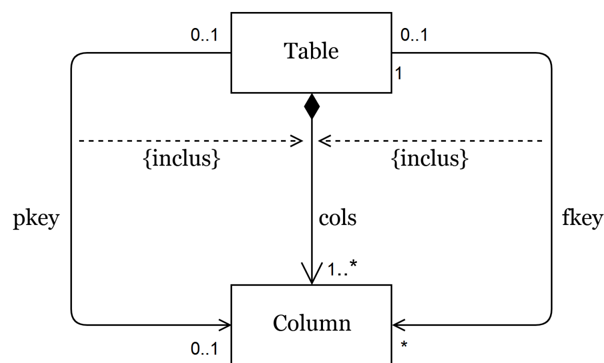


Figure 1.16 – Méta-modèle du modèle relationnel

La figure 1.17 montre une instance de ce méta-modèle illustrant le même exemple que celui de la figure 1.15.

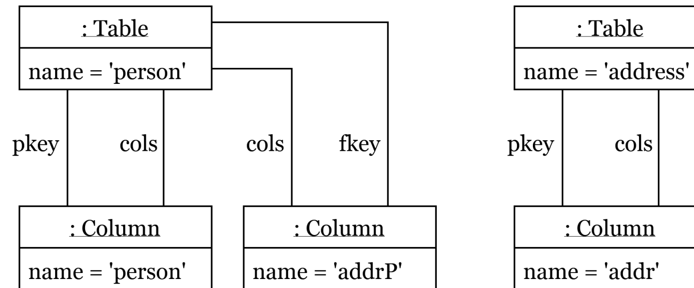


FIGURE 1.17 – Instance d'un modèle relationnel

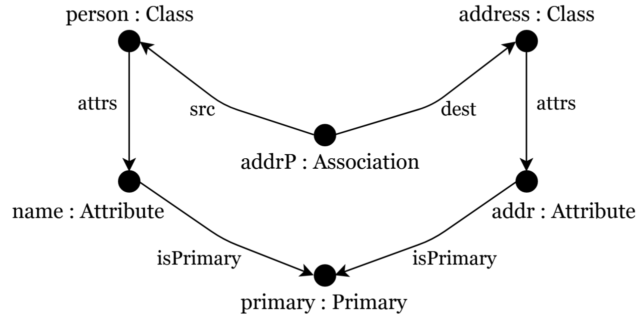
Les deux instances des méta-modèles peuvent s'assimiler à des ensembles de nœuds reliés par des arcs. Ainsi, le graphe source du programme traduit une instance du méta-modèle du diagramme de classes : chaque classe est représentée par un nœud, et est reliée, par des arcs étiquetés *attrs*, à des nœuds représentant ses propres attributs. Le choix d'un attribut comme clé primaire d'une classe est modélisé par un arc *isPrimary* entre le nœud représentant cet attribut et un nœud de type *Primary*. Les associations sont représentées par des nœuds, et sont reliées chacune à une classe source par une relation *src*, et à une classe cible par une relation *dest*.

Le programme Small-t$\mathcal{ALC}$ *uml2rdb* transformant une instance d'un diagramme de classes en une instance du modèle relationnel est présenté à la figure 1.18. Ce programme consiste en quatre règles de transformation : *class2Table* transforme une classe en une table, *attribute2Column* transforme un attribut d'une classe transformée en une colonne de la table correspondante, *setPrimaryKey* transforme une colonne désignée comme attribut primaire d'une classe en une relation *pkey* entre la table associée et la colonne, et *setForeignKey* transforme une association entre deux classes en une colonne d'une table munie de références vers l'autre table. Ces références se traduisent dans le graphe cible par des relations *ref* qui mettent en œuvre l'intégrité référentielle en liant les nœuds représentant les clés étrangères d'une table aux nœuds représentant les clés primaires d'une autre table. Les quatre règles sont appelées itérativement en séquence depuis le point d'entrée du programme.

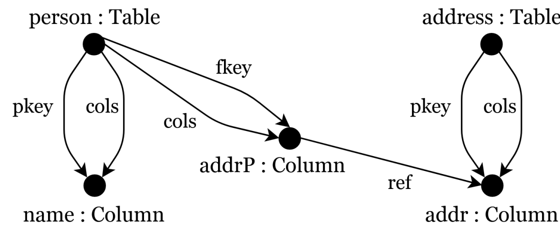
L'exécution du programme *uml2rdb* nécessite en entrée un graphe source fourni par le développeur comme mentionné à la sous-section 1.1.3.4. Ce graphe ainsi que le graphe transformé sont décrits à la figure 1.19.

```
program uml2rdb {
 rule class2Table {
  pre: c : Class;
  delete(c : Class);
  add(c : Table);
  post: c : Table and c : !Class;
 }
 rule attribute2Column {
  pre: t : Table and t : (≥ 1 attrs Attribute);
  inv: t : Table;
  while(t : (≥ 1 attrs Attribute)) do {
   select a with a : Attribute and t attrs a;
   delete(a : Attribute);
   add(a : Column);
   delete(t attrs a);
   add(t cols a);
  }
  post: t : Table and t : (≥ 1 cols Column) and t : (= 0 attrs Attribute);
 }
 rule setPrimaryKey {
  pre: col : Column and t : Table and t cols col and col isPrimary primary;
  delete(col isPrimary primary);
  add(t pkey col);
  post: col : Column and t : Table and t cols col and col !isPrimary primary
        and t pkey col;
 }
 rule setForeignKey {
  pre: ass : Association and ass src t1 and ass dest t2 and t2 pkey pkColumn;
  delete(ass : Association);
  add(ass : Column);
  add(t1 cols ass);
  add(t1 fkeys ass);
  add(ass ref pkColumn);
  delete(ass src t1);
  delete(ass dest t2);
  post: ass : Column & !Association and t1 cols ass and t1 fkeys ass
        and ass ref pkColumn;
 }
 main {
  class2Table!;
  attribute2Column!;
  setPrimaryKey!;
  setForeignKey!;
 }
}
```

FIGURE 1.18 – Programme Small-t$\mathcal{ALC}$ de transformation d'un diagramme de classes UML en un modèle relationnel



(a) Graphe source



(b) Graphe cible

FIGURE 1.19 – Graphes source et cible du programme Small-t$\mathcal{ALC}$ de la figure 1.18

Remarquons que Small-t$\mathcal{ALC}$ effectue des transformations sur place qui fusionnent le graphe source et cible. Les transformations peuvent être endogènes, ou exogènes comme dans cet exemple.

1.2 L'assistance au développement Small-t$\mathcal{ALC}$

Cette section présente la problématique, l'objectif et les contributions de cette thèse. La difficulté à écrire un triplet correct et à analyser une règle erronée nous conduit à proposer un environnement Small-t$\mathcal{ALC}$ permettant d'assister à écrire, exécuter et vérifier des transformations de graphes comme expliqué à la sous-section 1.2.2. Nos contributions principales sont ainsi :

1. un extracteur de préconditions *ABox* ayant pour but d'assister un développeur à produire une règle correcte par construction,
2. un moteur d'inférence qui exploite le niveau *TBox* de la logique $\mathcal{ALCQI}$ pour exprimer des propriétés globales du graphe,

3. d'autres diagnostics *ABox* et *TBox* permettant de localiser des erreurs dans les programmes Small-t$\mathcal{ALC}$.

Ces contributions sont résumées à la sous-section 1.2.3 ; elles font suite à la nature du verdict du composant de preuve Small-t$\mathcal{ALC}$ résumé à la sous-section 1.2.1.

1.2.1 Problématique

L'objectif principal de la conception du langage Small-t$\mathcal{ALC}$ est d'exécuter et de vérifier des transformations de graphes. Basé sur la logique $\mathcal{ALCQI}$, ce langage permet de raisonner sur les règles de transformation *ABox* en les exprimant sous la forme de triplets de Hoare. Une transformation est prouvée correcte si pour chaque graphe respectant la précondition, la postcondition est satisfaite après l'exécution de la transformation.

Pour automatiser ce processus de vérification, un composant de preuve Small-t$\mathcal{ALC}$ a été développé [17] dans le cadre du projet ANR Blanc CLIMT. Il s'agit de calculer la plus faible précondition wp à partir du code de transformation S et de la postcondition Q d'une règle *ABox*, puis de vérifier l'implication entre la précondition donnée P et celle calculée, soit $P \Rightarrow wp(S, Q)$. Comme le montre la figure 1.20, la non-vérification de cette implication résulte en un contre-exemple présentant un graphe qui respecte la précondition du triplet mais qui ne satisfait pas la postcondition.

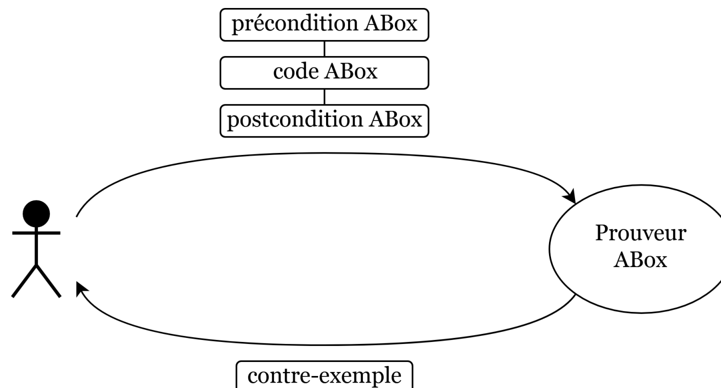


Figure 1.20 – Composant de preuve ABox

Ce prouveur Small-t$\mathcal{ALC}$ est la seule technique proposée pour analyser et prouver des règles. Il exige l'écriture de la précondition et de la postcondition en plus du code de transformation. Or, construire des triplets en ayant des spécifications cohérentes avec la transformation n'est pas une tâche facile pour les développeurs : un contre-exemple produit par le prouveur en cas d'échec s'avère parfois compliqué et non évident à analyser. Ainsi, la rétroaction du composant de preuve Small-t$\mathcal{ALC}$ ne permet pas de trouver les erreurs aisément dans une règle. De plus, ce prouveur se

limite à l'examen de chaque règle séparément et n'apporte ainsi aucune aide à la conception d'un programme Small-t$\mathcal{ALC}$ qui consiste en une séquence de plusieurs règles de transformation.

Considérons le triplet de la règle *ABox* de la figure 1.21. Dans cet exemple, la précondition indique que le nœud a est de concept A, et que a est lié par r à au moins trois nœuds de A. Le code de la transformation sélectionne d'abord un nœud n de concept A et lié à a par r, puis supprime l'arc étiqueté r entre a et n et supprime a du concept A. Il semble plausible que la postcondition soit telle que a ne soit plus de concept A et qu'il soit connecté à au moins deux nœuds de A. Par contre, en soumettant ce triplet au composant de preuve Small-t$\mathcal{ALC}$, la preuve échoue et le contre-exemple de la figure 1.22 est fourni.

```
pre: a : A and a : (≥ 3 r A);
select n with a r n and n : A;
delete(a r n);
delete(a : A);
post: a :!A and a : (≥ 2 r A);
```

FIGURE 1.21 – Triplet de Hoare incorrect

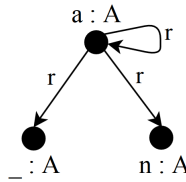


FIGURE 1.22 – Contre-exemple du triplet de la figure 1.21

A première vue, il n'est pas évident d'identifier l'erreur dans le triplet. En effet, en supprimant l'arc r entre a et n, le nombre de restrictions du fait $a : (\geq 3\ r\ A)$ décrémente et sera ainsi $a : (\geq 2\ r\ A)$. Mais l'instruction $delete(a : A)$ affecte aussi la restriction. Ceci peut être remarqué dans le contre-exemple produit où il existe un arc r sortant de a vers a qui était préalablement de concept A. Dans ce cas, la suppression du nœud a de l'ensemble associé au concept A réduit le nombre d'arcs entrants vers les nœuds de concept A et donc la restriction diminue à 1 soit $a : (\geq 1\ r\ A)$.

Le contre-exemple donné par le prouveur en cas d'échec d'un triplet n'est pas toujours facile à interpréter, et donc l'identification des erreurs reste une tâche difficile. Dans ce contexte, il s'agit d'accompagner le développeur dans l'écriture de ses transformations en lui fournissant des diagnostics plus riches que celui fourni par ce seul outil formel.

1.2.2 Objectif

Nous visons dans cette thèse à aider un développeur à atteindre des transformations de graphes correctes en l'incluant dans la boucle d'élaboration d'un triplet *ABox*. Un développeur ne sera pas alors contraint de donner un triplet complet pour l'interpréter et le vérifier. Au contraire, le but est de démarrer avec des bouts de code pour construire au fur et à mesure des programmes corrects en lui donnant la possibilité d'utiliser plusieurs outils exploitant diverses techniques formelles et semi-formelles d'analyse d'un code.

D'autre part, vu que les pré- et postconditions d'une règle spécifient uniquement des propriétés *ABox* sur des nœuds et arcs particuliers du graphe, nous proposons de plus une vérification à un niveau plus abstrait permettant d'exprimer des propriétés globales d'un graphe pouvant être examinées à différents points d'un programme. En exploitant la *TBox* des logiques de description, ces propriétés globales Small-t$\mathcal{ALC}$ peuvent se traduire par des relations d'inclusion entre les concepts.

Afin d'assister le développeur à établir des vérifications à ces deux niveaux *ABox* et *TBox*, des analyses des règles sont proposées permettant de traiter des triplets partiels ou complets. L'approche logique adoptée facilite naturellement l'analyse d'un code de transformation en considérant des spécifications *ABox* et *TBox* données. En effet, cette approche basée sur la logique $\mathcal{ALCQI}$ qui traduit aisément les graphes, permet de représenter le code et sa spécification avec le même formalisme comme le montre le tableau 1.1 pour l'exemple d'un nœud x appartenant au concept C : les pré- et postconditions sont exprimées par des formules Small-t$\mathcal{ALC}$ et le code manipule les faits de ces formules. Cette similitude est exploitée afin d'analyser au niveau *ABox* un triplet donné éventuellement incomplet et d'étudier au niveau *TBox* l'effet d'une règle sur un graphe dans sa globalité :

— Au niveau *ABox*, dans le cas où le code de transformation et une postcondition sont fournis, une analyse statique *ABox* peut être effectuée en mode régressif pour extraire une précondition. Par ailleurs, si le code et la spécification sont déjà écrits, un générateur de cas de test permet d'identifier éventuellement des erreurs dans le triplet en générant automatiquement des tests dans une syntaxe proche de celle du langage. Ceci peut être fait soit par analyse

Table 1.1 – Variation de l'instanciation $x : C$

Entité Small-t$\mathcal{ALC}$	Ecriture Small-t$\mathcal{ALC}$
Instruction *ABox*	$add(x : C)$
Pré- et postcondition *ABox*	$x : C$
Assertion *TBox*	$C = \{.., x, ..\}$
Graphe Small-t$\mathcal{ALC}$	$\bullet\ x : C$
Test Small-t$\mathcal{ALC}$	$assertExistNode(x, C)$

statique *ABox* du code en comparant les spécifications données et extraites, soit par analyse dynamique *ABox* en générant un graphe d'entrée respectant la précondition.

— Au niveau *TBox*, deux analyses sont mises en place : la première, statique, permet d'inférer des assertions *TBox* à partir d'une assertion *TBox* initiale et des assertions *ABox* des règles, et la deuxième, dynamique, vérifie des formules *TBox* par le biais d'un raisonneur.

Le tableau 1.2 résume les objectifs fixés de cette thèse avec l'outil Small-t$\mathcal{ALC}$ correspondant à chacun.

TABLE 1.2 – Les objectifs et les outils Small-t$\mathcal{ALC}$ proposés

Niveau	Objectif	Outil
ABox	Vérifier formellement un triplet	Prouveur *ABox*
	Construire un triplet correct	Extracteur de préconditions *ABox*
	Identifier des erreurs dans un triplet	Générateur de cas de test *ABox*
TBox	Examiner l'évolution des propriétés *TBox*	Moteur d'inférence *TBox*
	Vérifier dynamiquement des propriétés *TBox*	Raisonneur *TBox*

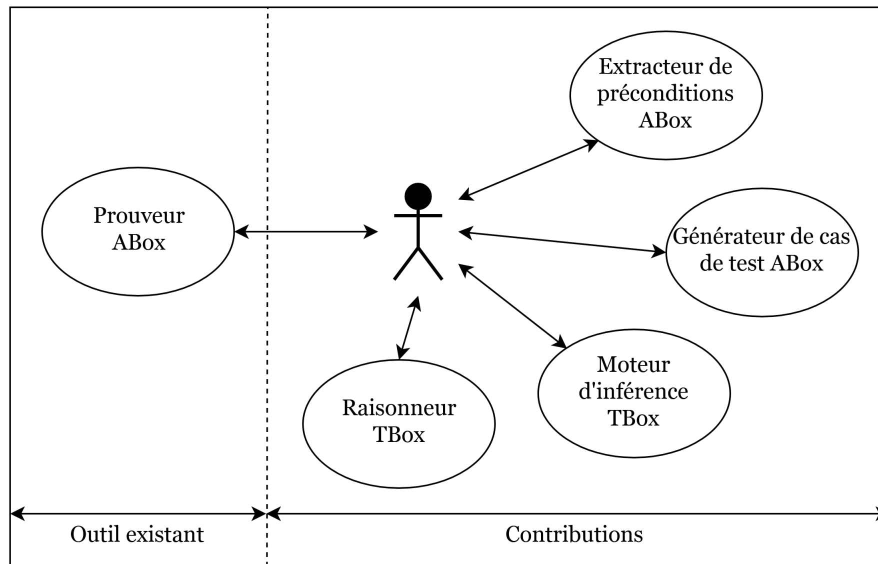


FIGURE 1.23 – L'environnement Small-t$\mathcal{ALC}$

Ces analyseurs statiques et dynamiques qui correspondent aux deux niveaux *ABox* et *TBox* forment, avec le composant de preuve déjà développé, un environnement d'assistance Small-t$\mathcal{ALC}$ complet illustré à la figure 1.23 et détaillé au chapitre 4 du manuscrit.

1.2.3 Environnement d'assistance au développement Small-t$\mathcal{ALC}$

Cette section résume les trois principales contributions de cette thèse visant à assister un développeur lors de l'écriture de ses transformations afin d'obtenir des programmes de transformation Small-t$\mathcal{ALC}$ corrects par construction. La première contribution consiste à extraire des préconditions *ABox* relatives à un code de transformation et une postcondition en simplifiant la plus faible simple précondition. La deuxième est l'inférence d'une formule *TBox* relative à une transformation exprimant des propriétés globales du graphe. La troisième contribution vise à offrir des diagnostics riches fournis par des générateurs de cas de test *ABox* et par un raisonneur *TBox* pour localiser des erreurs dans un programme. Ces contributions donnent lieu à différents outils indépendants qui ne préjugent pas d'un ordonnancement d'actions de la part du développeur.

1.2.3.1 Assistance à la construction des règles *ABox*

Étant donné qu'un développeur n'est pas toujours en mesure d'écrire une règle complète et correcte, nous lui suggérons, à partir du code de la transformation qu'il souhaite effectuer et de la postcondition qu'il désire atteindre, une précondition qui complète la règle et la rend correcte. Or, suggérer la plus faible précondition calculée par le prouveur n'est pas aisée. Cette formule, présente souvent la caractéristique de croître exponentiellement en fonction de la taille du programme d'entrée. Une des sources de complexité de cette formule est l'*aliasing* : d'une part, en Small-t$\mathcal{ALC}$, un nœud est aléatoirement affecté à une variable selon les propriétés spécifiées, et d'autre part, plusieurs variables peuvent designer le même nœud car les morphismes non-injectifs sont admis. Un graphe peut ainsi évoluer différemment selon les nœuds sélectionnés à la transformation. De ce fait, nous cherchons à simplifier la plus faible précondition d'une règle par une analyse statique fine du code *ABox* basée sur un calcul d'alias qui identifie préalablement les variables pouvant désigner à l'exécution les mêmes nœuds du graphe. Cette analyse résulte en une formule normale disjonctive dans laquelle chaque conjonction est proposée comme une précondition valide de la règle, comme le montre la figure 1.24. Le développeur pourra ensuite choisir une des préconditions ou modifier sa règle en conséquence.

Par exemple, soit la quatrième règle *setForeignKey* du programme *uml2rdb* de la section 1.1.4 qui transforme les associations en des clés étrangères des classes sources. Considérons que le développeur souhaite maintenir les tables indépendantes et qu'il remplace ainsi *setForeignKey* par la règle *association2Table* qui transforme les associations entre les classes en des tables de liaison ayant deux colonnes : la première considérée comme clé primaire est celle de la classe source, et la seconde est la clé primaire de la classe cible. Le graphe de la figure 1.25a présente un exemple possible d'un graphe source de cette règle avec une association entre deux classes ayant chacune une colonne considérée comme clé primaire. Après la transformation, le graphe cible obtenu est celui de la figure 1.25b.

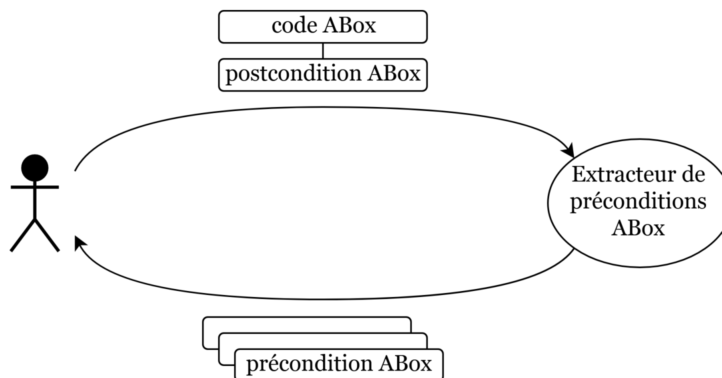


FIGURE 1.24 – Extracteur de préconditions *ABox*

(a) Exemple d'un graphe source (b) Graphe cible après la transformation

FIGURE 1.25 – Exemple d'un graphe source et de sa transformation

Pour ce faire, supposons que le développeur commence par écrire le code de la règle *association2Table* et la postcondition à atteindre comme spécifié à la figure 1.26. Une analyse statique du code et de la postcondition conduit à lui suggérer plusieurs préconditions comprenant des égalités entre les variables déclarées : $(ass, t1)$, $(ass, t2)$ et $(p1, p2)$. Afin d'interdire l'affectation d'une paire de variables parmi celles identifiées à un même nœud, le développeur peut spécifier dans sa postcondition que $ass \neq t1, ass \neq t2$ et $p1 \neq p2$. Dans ce cas, une seule conjonction sera extraite : $t1$ *pkey* $p1$ *and* $t2$ *pkey* $p2$ *and* ass $! = t2$ *and* ass $! = t1$ *and* $p1$ $! = p2$. Le développeur peut alors renforcer cette précondition par les faits ass *src* $t1$, ass *dest* $t2$ et ass : *Association* pour filtrer un nœud *ass* correct rendant les trois premières instructions de la règle applicables.

L'extracteur de préconditions *ABox* procède en deux étapes : (1) en parcourant le code et la postcondition une première fois en mode progressif pour identifier les variables possiblement équivalentes, (2) en parcourant le code et la postcondition une deuxième fois dans un mode régressif à partir de la postcondition pour extraire une précondition, transformée en forme disjonctive normale

```
delete(ass : Association);
delete(ass src t1);
delete(ass dest t2);
add(ass : Table);
add(ass cols p1);
add(ass cols p2);
add(ass pkey p1);
post: ass !src t1 and ass !dest t2 and ass : !Association & Table and t1 pkey p1
      and t2 pkey p2 and ass pkey p1 and ass cols p1 and ass cols p2;
```

FIGURE 1.26 – Code et postcondition de la règle *association2Table*

et simplifiée selon les variables possiblement équivalentes identifiées lors de la première étape. Ce processus permet finalement de suggérer au développeur plusieurs préconditions possibles dont chacune correspond à une conjonction de la précondition extraite. Le développeur choisit ensuite une des préconditions selon son intention. La règle résultante est correcte par construction.

1.2.3.2 Vérification *TBox* des règles Small-t$\mathcal{ALC}$

L'assistance à établir un triplet de Hoare correct par construction assure la véracité de la transformation des nœuds et arcs spécifiés par la règle. Or, à ce point, aucune vérification n'existe vis-à-vis d'un graphe dans sa globalité, c'est-à-dire en considérant des ensembles de nœuds et arcs ayant des propriétés particulières.

La deuxième contribution consiste à exploiter le niveau *TBox* des logiques de description afin de vérifier des propriétés globales sur le graphe en le réduisant à des ensembles de nœuds selon leurs concepts d'appartenance. En effet, ce niveau est formé d'axiomes exprimant des relations de subsomption entre concepts. La manipulation des nœuds et des arcs du graphe, c'est-à-dire le niveau *ABox*, affecte indirectement ces axiomes. Par exemple, l'ajout d'une instance a à A affecte le fait $A = empty$ et l'inclusion $A \subseteq B$ où A et B représentent les ensembles des nœuds appartenant aux concepts A et B respectivement.

Dans ce cadre, un moteur d'inférence est proposé pour étudier l'effet des règles *ABox* sur des propriétés *TBox*. A partir d'une précondition *TBox* donnée en hypothèse, le moteur d'inférence, illustré à la figure 1.27, déduit par analyse statique une postcondition *TBox* des appels *ABox*. Il s'agit d'interpréter le corps de chacune des règles appelées dans l'ordre de la séquence du point d'entrée *main*. Cette analyse considère non seulement les instructions, mais aussi les préconditions *ABox* des règles. L'effet des règles donne lieu à une formule *TBox* permettant au développeur de confronter sa postcondition *TBox* à celle inférée. Dès lors, les formules *TBox* constituent des points d'observation de la transformation ; elles s'introduisent par une clause *assert*.

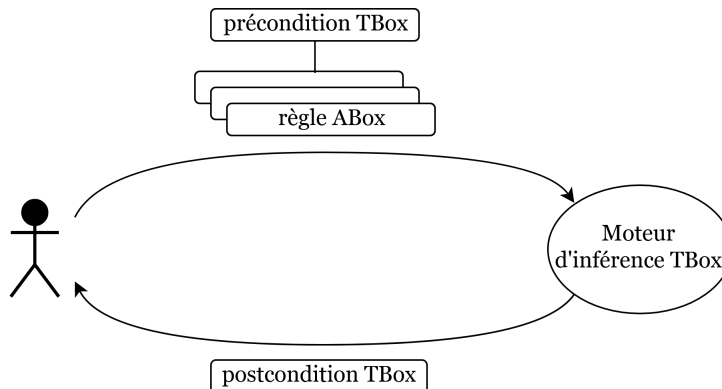


FIGURE 1.27 – Moteur d'inférence de formules *TBox*

Concrètement, considérons le programme *uml2rdb*. Après l'appel itératif des règles *class2Table* et *attribute2Column*, une vérification que tous les attributs d'une classe sont bien transformés est indispensable. Cette vérification peut être établie en exploitant les relations de subsomption des concepts *TBox*. Par exemple, les faits $(some\ cols\text{-}\ Table) = empty$, $!(Attribute = empty)$ et $Column = empty$ de la formule donnée initialement dans le point d'entrée à la figure 1.28 signifient respectivement qu'il n'existe aucun arc étiqueté *cols* sortant des nœuds de concept *Table*, qu'il existe des nœuds typés *Attribute* et qu'aucun nœud n'est de concept *Column*. Ces assertions *TBox* constituent des prémisses avant l'appel itératif de la règle *attribute2Column*. Après analyse de la règle, le raisonneur infère la formule $(some\ cols - Table) \subseteq Column\ and\ !(Column = empty)$ indiquant que les arcs *cols* sortants des nœuds typés *Table* se dirigent vers des nœuds de concept *Column* qui n'est pas vide. Le développeur pourra à ce moment confronter cette formule inférée à la formule qu'il a fourni. Cela lui permet de vérifier sa formule et de la compléter éventuellement par d'autres faits *TBox* inférés.

```
main {
 class2Table!;
 assert: (some cols- Table) = empty and!(Attribute = empty)
          and Column = empty;
 attribute2Column!;
 assert: (some cols- Table) ⊆ Column and!(Column = empty);
 setPrimaryKey!;
 setForeignKey!;
}
```

FIGURE 1.28 – Point d'entrée du programme *uml2rdb* annoté par des formules *TBox*

Étant donné que l'analyse statique *TBox* se base fondamentalement sur l'analyse *ABox*, la vérification à ce niveau abstrait est dépendante de la vérification séparée des règles *ABox*. D'où l'importance d'assurer préalablement la véracité de chacune des règles tout en renforçant les spécifications *ABox*. Les deux niveaux de vérification sont de plus complémentaires, l'un vérifie des propriétés locales du graphe et l'autre des propriétés globales.

1.2.3.3 Diagnostics *ABox* et *TBox*

Les deux premières contributions aident le développeur lors de la construction de son programme en lui proposant des préconditions d'une règle *ABox* et en inférant des formules *TBox*. Ces deux composants formels de l'environnement Small-t$\mathcal{ALC}$ sont complétés par d'autres analyseurs, statiques et dynamiques. Ils se différencient des outils précédents par le diagnostic fourni qui prend ici la forme d'un test d'une formule *ABox* via un générateur de cas de test pour examiner les corps des règles, ou l'évaluation de formules *TBox* par un raisonneur *TBox* pour une séquence de règles.

1.2.3.3.1 Le générateur de cas de test *ABox*

Comme le prouveur, cet outil nécessite un triplet complet *ABox*. Il permet d'identifier des incohérences au sein du triplet à l'aide d'un générateur de cas de test *ABox* comme le montre la figure 1.29. En pratique, l'environnement d'assistance Small-t$\mathcal{ALC}$ offre deux générateurs de cas de test : le premier mis en œuvre par une analyse statique du code semblable à celle de l'extracteur de préconditions *ABox*, et le second, basé sur une analyse dynamique, exécute le corps de la règle en permettant au développeur de définir éventuellement le graphe source.

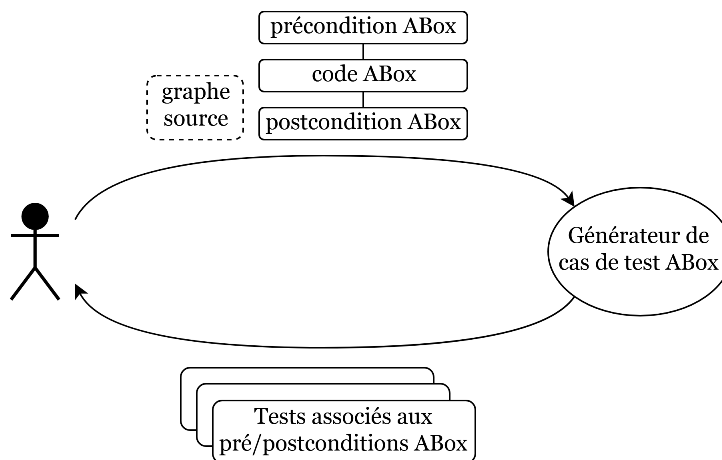


FIGURE 1.29 – Générateur de cas de test *ABox*

L'analyse statique permet d'extraire dans un mode progressif une postcondition à partir du code et de la précondition, et dans un mode régressif une précondition à partir du code et de la postcondition. En utilisant le générateur de cas de test, cet analyseur compare la précondition extraite à celle donnée et agit de même avec la postcondition extraite. Les tests qui échouent permettent d'identifier les faits de ces formules qui ne sont pas satisfaits.

L'analyse dynamique du code identifie des incohérences dans un triplet sur la base de données de test. Il s'agit d'exécuter le code de la transformation et de tester la satisfiabilité de la postcondition du graphe cible. Pour ce faire, un graphe source peut être soit fourni par l'utilisateur, soit généré automatiquement en respectant la précondition. Après exécution de la transformation sur ce graphe, un ensemble de tests correspondant à la postcondition est généré et appliqué sur le graphe cible. L'échec d'au moins un test implique que la postcondition du triplet est incorrecte. L'exécution des tests Small-t$\mathcal{ALC}$ aide le développeur à localiser les erreurs dans sa règle en interprétant les tests qui échouent.

Considérons la première règle *class2Table* du programme *uml2rdb* et supposons que le développeur oublie d'écrire la première instruction $delete(c : Class)$. L'analyseur dynamique génère tout d'abord un graphe composé d'un nœud de concept *Class* conforme à la précondition du triplet. En exécutant l'instruction $add(c : Table)$, ce nœud s'avère de concept $Class\&Table$. A partir de la postcondition $c : Table\ and\ c : !Class$, deux tests distincts sont produits qui correspondent aux faits de la formule, présentés à la figure 1.30.

```
assertExistNode(graph, c, Table);
assertNotExistNode(graph, c, Class);
```

FIGURE 1.30 – Génération des tests associés à la postcondition de la règle *class2Table*

Le premier test réussit, alors que le second échoue signifiant que le fait $c : !Class$ de la postcondition n'est pas satisfait. L'échec indique ainsi au développeur que le nœud c est toujours de concept $Class$.

Cet analyseur aide aussi à identifier les faits inconsistants dans un triplet après l'échec de la preuve en choisissant le graphe contre-exemple du prouveur. Par exemple, en définissant comme graphe d'entrée celui donné à la figure 1.22, l'analyseur dynamique est capable d'identifier que le fait $a : (\geq 2\ r\ A)$ de la règle de la figure 1.21 est inconsistant.

1.2.3.3.2 Le raisonneur *TBox*

Le raisonneur *TBox* vérifie dynamiquement la satisfiabilité de chaque formule *TBox* du programme sur un graphe en exécutant les règles appelées. Chaque fait de la formule *TBox* donnée

est ainsi traduit en un service d'inférence appliqué à une ontologie associée au graphe en un point donné. Cela permet d'indiquer la satisfaction des faits correspondants dans la formule *TBox* par rapport au graphe comme le montre la figure 1.31.

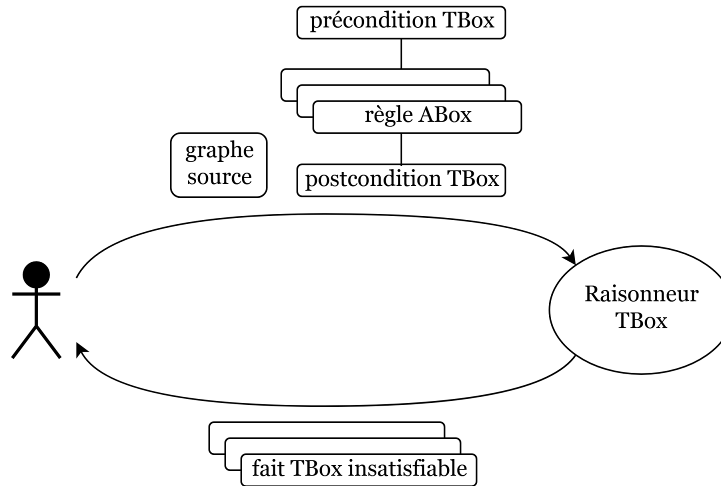


FIGURE 1.31 – Le raisonneur *TBox*

Considérant le point d'entrée du programme *uml2rdb* de la figure 1.28 et ses assertions *TBox* intermédiaires, nous supposons que le développeur souhaite maintenant caractériser le graphe cible à obtenir. Un tel scénario de mise au point du code est retranscrit à figure 1.32, matérialisé par l'ajout d'une formule *TBox* en tant qu'état final de la transformation.

Le raisonneur identifie les faits des formules *TBox* ne satisfaisant pas l'état du graphe au point d'exécution considéré, à partir d'une instance d'un graphe source fournie par le développeur. Il est

```
main {
 class2Table!;
 assert: (some cols- Table) = empty and (some attrs- Table) = Attribute;
 attribute2Column!;
 assert: (some cols- Table) = Column and (some attrs- Table) = empty;
 setPrimaryKey!;
 setForeignKey!;
 assert: Class = empty and Attribute = empty and (some pkey- Table) = Column
         and Association = empty;
}
```

FIGURE 1.32 – Point d'entrée du programme *uml2rdb* annoté par des formules *TBox*

à noter que la première formule *TBox* rencontrée est vérifiée par l'outil et n'est plus considérée comme une hypothèse de vérification. Elle correspond, comme les autres formules *TBox*, à un état de calcul attendu. Dans l'exemple de la figure 1.32, le raisonneur *TBox* indique que les faits des formules *TBox* fournies sont corrects sauf (*some pkey- Table*) = *Column* de la dernière formule qui est insatisfiable car l'ensemble des clés primaires d'une table est un sous-ensemble de ses colonnes.

Offrant une assistance à différentes étapes de construction d'un programme Small-t$\mathcal{ALC}$ correct, les outils présentés font partie, avec le composant de preuve déjà développé et un compilateur, d'un environnement Small-t$\mathcal{ALC}$ complet visant à aider un développeur à écrire, exécuter et vérifier un programme de transformation de graphes. Les trois contributions résumées dans cette section sont détaillées chacune dans les trois chapitres à venir.

1.3 Conclusion

Dans le cadre du projet CLIMT, un langage Small-t$\mathcal{ALC}$ basé sur la logique $\mathcal{ALCQI}$ a été conçu afin de vérifier formellement des transformations de graphes. A cet égard, un prouveur a été développé pour automatiser le processus de vérification d'une règle de transformation Small-t$\mathcal{ALC}$. Or, ce prouveur exige l'écriture complète de la règle et se limite à la vérification des transformations locales des nœuds et arcs d'un graphe dont les propriétés sont exprimées par des faits *ABox* de la logique.

Table 1.3 – Les outils Small-t$\mathcal{ALC}$ proposés

Niveau	Outil	Données en entrée	Données en sortie
ABox	Prouveur *ABox*	Précondition *ABox* Code *ABox* Postcondition *ABox*	Graphe contre-exemple
	Extracteur de préconditions *ABox*	Code *ABox* Postcondition *ABox*	Préconditions *ABox*
	Générateur de cas de test *ABox*	Graphe source (optionnel) Précondition *ABox* Code *ABox* Postcondition *ABox*	Tests associés à la précondition *ABox* Tests associés à la postcondition *ABox*
TBox	Moteur d'inférence TBox	Précondition *TBox* Règles *ABox*	Postcondition *TBox*
	Raisonneur *TBox*	Graphe source Précondition *TBox* Règles *ABox* Postcondition *TBox*	Faits TBox instatisfiables

Cette thèse poursuit ce premier travail et explore les capacité d'expressivité des logiques de description en réinterprétant leurs concepts vis-à-vis d'un modèle impératif de transformation à la Hoare. En mettant en exergue les niveaux *ABox* et *TBox* des formules, instructions et règles Small-t$\mathcal{ALC}$, elle propose un environnement d'assistance dédié à ce langage composé de plusieurs outils formels et semi-formels complémentaires. Les outils sont résumés au tableau 1.3 et les contributions sont détaillées dans les trois chapitres à venir.

Chapitre 2

Extraction des préconditions *ABox* des règles Small-t$\mathcal{ALC}$

Sommaire

2.1 Vérification d'une règle Small-t$\mathcal{ALC}$ . . . 56
2.2 Analyse statique d'une règle Small-t$\mathcal{ALC}$ . . . 58
2.2.1 Approche générale . . . 58
2.2.2 Calcul d'alias . . . 60
2.2.2.1 Non-déterminisme du langage Small-t$\mathcal{ALC}$ . . . 60
2.2.2.2 Variables possiblement équivalentes . . . 62
2.2.2.3 Calcul des variables non possiblement équivalentes . . . 64
2.2.2.4 Exemple de calcul d'alias . . . 66
2.2.3 Transformateur de prédicats Small-t$\mathcal{ALC}$ . . . 68
2.2.3.1 Réécriture des *wp* en formules $\mathcal{ALCQI}$ . . . 70
2.2.3.2 Simplification des $wp_{\mathcal{ALCQI}}$ . . . 72
2.2.3.2.1 Les instructions *add* et *delete* . . . 72
2.2.3.2.2 L'instruction *select* . . . 74
2.2.3.2.3 Les autres instructions . . . 76
2.2.3.3 Spécificités du transformateur . . . 77
2.2.4 Exemple . . . 78
2.3 Mise au point d'un triplet d'une règle Small-t$\mathcal{ALC}$ . . . 79
2.4 Travaux similaires . . . 85
2.5 Conclusion . . . 88

Construire un triplet de Hoare Small-t$\mathcal{ALC}$ ayant des spécifications conformes à la transformation n'est pas toujours une tâche simple pour les praticiens qui ne sont pas nécessairement familiers avec les méthodes formelles du génie logiciel. Le composant de preuve, présenté à la section 2.1, prouve la correction d'une règle en vérifiant l'implication entre la précondition du triplet et la plus faible précondition calculée, mais ne fournit pas de retours suffisants au développeur s'il doit corriger son triplet en cas d'échec. Dans ce cadre, l'étude vise à aider les développeurs à construire des règles correctes par construction. Pour ce faire, à partir d'un code de transformation et d'une postcondition à atteindre, des préconditions qui valident le triplet lui sont suggérées. Les préconditions des règles sont ainsi extraites par une analyse statique du code, décrite à la section 2.2, en exploitant les plus faibles préconditions transformées en formules $\mathcal{ALCQI}$ et en réduisant leur complexité après chaque instruction par un calcul d'alias. La formule finale, sous forme normale disjonctive, donne lieu à plusieurs préconditions possibles d'une règle où chaque conjonction rend le triplet correct. Le développeur affine cet indéterminisme en sélectionnant une conjonction en prémisse, qu'il peut reporter, tout ou en partie, en postcondition ou dans le code. La spécification du triplet donnée par les annotations est ainsi enrichie par un processus de raffinement outillé permettant d'exprimer l'intention de la règle comme le montre la section 2.3. Ceci donne lieu, par des allers-retours entre intention et spécification, à une mise au point continue et incrémentale d'un triplet toujours correct par construction.

2.1 Vérification d'une règle Small-t$\mathcal{ALC}$

Étant donné un code de transformation S, et une postcondition Q, la plus faible précondition $wp(S, Q)$ est la formule logique telle que, quel que soit le graphe g satisfaisant wp, si S appliqué à g termine, alors le graphe transformé g' satisfait Q. Autrement dit, si $P \Rightarrow wp(S, Q)$, le triplet $\{P\}S\{Q\}$ est valide. Ainsi, sachant que l'implication induit une relation d'ordre sur les prédicats, il suffit généralement de prouver que la précondition P est plus forte que $wp(S, Q)$ pour montrer que $\{P\}S\{Q\}$ est valide.

Le calcul de la plus faible précondition d'une règle se fait en mode régressif en propageant la postcondition sur les instructions. La plus faible précondition $wp(S, Q)$ de chaque instruction vis-à-vis d'une postcondition Q est donnée à la figure 2.1. Ce calcul est classique pour les instructions atomiques, la sélection et la séquence. Par contre, le calcul wp d'une boucle à l'égard d'une postcondition Q est plus complexe : l'instruction *while c do s*, sémantiquement équivalente à $if\ c\ then\ \{s; while\ c\ do\ s\}\ else\ skip$, $wp(while\ c\ do\ s, Q)$ peut s'approximer par $(c \Rightarrow wp(c, wp(while\ c\ do\ s, Q))) \wedge\ (\neg c \Rightarrow Q)$. Cette équation récursive étant difficile à calculer, les boucles sont annotées par des formules devant être correctes avant et après chaque itération, appelées invariants. La plus faible précondition d'une boucle est ainsi son invariant qui est vérifié par une fonction appelée condition de vérification vc. Comme explicité à la figure 2.2, cette fonction résulte

$$
\begin{array}{l}
wp(add\ (i:C),Q) = Q[C+i\backslash C] \\
wp(delete\ (i:C),Q) = Q[C-i\backslash C] \\
wp(add\ (i\ r\ j),Q) = Q[r+(i,j)\backslash r] \\
wp(delete\ (i\ r\ j),Q) = Q[r-(i,j)\backslash r] \\
wp(select\ v\ with\ F,Q) = \forall v(F \Rightarrow Q) \\
wp(s1;s2,\ Q) = wp(s1,wp(s2,\ Q)) \\
wp(if\ c\ then\ s1\ else\ s2,Q) = (c \Rightarrow wp(s1,Q)) \wedge\ (\neg c \Rightarrow wp(s2,Q)) \\
wp(\{inv\}\ while\ c\ do\ s,Q) = inv
\end{array}
$$

FIGURE 2.1 – Les plus faibles préconditions des instructions Small-t$\mathcal{ALC}$

$$
\begin{array}{l}
vc(add\ (i:C),Q) = \top \\
vc(delete\ (i:C),Q) = \top \\
vc(add\ (i\ r\ j),Q) = \top \\
vc(delete\ (i\ r\ j),Q) = \top \\
vc(select\ v\ with\ F,Q) = \top \\
vc(s1;s2,\ Q) = vc(s1,wp(s2,\ Q)) \wedge\ vc(s2,Q) \\
vc(if\ c\ then\ s1\ else\ s2,Q) = vc(s1,Q) \wedge\ vc(s2,Q) \\
vc(\{inv\}\ while\ c\ do\ s,Q) = (inv \wedge\ \neg c \Rightarrow Q) \wedge\ (inv \wedge\ c \Rightarrow wp(s,inv)) \wedge\ vc(s,inv)
\end{array}
$$

FIGURE 2.2 – Les conditions de vérification des instructions Small-t$\mathcal{ALC}$

en une formule valide ($\top$) pour toutes les instructions atomiques, des formules classiques pour la sélection et la séquence, et une formule plus complexe pour l'instruction *while* où l'invariant *inv* doit être vérifié. Dès lors, la correction d'un triplet $\{P\}S\{Q\}$ découle de la véracité de la formule $vc(S,Q) \wedge\ (P \Rightarrow wp(S,Q))$ [25]. Le composant de preuve Small-t$\mathcal{ALC}$ automatise ce processus de vérification en prouvant la validité de cette formule et en renvoyant un graphe contre-exemple dans le cas où la formule n'est pas satisfiable.

Remarquons que l'application d'un calcul de plus faible précondition ne code pas une formule $\mathcal{ALCQI}$ mais une formule qui est : (1) non close par substitution, (2) et essentiellement universellement quantifiée. Pour le premier cas, citons par exemple, $wp(add(i : C),(x : C)) = (x : C)[C+i/C] = x : (C+i)$ et pour le second $wp(select\ a\ with\ a : A,(a : A)) = \forall a(a : A \Rightarrow a : A)$. Le composant de preuve Small-t$\mathcal{ALC}$ effectue ainsi un calcul interne pour manipuler ces *wp* en propageant les substitutions dans les formules et en supprimant les quantificateurs universels. Ce calcul est complexe et enfoui au sein du prouveur dès lors qu'il ne s'attache qu'à la vérification du triplet. La preuve est ensuite mise en œuvre par la méthode des tableaux sémantiques à partir des formules transformées. Dès lors, le composant de preuve n'est pas en mesure d'expliciter la plus faible précondition d'une règle à un développeur Small-t$\mathcal{ALC}$.

2.2 Analyse statique d'une règle Small-t$\mathcal{ALC}$

Afin d'aider les développeurs à bâtir des règles correctes par construction, les formules *wp* et *vc* du prouveur sont exploitées dans l'objectif de suggérer, via une analyse statique, des préconditions valides à leur code et postcondition. Le processus général de cette analyse [86] est décrit à la sous-section 2.2.1. Il s'agit d'effectuer un calcul d'alias, présenté à la sous-section 2.2.2, permettant de parcourir le code et la postcondition une première fois en mode progressif afin d'identifier les variables ne pouvant jamais désigner le même nœud du graphe, ensuite, d'extraire des préconditions via un transformateur de prédicats qui procède en mode régressif en se basant sur les résultats du calcul d'alias comme détaillé à la sous-section 2.2.3. Un exemple d'extraction de préconditions relatives à une séquence d'instructions et une postcondition est donné à la sous-section 2.2.4.

2.2.1 Approche générale

Le calcul de la plus faible précondition du prouveur aboutit à une précondition non close par substitution et essentiellement universellement quantifiée. Ne respectant pas la syntaxe Small-t$\mathcal{ALC}$, cette *wp* est peu lisible et donc non exploitable par le développeur. De ce fait, il est nécessaire de la transformer en une formule $\mathcal{ALCQI}$ notée par la suite $wp_{\mathcal{ALCQI}}$.

Dès lors, le processus de l'analyse statique consiste en deux étapes permettant d'extraire des préconditions $\mathcal{ALCQI}$ valides correspondantes à un code et une postcondition :

(1) Un calcul d'alias qui effectue un premier parcours du code et de la postcondition en mode progressif afin d'identifier les variables de la règle ne pouvant jamais désigner le même nœud à l'exécution.

(2) Un transformateur de prédicats qui procède en mode régressif afin de calculer $wp_{\mathcal{ALCQI}}$, et de la simplifier en se basant sur les variables identifiées non équivalentes lors de la première étape.

Le résultat de l'analyse se présente sous la forme d'une précondition $\mathcal{ALCQI}$ en forme normale disjonctive où chaque conjonction est une précondition valide. En choisissant une des propositions, le développeur affine l'intention de la règle tout en assurant sa correction.

Pour illustrer la démarche, considérons l'exemple de la plus faible précondition de l'instruction $add(i\ r\ j)$ à l'égard de la postcondition $x : (\leq 3\ r\ C)$:

$$wp\big(add(i\ r\ j),\ x : (\leq 3\ r\ C)\big) = x : \big(\leq 3\ (r + (i,j))\ C\big)$$

La formule $wp_{\mathcal{ALCQI}}$ correspondante conduit, comme détaillé ultérieurement, à celle de la figure 2.3 comportant plusieurs conjonctions :

$$\begin{aligned} wp_{\mathcal{ALCQI}}\big(add(i\ r\ j),\ x : (\leq 3\ r\ C)\big) = &\ (x = i\ \wedge\ j : C\ \wedge\ i\ !r\ j\ \wedge\ x : (\leq 2\ r\ C)) \\ &\vee (x\ != i\ \wedge\ x : (\leq 3\ r\ C)) \\ &\vee (j : !C \wedge\ x : (\leq 3\ r\ C)) \end{aligned}$$

$$\lor\ (i\ r\ j \land\ x : (\leq 3\ r\ C))$$
$$\lor\ (x : (\leq 2\ r\ C))$$

Cette formule croît exponentiellement en fonction de la taille du programme et devient complexe. Toutefois, mise en forme normale disjonctive, elle peut se réduire si la validité de l'un des faits qui

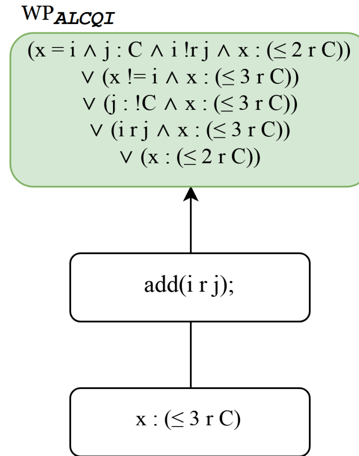


FIGURE 2.3 – $wp_{\mathcal{ALCQI}}\big(add(i\ r\ j), x : (\leq 3\ r\ C)\big)$

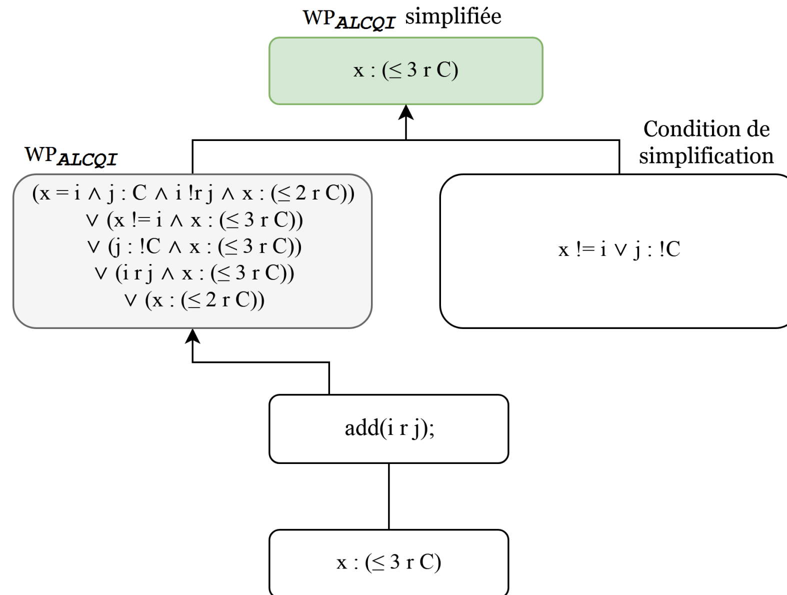


FIGURE 2.4 – $wp_{\mathcal{ALCQI}}(add(i\ r\ j), x : (\leq 3\ r\ C))$ simplifiée

la compose peut être établie : s'il est prouvé que $x \; != i$ ou $j : !C$, la formule extraite se réduit à $x : (\leq 3 \; r \; C)$ comme le montre la figure 2.4, en s'appuyant sur l'implication logique :

$$\Big(wp_{\mathcal{ALCQI}}\big(add(i \; r \; j), x : (\leq 3 \; r \; C)\big) \wedge \; \big(x \; != i \; \vee \; j : !C\big)\Big) \implies x : (\leq 3 \; r \; C)$$

Dès lors qu'ils sont explicitement mentionnés par le développeur dans la postcondition de la règle, les faits $x \, != i$ et $j : !C$ simplifient $wp_{\mathcal{ALCQI}}(add(i \; r \; j), x : (\leq 3 \; r \; C))$. Dans le cas contraire, $x \, != i$ peut être inféré statiquement par un calcul d'alias qui s'assure que x et i ne pourront jamais pointer à l'exécution le même nœud. De telles variables sont alors identifiées non possiblement équivalentes. A noter que le fait $i \; r \; j$ ne peut pas être démontré valide car il est directement stipulé par l'instruction $add(i \; r \; j)$.

2.2.2 Calcul d'alias

Le calcul d'alias a été introduit par Bertrand Meyer en 2011 [91] comme point de départ d'une solution qui s'intéresse généralement à la modélisation de l'effet des actions en vérifiant la préservation des propriétés non modifiées par ces actions (*frame problem*). En effet, le problème d'*aliasing* constitue actuellement le principal obstacle qui empêche d'établir une preuve complète des programmes séquentiels et concurrents. Le principe du calcul d'alias est de déterminer si une même donnée est référencée par deux variables distinctes. Ce calcul peut être exploité dans des systèmes axiomatiques comme c'est le cas pour Small-t$\mathcal{ALC}$ qui autorise le morphisme non-injectif. En effet, plusieurs variables d'une règle, dites possiblement équivalentes, peuvent partager un même nœud du graphe. Ceci impacte le résultat de la transformation et rend souvent les spécifications incohérentes avec le code de la transformation comme le montre la section 2.2.2.1. L'identification des variables possiblement équivalentes d'une règle Small-t$\mathcal{ALC}$, définie à la section 2.2.2.2, est établie par un calcul d'alias présenté à la section 2.2.2.3. Un exemple de ce calcul est donné à la section 2.2.2.4. L'objectif est de simplifier les plus faibles préconditions $\mathcal{ALCQI}$ des instructions.

2.2.2.1 Non-déterminisme du langage Small-t$\mathcal{ALC}$

En Small-t$\mathcal{ALC}$, le morphisme non-injectif est autorisé, d'où plusieurs variables d'une règle peuvent designer à l'exécution le même nœud du graphe. Il est à la source d'un indéterminisme lié à la sélection d'un graphe à transformer. Considérons comme premier exemple le triplet de la figure 2.5 qui sélectionne, par l'intermédiaire de la précondition, un nœud de concept C connecté par r à un nœud y. Généralement, la suppression du nœud x du concept C et l'ajout de y à C résulte en une postcondition précisant que $x : !C$ et $y : C$. Sauf que le filtrage d'un modèle satisfaisant $x : C$ et $x \; r \; y$ dans un graphe n'interdit pas la sélection d'un seul nœud ayant un arc r bouclant. Dès lors x et y désignent ce même nœud comme illustré à la figure 2.6. Dans ce cas, la suppression du nœud filtré de concept C par *delete(x : C)*, puis son ajout de nouveau à C par *add(y : C)*, conduit

```
pre: x : C and x r y;
delete(x : C);
add(y : C);
post: x :!C and x r y and y : C;
```

FIGURE 2.5 – Triplet incorrect lorsque x a pour alias y

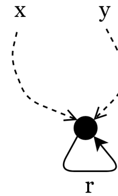


FIGURE 2.6 – Modèle de graphe satisfaisant la précondition $x : C$ *and* $x\ r\ y$

à $x : C$ et $y : C$. D'où la postcondition de la figure 2.5 ne sera pas satisfaite. Pour rendre le triplet valide, il suffit de spécifier $x\ \ != y$ dans la précondition.

Un autre exemple est celui de la figure 2.7. Ce triplet qui supprime un arc r entre deux nœuds a et b et en ajoute un autre entre a et c semble correct. Toutefois, la sélection d'un nœud c de concept C n'empêche pas que ce nœud soit le même que celui indiqué par a ou b. En effet, la spécification qu'un nœud appartient à un concept C ne signifie pas son appartenance à C exclusivement ; un nœud peut appartenir à un ou plusieurs concepts. Citons le cas où b appartient à B et à C simultanément. Le nœud affecté à c peut aussi être atteint par b comme le montre le graphe de la figure 2.8. Lorsque

```
pre: a : A and b : B and a r b;
select c with c : C;
delete(a r b);
add(a r c);
post: a : A and b : B and a!r b and c : C and a r c;
```

FIGURE 2.7 – Triplet incorrect lorsque b a pour alias c

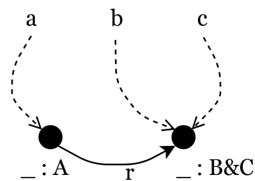


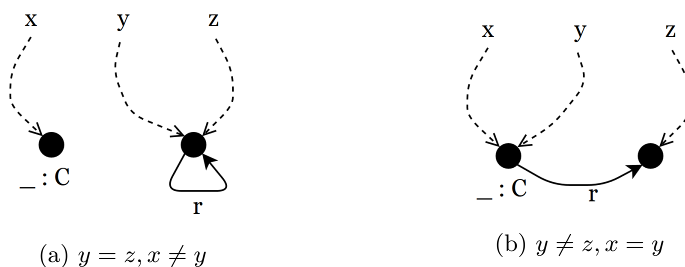
FIGURE 2.8 – Modèle de graphe où b et c référencent le même nœud

le nœud c a pour alias le nœud b, les instructions $add(a\ r\ c)$ et $delete(a\ r\ b)$ sont respectivement équivalentes à $add(a\ r\ b)$ et $delete(a\ r\ c)$, ce qui conduit à l'état de calcul $a\ r\ b$ et $a\ r\ c$, qui contredit la postcondition $a\ !r\ b$ de la figure 2.7.

Afin d'éviter ce cas, il faut spécifier que les variables b et c sont distinctes en ajoutant $b\ != c$ dans la formule de l'instruction *select* de la figure 2.7.

2.2.2.2 Variables possiblement équivalentes

Le filtrage effectué par une précondition Small-t$\mathcal{ALC}$ ou une instruction *select* attribue aléatoirement des nœuds du graphe aux variables. A noter que les autres instructions atomiques *add* et *delete* peuvent également assigner aléatoirement des nœuds aux variables qu'elles manipulent dans le cas où ces variables n'ont pas été déjà référencées par la règle. Une formule Small-t$\mathcal{ALC}$ peut ainsi représenter plusieurs modèles. Par exemple, la figure 2.9 illustre deux modèles parmi d'autres satisfiant la formule $x : C\ and\ y\ r\ z$. Les deux variables y et z sont affectées au même nœud dans le modèle de la figure 2.9a. A la figure 2.9b, elles désignent deux nœuds différents, mais x et y partagent le même nœud. On dit que les variables y et z de la figure 2.9a sont possiblement équivalentes. Il en est de même pour x et y de la figure 2.9b. Des variables non possiblement équivalentes sont telles qu'elles ne peuvent jamais désigner un même nœud du graphe.



(a) $y = z, x \neq y$

(b) $y \neq z, x = y$

FIGURE 2.9 – Exemple de deux modèles satisfaisant la formule $x : C\ and\ y\ r\ z$

Dans ce cadre, le calcul d'alias est exploité pour identifier dans une règle Small-t$\mathcal{ALC}$ les variables ne pouvant jamais référencer dynamiquement les même nœuds. Cela permet de simplifier les plus faibles préconditions $\mathcal{ALCQI}$ comme expliqué dans le reste du chapitre.

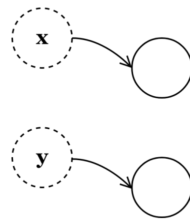
Deux variables x et y sont non possiblement équivalentes si l'une des conditions suivantes est satisfaite :

1. $x \neq y$
2. $\exists C \mid x : C \wedge\ y : !C$
3. $\exists r\ \exists z \mid x\ r\ z \wedge\ y\ !r\ z$
4. $\exists r\ \exists z \mid z\ r\ x \wedge\ z\ !r\ y$

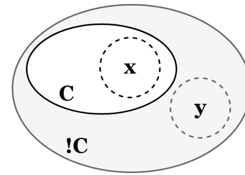
Ces conditions, illustrées à la figure 2.10, garantissent que les nœuds référencés par x et y ne peuvent être confondus lors de la réécriture. La première, $x \neq y$, stipule que les instances x et y ont forcément une interprétation différente ; x et y sont alors dites séparées dans le jargon des logiques de description. Pour un concept C donné, C et $!C$ sont incompatibles : cette seconde contrainte exprime que l'ensemble des individus de $!C$ est inclus dans le complémentaire de celui de C. Il en est de même pour les contraintes portant sur un rôle : des nœuds connectés respectivement par r et $!r$ référencent des ensembles disjoints d'individus. Ce résultat intuitif est corroboré par la sémantique associée aux descriptions des concepts et des rôles : rappelons que les concepts sont interprétés comme des sous-ensembles d'un domaine d'interprétation $\Delta^{\mathcal{I}}$ et les rôles comme des sous-ensembles du produit $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$. Dans cette interprétation où $\mathcal{I} = (\Delta^{\mathcal{I}}, .^{\mathcal{I}})$ est la donnée d'un ensemble $\Delta^{\mathcal{I}}$ et d'une fonction d'interprétation $.^{\mathcal{I}}$ qui fait correspondre à un concept un sous-ensemble de $\Delta^{\mathcal{I}}$ et à un rôle un sous-ensemble de $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$, les deux équations suivantes sont satisfaites :

— $(!C)^{\mathcal{I}} = \Delta^{\mathcal{I}} \backslash C^{\mathcal{I}}$
— $(!r)^{\mathcal{I}} = \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} \backslash r^{\mathcal{I}}$

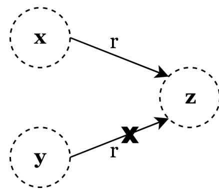
Soit la relation de possible équivalence entre deux variables x et y notée $\simeq$. Cette relation est réflexive, symétrique et non transitive. A l'inverse, nous notons par $x \not\simeq y$ la situation où x et y sont non possiblement équivalentes. La relation $\not\simeq$ est symétrique et non réflexive. Une analyse statique de code pouvant certifier que $x \not\simeq y$ certifie dynamiquement que $x \neq y$, d'où $x \not\simeq y \Rightarrow x \neq y$. Rien



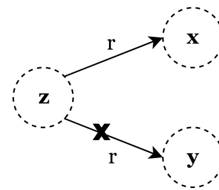
(a) $x \neq y$



(b) $\exists C \mid x : C \wedge \ y : !C$



(c) $\exists r \ \exists z \mid x \ r \ z \wedge \ y \ !r \ z$



(d) $\exists r \ \exists z \mid z \ r \ x \wedge \ z \ !r \ y$

FIGURE 2.10 – Conditions garantissant que x et y sont non possiblement équivalentes

ne peut être conclu quant à l'égalité de deux nœuds x et y lorsqu'ils sont possiblement équivalents : $x \simeq y \Rightarrow x = y \vee x \neq y$.

2.2.2.3 Calcul des variables non possiblement équivalentes

Le calcul d'alias d'une règle se fait en parcourant ses instructions dans un mode progressif commençant par la première instruction jusqu'à la postcondition. Plusieurs ensembles sont bâtis lors de ce calcul :

— Pour chaque variable v de la règle, un ensemble $E(v)$ comprenant l'ensemble des variables possiblement équivalentes à v.

— Pour tout concept C (respectivement $!C$) mis en jeu par une variable v de la règle, un ensemble C (respectivement $!C$) mémorisant les variables v typées par C (respectivement $!C$).

— Pour tout rôle r (respectivement $!r$) mis en jeu par deux variables v_1 et v_2, un ensemble r (respectivement $!r$) mémorisant les couples (v_1, v_2) de variables typées par r (respectivement $!r$).

Dès lors, après chaque instruction, une première étape consiste à mettre à jour les ensembles d'appartenance C, $!C$, r et $!r$ de la règle comme suit :

— Après l'instruction $add(i : C)$ (respectivement $delete(i : C)$), l'instance i est rajoutée à l'ensemble C (respectivement $!C$), soit $C \supseteq \{..., i\}$ (respectivement $!C \supseteq \{..., i\}$).

— Après $add(i\ r\ j)$ (respectivement $delete(i\ r\ j)$), la paire d'instances (i, j) est rajoutée à l'ensemble r (respectivement $!r$).

— Après *select v with F*, les ensembles C, $!C$, r et $!r$ seront modifiés selon les faits de la formule F manipulant les variables v, conformément aux mises à jour mentionnées ci-dessus.

En plus de l'ajout des variables manipulées par chaque instruction aux ensembles correspondants, d'autres variables de ces ensembles peuvent être éventuellement affectées via les instructions *add* et *delete* qui établissent une interdépendance des variables liée à l'*aliasing*. Tout nœud du graphe peut être attribué à une variable, y compris celui déjà assigné à une variable préalablement examinée par l'analyse ; ceci remet en cause l'ensemble d'appartenance de cette dernière. Par exemple, si x est une variable déjà déclarée de concept C par la règle, l'instruction $delete(y : C)$, affecte C si x et y partagent le même nœud. De ce fait, outre l'ajout de y à l'ensemble $!C$, x sera supprimée de l'ensemble C.

La deuxième étape consiste à mettre à jour chaque ensemble $E(v)$ associé à une variable v en se basant sur les quatre conditions du calcul d'alias :

— une variable v_1 appartenant à C est considérée non possiblement équivalente à toute variable v_2 appartenant à $!C$,

— une variable v_1 du couple (v_1, v) (respectivement (v, v_1)) appartenant à r est non possiblement équivalente à v_2 de la paire (v_2, v) (respectivement (v, v_2)) appartenant à $!r$.

```
Pour chaque instruction s de la séquence S de la règle :
   switch(s) {

     case(add(vn : C)) :
       C ⊇ {vn};
       Si vn ∉ V alors
          !C = ∅;
          V = V ∪ {vn};
          Créer E(vn);
          E(vn) = V;
          Ajouter vn à tous les E(vi);
       Mettre à jour tous les E(vi) selon les ensembles C, !C, r et !r.

     case(delete(vn : C)) :
       !C ⊇ {vn};
       Si vn ∉ V alors
          C = ∅;
          V = V ∪ {vn};
          Créer E(vn);
          E(vn) = V;
          Ajouter vn à tous les E(vi);
       Mettre à jour tous les E(vi) selon les ensembles C, !C, r et !r.

     case(add(vn r vm)) :
       r ⊇ {(vn,vm)};
       Si vn ∉ V et vm ∈ V alors
          Supprimer de l'ensemble !r toutes les paires de la forme (x, vm);
          V = V ∪ {vn};
          Créer E(vn);
          E(vn) = V;
          Ajouter vn à tous les E(vi);
       Si vn ∈ V et vm ∉ V alors
          Supprimer de l'ensemble !r toutes les paires de la forme (vn, x);
          V = V ∪ {vm};
          Créer E(vm);
          E(vm) = V;
          Ajouter vm à tous les E(vi);
       Si vn ∉ V et vm ∉ V alors
          !r = ∅;
          V = V ∪ {vn,vm};
          E(vn) = V;
          E(vm) = V;
          Ajouter vn,vm à tous les E(vi);
       Mettre à jour tous les E(vi) selon les ensembles C, !C, r et !r.
```

```
      case(delete(vn r vm)) :
        !r ⊇ {(vn,vm)};
        Si vn ∉ V et vm ∈ V alors
           Supprimer de l'ensemble r toutes les paires de la forme (x, vm);
           V = V ∪ {vn};
           E(vn) = V;
           Ajouter vn à tous les E(vi);
        Si vn ∈ V et vm ∉ V alors
           Supprimer de l'ensemble r toutes les paires de la forme (vn, x);
           V = V ∪ {vm};
           E(vm) = V;
           Ajouter vm à tous les E(vi);
        Si vn ∉ V et vm ∉ V alors
           r = ∅;
           V = V ∪ {vn,vm};
           E(vn) = V;
           E(vm) = V;
           Ajouter vn,vm à tous les E(vi);
        Mettre à jour tous les E(vi) selon les ensembles C, !C, r et !r.

      case(select V with F) :
        Pour chaque vn de V :
           V = V ∪ {vn};
           E(vn) = V;
           Ajouter vn à tous les E(vi);
        Ajouter vn aux ensembles C, !C, r et !r selon les faits de F;
        Mettre à jour tous les E(vi) selon les ensembles C, !C, r et !r.
   }
Mettre à jour les ensembles C, !C, r et !r selon les faits de la postcondition;
Mettre à jour tous les E(vi) selon les ensembles C, !C, r et !r.
```

FIGURE 2.11 – Algorithme du calcul d'alias à partir de la séquence d'instructions et la postcondition d'une règle

Dans ces deux situations, v_1 sera supprimée de l'ensemble $E(v_2)$ et v_2 supprimée de l'ensemble $E(v_1)$. L'algorithme illustrant ce calcul est donné à la figure 2.11. A noter que l'ensemble V mentionné dénote les variables de la règle déjà examinées au regard du mode progressif d'analyse.

2.2.2.4 Exemple de calcul d'alias

Pour éclaircir le processus, considérons le code de transformation et la postcondition de la figure 2.12. Afin de montrer le calcul d'alias à chaque étape, les ensembles E, C, $!C$, r et $!r$ sont exposés après chaque instruction.

Instruction	Contraintes
`delete(x:C);`	$!C \supseteq \{x\}$ $E(x) = \{x\}$
`add(y:C);`	$!C \supseteq \{\not{x}\}$ $C \supseteq \{y\}$ $E(x) = \{x, y\}$ $E(y) = \{y, x\}$
`select z with z : !C;`	$!C \supseteq \{z\}$ $C \supseteq \{y\}$ $E(x) = \{x, y, z\}$ $E(y) = \{y, x\}$ $E(z) = \{z, x\}$
`add(y r z);`	$!C \supseteq \{z\}$ $C \supseteq \{y\}$ $r \supseteq \{(y, z)\}$ $E(x) = \{x, y, z\}$ $E(y) = \{y, x\}$ $E(z) = \{z, x\}$
`post : x : !C and y : C and y r z;`	$!C \supseteq \{z, x\}$ $C \supseteq \{y\}$ $r \supseteq \{(y, z)\}$ $E(x) = \{x, z\}$ $E(y) = \{y\}$ $E(z) = \{z, x\}$

FIGURE 2.12 – Calcul d'alias sur un exemple

i) *delete(x : C)*

Après la première instruction $delete(x : C)$, la variable x est rajoutée à l'ensemble $!C$, puis l'ensemble $E(x)$ associé à x est défini en y insérant x, l'unique variable du code à ce point d'analyse.

ii) *add(y : C)*

L'instruction suivante $add(y : C)$ manipule une nouvelle variable de la règle. Cette variable peut être affectée à l'un des nœuds désignés par les autres variables déjà analysées, notamment x. De ce fait, x peut éventuellement être ajoutée à C, donc $x \in !C$ ne sera plus valide. La première étape supprime de fait la variable x de l'ensemble $!C$ et ajoute la variable y à l'ensemble C. La deuxième

étape identifie les variables non possiblement équivalentes. La variable x n'appartenant à aucun ensemble, x et y sont possiblement équivalentes, d'où la mise à jour de $E(x)$ en lui rajoutant y et la construction de $E(y)$ en lui rajoutant x et y.

iii) *select z with z : !C*

La troisième instruction *select* affecte la variable z à un nœud de concept $!C$. La première étape consiste à rajouter z à l'ensemble $!C$. La deuxième étape révèle que y et z sont non possiblement équivalentes en se basant sur la condition (2) de la sous-section précédente : $y : C \wedge \; z : !C$. Il s'en suit la mise à jour des trois ensembles $E(x), E(y)$ et $E(z)$ comme indiqué à la figure 2.12 à ce point d'analyse.

iv) *add(y r z)*

La dernière instruction connecte par r les deux variables déjà déclarées y et z. La paire (y, z) est donc rajoutée à l'ensemble r et les ensembles $E(x), E(y)$ et $E(z)$ restent inchangés selon les conditions définies.

v) Postcondition

Finalement, les ensembles $E(x), E(y)$ et $E(z)$ sont mis à jour selon les faits de la postcondition : en rajoutant x à $!C$, les variables x et y sont non équivalentes. Dès lors, après le code de transformation et cette postcondition, $x \simeq z$, $x \not\simeq y$ et $z \not\simeq y$.

Remarquons que la première étape de l'analyse statique considère implicitement les éléments de la $TBox$; elle reconstruit les sous-ensembles du domaine d'interprétation Δ^I (les concepts et leur complément) et les sous-ensembles du produit $\Delta^I \times \Delta^I$ (les rôles et leur complément) en énumérant respectivement les individus et les couples d'individus.

La relation $\not\simeq$ de non possible équivalence permet de renforcer une précondition car elle implique une inégalité des variables mises en jeu. Ces résultats sont exploités afin de simplifier les plus faibles préconditions $\mathcal{ALCQI}$ des instructions.

2.2.3 Transformateur de prédicats Small-t$\mathcal{ALC}$

Au niveau d'une règle, le transformateur de prédicats consiste d'abord à transformer la postcondition Q en forme normale disjonctive, c'est-à-dire $Q = \vee Q_i$ où $Q_i = \wedge \, q_j$. Ensuite, un calcul de préconditions est établi en considérant la séquence d'instructions S vis-à-vis de chaque conjonction Q_i comme l'illustre la figure 2.13. Ainsi, la précondition $\mathcal{ALCQI}$ extraite assure la validité de la règle du fait que $wp(S, Q_1) \vee wp(S, Q_2) \Rightarrow wp(S, Q_1 \vee Q_2)$.

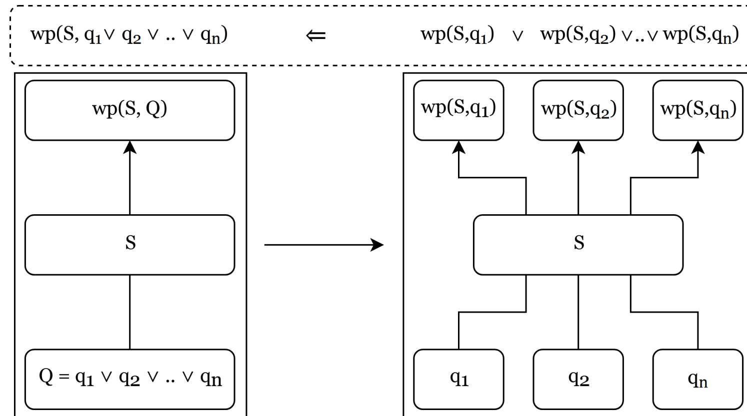


FIGURE 2.13 – Extraction des préconditions $\mathcal{ALCQI}$ en transformant la postcondition en forme normale disjonctive

La précondition de la règle à l'égard de chaque conjonction Q_i est calculée dans un parcours régressif de S en exploitant les plus faibles préconditions transformées en $\mathcal{ALCQI}$ comme décrit à la sous-section 2.2.3.1 et en les simplifiant comme détaillé à la sous-section 2.2.3.2. Ceci est illustré, pour une instruction, à la figure 2.14. Les boucles exigent en plus la vérification des invariants par les conditions de vérification. Si ces dernières sont satisfaites, alors la précondition P extraite sous forme normale disjonctive rend la règle $\{P\}S\{Q\}$ concernée correcte du fait qu'elle est telle que $P \Rightarrow wp(S, Q)$ comme le mentionne la sous-section 2.2.3.3.

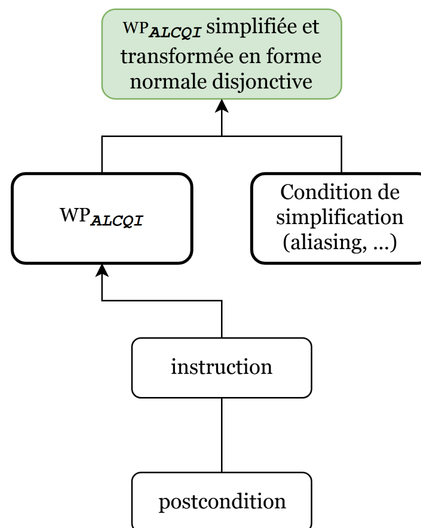


FIGURE 2.14 – Simplification de la plus faible précondition $\mathcal{ALCQI}$ en exploitant les variables non possiblement équivalentes

2.2.3.1 Réécriture des wp en formules $\mathcal{ALCQI}$

Afin d'obtenir des préconditions $\mathcal{ALCQI}$ lisibles et exploitables par les développeurs, il convient d'éliminer les substitutions et les quantificateurs des wp. Par exemple, en considérant $wp(add(i : C), (x : C))$ et par injection du terme $C+i$ dans l'état obtenu, $x : (C+i)$ se réécrit en $x : C \vee x = i$. Le même raisonnement s'applique aux substitutions $C - i, r + (i, j)$ et $r - (i, j)$. Par exemple, $(x : C)[C - i/C] = x : (C - i)$ se réécrira en $x : C \ \wedge \ x \neq i$. Remarquons l'émergence des cas $\mathcal{ALCQI}$ possibles suite à la substitution.

Les tableaux 2.1 et 2.2 donnent les formules $\mathcal{ALCQI}$ qui expriment, en éliminant les substitutions pour les concepts et les rôles, la plus faible précondition $wp_{\mathcal{ALCQI}}$ correspondante à chaque instruction atomique vis-à-vis des faits Small-t$\mathcal{ALC}$ pouvant être affectés par l'instruction ; ces faits sont intitulés «*Fait identifié* » dans la deuxième colonne des tableaux. Par exemple, l'ajout d'une instance à un concept C par l'instruction $add(i : \underline{C})$ ne considère que les faits de Q qui manipulent C, par exemple $x : \underline{C}, x : !\underline{C}$ et $x : (\leq n \ r \ \underline{C})$.

Pour éclaircir le cas des restrictions, considérons la troisième ligne de la table 2.2 concernant la plus faible précondition de l'instruction $add(i \ r \ j)$ par rapport au fait $x : (\leq n \ r \ C)$ qui résulte en $x : (\leq n \ (r + (i, j)) \ C)$. L'ajout de la paire (i, j) à l'ensemble des paires r décrémente le nombre de restrictions pour $x : (\leq n \ r \ C)$ si x et i désignent le même nœud, si j est de concept C ou si la paire (i, j) n'appartient pas préalablement à r, ce qui se code en Small-t$\mathcal{ALC}$ respectivement par $x = i$, $j : C$ et $i \ !r \ j$. Cette situation se traduit par la première conjonction de la formule $\mathcal{ALCQI}$ donnée, soit $x = i \ \wedge \ j : C \ \wedge \ i \ !r \ j \ \wedge \ x : (\leq (n-1) \ r \ C)$. Le nombre de restrictions ne peut rester constant que si la négation de l'une des conditions citées au moins est indiquée : $x \ != i$,

TABLE 2.1 – Élimination des substitutions des *wp* de *add(i : C)* et *delete(i : C)*

Instruction	Fait identifié	wp	$wp_{\mathcal{ALCQI}}$
$add(i : C)$	$x : C$	$x : (C+i)$	$x : C \vee x = i$
	$x : !C$	$x : !(C+i)$	$x : !C \wedge \ x \ !{=}i$
	$x : (\leq n \ r \ C)$	$x : (\leq n \ r \ (C+i))$	$(x \ r \ i \ \wedge \ i : !C \wedge \ x : (\leq (n-1) \ r \ C))$ $\vee \ (x \ !r \ i \wedge \ x : (\leq n \ r \ C))$ $\vee \ (i : C \wedge \ x : (\leq n \ r \ C))$ $\vee \ (x : (\leq (n-1) \ r \ C))$
$delete(i : C)$	$x : !C$	$x : !(C-i)$	$x : !C \vee x = i$
	$x : C$	$x : (C-i)$	$x : C \wedge \ x \ != i$
	$x : (\geq n \ r \ C)$	$x : (\geq n \ r \ (C-i))$	$(x \ r \ i \ \wedge \ i : C \wedge \ x : (\geq n+1 \ r \ C))$ $\vee \ (x \ !r \ i \wedge \ x : (\geq n \ r \ C))$ $\vee \ (i : !C \wedge \ x : (\geq n \ r \ C))$ $\vee \ (x : (\geq n+1 \ r \ C))$

Table 2.2 – Élimination des substitutions des wp de *add(i r j)* et *delete (i r j)*

Instruction	Fait identifié	wp	$wp_{\mathcal{ALCQI}}$
$add(i\ r\ j)$	$x\ r\ y$	$x\ (r + (i,j))\ y$	$(x = i \wedge\ y = j) \vee x\ r\ y$
	$x\ !r\ y$	$x\ !(r + (i,j))\ y$	$(x\ \ != i \vee y\ \ != j) \wedge (x\ !r\ y)$
	$x : (\leq n\ r\ C)$	$x : (\leq n\ (r + (i,j))\ C)$	$(x = i\ \wedge\ j : C\ \wedge\ i\ !r\ j\ \wedge\ x : (\leq (n-1)\ r\ C))$ $\vee\ (x\ ! = i\ \wedge\ x : (\leq n\ r\ C))$ $\vee\ (j : !C \wedge\ x : (\leq n\ r\ C))$ $\vee\ (i\ r\ j \wedge\ x : (\leq n\ r\ C))$ $\vee\ (x : (\leq (n-1)\ r\ C))$
$delete(i\ r\ j)$	$x\ !r\ y$	$x\ !(r - (i,j))\ y$	$(x = i \wedge\ y = j) \vee x\ !r\ y$
	$x\ r\ y$	$x\ (r - (i,j))\ y$	$(x\ \ != i \vee y\ \ != j) \wedge (x\ r\ y)$
	$x : (\geq n\ r\ C)$	$x : (\geq n\ (r - (i,j))\ C)$	$(x = i \wedge\ j : C \wedge\ i\ r\ j \wedge\ x : (\geq n+1\ r\ C))$ $\vee\ (x\ \ != i \wedge\ x : (\geq n\ r\ C))$ $\vee\ (j : !C \wedge\ x : (\geq n\ r\ C))$ $\vee\ (i\ !r\ j \wedge\ x : (\geq n\ r\ C))$ $\vee\ (x : (\geq n+1\ r\ C))$

$j : !C$, ou $i\ r\ j$ comme le montre les trois conjonctions qui suivent : $(x != i\ \wedge\ x : (\leq n\ r\ C))$, $(j : !C \wedge\ x : (\leq n\ r\ C))$ et $(i\ r\ j \wedge\ x : (\leq n\ r\ C))$. Dans les autres cas, le nombre de restrictions sera décrémenté par défaut, conduisant à la dernière conjonction $x : (\leq (n-1)\ r\ C)$.

Notons que chaque conjonction de la formule transformée en $\mathcal{ALCQI}$ peut être exploitée elle-même comme une précondition en considérant à la fois la tautologie $A \Rightarrow A \vee B$ et la règle de la conséquence gauche qui autorise à renforcer une précondition :

$$\frac{P \Rightarrow P' \quad \{P'\}S\{Q\}}{\{P\}S\{Q\}}$$

Ainsi, par le biais des conjonctions, différents niveaux de renforcement de la précondition peuvent être considérés. Par exemple, la première conjonction de la formule de l'exemple considéré implique la dernière, explicitement $(x = i\ \wedge\ j : C\ \wedge\ i\ !r\ j\ \wedge\ x : (\leq (n-1)\ r\ C)) \Rightarrow x : (\leq (n-1)\ r\ C)$; elle est donc plus forte. Par contre, la première conjonction et les trois conjonctions qui suivent sont des formules qui s'excluent mutuellement, du fait de la présence ou l'absence d'un arc entre i et j, ou bien de l'égalité et l'inégalité entre x et i, ou même de l'appartenance de j au concept C. Par exemple, $\big(x = i\ \wedge\ j : C\ \wedge\ i\ !r\ j\ \wedge\ x : (\leq (n-1)\ r\ C)\big) \not\Rightarrow \big(x != i\ \wedge\ x : (\leq n\ r\ C)\big)$. Le procédé d'élimination des substitutions offre ainsi un partitionnement des formules de la plus faible précondition.

TABLE 2.3 – Élimination du quantificateur de $wp(select\ v\ with\ F, Q)$

Instruction	Fait identifié	wp	$wp_{\mathcal{ALCQI}}$
select v with F	Q	$\forall v\ (F \Rightarrow Q)$	$\neg F \vee Q$

La quantification essentiellement universelle est une quantification qui s'élimine en substituant à la variable liée une constante qui n'a pas d'occurrence dans la formule. Dans ce cadre, les occurrences des variables v de $\forall v(F \Rightarrow Q)$ sont remplacées par des constantes. Dès lors que Small-t$\mathcal{ALC}$ interdit syntaxiquement la sélection d'une variable déjà manipulée, les variables v n'ont naturellement pas d'occurrences dans la règle au point d'analyse examiné. Ces variables sont ainsi remplacées par des constantes dotées des mêmes noms dans notre cas d'analyse, ce qui élimine le quantificateur $\forall$. Par exemple, $\forall a(a : A \Rightarrow a : A)$, où a est le nom d'une variable, devient $a : A \Rightarrow a : A$ où a s'assimile désormais à une constante. Cette dernière formule n'est pas une formule $\mathcal{ALCQI}$ à cause de l'implication ; en exploitant l'équivalence logique entre $F \Rightarrow Q$ et $\neg F \vee Q$, $a : A \Rightarrow a : A$ se transforme en $\neg(a : A) \vee a : A$ comme le montre le tableau 2.3.

2.2.3.2 Simplification des $wp_{\mathcal{ALCQI}}$

Les formules $\mathcal{ALCQI}$ exprimant les plus faibles préconditions peuvent être simplifiées après les instructions *add* et *delete* en exploitant la relation de possible équivalence entre les variables et des faits de la postcondition comme présenté à la sous-section 2.2.3.2.1. La sous-section 2.2.3.2.2 montre comment la formule correspondante à l'instruction *select* est interprétée et la sous-section 2.2.3.2.3 fait de même pour les autres instructions.

2.2.3.2.1 Les instructions *add* et *delete*

Considérons tout d'abord l'instruction $add(i : C)$. La plus faible précondition $\mathcal{ALCQI}$ par rapport à $x : C$ est $x : C \ \wedge \ x = i$. Cela signifie que, pour atteindre la postcondition $x : C$, soit le nœud x est préalablement de concept C, soit x et i désignent le même nœud. Sachant, après une première analyse du code, que les variables x et i sont non possiblement équivalentes, on peut affirmer que $x\ != i$, et dès lors la plus faible précondition peut se simplifier en $x : C$.

Considérons maintenant l'action $add(i\ r\ j)$ vis-à-vis du fait $x\ r\ y$. Si $x\ r\ y$ est identifié dans la postcondition, $wp_{\mathcal{ALCQI}}(add(i\ r\ j), x\ r\ y) = (x = i \wedge\ y = j) \vee x\ r\ y$ sera simplifiée dans le cas où le calcul d'alias révèle que les variables des paires (x, i) ou (y, j) sont non possiblement équivalentes, soit respectivement $x \not\simeq i$ ou $y \not\simeq j$. La précondition sera alors réduite à $x\ r\ y$.

Un exemple plus notable est la simplification de la plus faible précondition de l'instruction $add(i\ r\ j)$ par rapport à $x : (\leq n\ r\ C)$ indiquant qu'il existe au plus n arcs r sortants de x vers

des nœuds de concept C. L'ajout d'un arc r entre i et j peut avoir une incidence directe sur ce fait selon le concept de j, l'existence d'un arc r entre i et j et l'égalité entre i et x. Dès lors,

$$
\begin{aligned}
wp_{\mathcal{ALCQI}}(add(i\ r\ j), x : (\leq n\ r\ C)) = & (x = i\ \wedge\ j : C\ \wedge\ i\ !r\ j\ \wedge\ x : (\leq (n-1)\ r\ C)) \\
& \vee (x\ != i\ \wedge\ x : (\leq n\ r\ C)) \\
& \vee (j : !C\ \wedge\ x : (\leq n\ r\ C)) \\
& \vee (i\ r\ j\ \wedge\ x : (\leq n\ r\ C)) \\
& \vee (x : (\leq (n-1)\ r\ C))
\end{aligned}
$$

Sachant que $x \not\approx i$ grâce au calcul d'alias ou que $j : !C$ en examinant la précondition calculée à cette étape, la première conjonction $x = i\ \wedge\ j : C\ \wedge\ i\ !r\ j\ \wedge\ x : (\leq (n-1)\ r\ C)$ de la wp peut être écartée. Dans ces différentes situations portant sur x, i et j, $wp_{\mathcal{ALCQI}}$ peut être simplifiée en $x : (\leq n\ r\ C)$ en se référant aux deuxième et troisième conjonctions qui indiquent que le nombre de restrictions reste constant si $x \not\approx i$ ou $j : !C$.

Les tableaux 2.4 et 2.5 synthétisent la simplification des préconditions correspondantes aux instructions $add(i\ :\ C)$ et $add(i\ r\ j)$ respectivement. Pour chaque instruction s, la deuxième colonne

TABLE 2.4 – Simplification de la plus faible précondition $\mathcal{ALCQI}$ de $add(i : C)$

Instruction	Fait identifié	$wp_{\mathcal{ALCQI}}$	Condition	Précondition
$add(i : C)$	$x : C$	$x : C \vee x = i$	$x \not\approx i$	$x : C$
	$x : (\leq n\ r\ C)$	$(x\ r\ i\ \wedge\ i : !C \wedge\ x : (\leq (n-1)\ r\ C))$ $\vee (x\ !r\ i \wedge\ x : (\leq n\ r\ C))$ $\vee (i : C \wedge\ x : (\leq n\ r\ C))$ $\vee (x : (\leq (n-1)\ r\ C))$	$x\ !r\ i$	$x : (\leq n\ r\ C)$

TABLE 2.5 – Simplification de la plus faible précondition $\mathcal{ALCQI}$ de $add(i\ r\ j)$

Instruction	Fait identifié	$wp_{\mathcal{ALCQI}}$	Condition	Précondition
$add(i\ r\ j)$	$x\ r\ y$	$(x = i \wedge y = j) \vee x\ r\ y$	$x \not\approx i \vee y \not\approx j$	$x\ r\ y$
	$x\ !r\ y$	$(x\ != i \vee y\ != j) \wedge (x\ !r\ y)$	$x \not\approx i$	$x\ != i \wedge x\ !r\ y$
			$y \not\approx j$	$y\ != j \wedge x\ !r\ y$
	$x : (\leq n\ r\ C)$	$(x = i\ \wedge\ j : C\ \wedge\ i\ !r\ j\ \wedge\ x : (\leq (n-1)\ r\ C))$ $\vee (x\ != i\ \wedge\ x : (\leq n\ r\ C))$ $\vee (j : !C \wedge\ x : (\leq n\ r\ C))$ $\vee (i\ r\ j \wedge\ x : (\leq n\ r\ C))$ $\vee (x : (\leq (n-1)\ r\ C))$	$x \not\approx i \vee j : !C$	$x : (\leq n\ r\ C)$

représente les faits devant être identifiés dans la postcondition. La troisième colonne montre la plus faible précondition $wp_{\mathcal{ALCQI}}(s, f)$ de l'instruction s par rapport au fait identifié f. La quatrième colonne donne les conditions permettant d'ignorer des conjonctions de cette formule afin de la simplifier. La précondition simplifiée est présentée à la dernière colonne.

Les tableaux 2.6 et 2.7 des instructions *delete(i : C)* et *delete(i r j)* sont symétriques aux tableaux correspondants aux instructions *add*.

TABLE 2.6 – Simplification de la plus faible précondition $\mathcal{ALCQI}$ de $delete(i : C)$

Instruction	Fait identifié	$wp_{\mathcal{ALCQI}}$	Condition	Précondition
$delete(i : C)$	$x : !C$	$x : !C \vee x = i$	$x \not\approx i$	$x : !C$
	$x : (\geq n\ r\ C)$	$(x\ r\ i\ \wedge\ i : C \wedge\ x : (\geq n+1\ r\ C))$ $\vee\ (x\ !r\ i \wedge\ x : (\geq n\ r\ C))$ $\vee\ (i : !C \wedge\ x : (\geq n\ r\ C))$ $\vee\ (x : (\geq n+1\ r\ C))$	$x\ !r\ i$	$x : (\geq n\ r\ C)$

TABLE 2.7 – Simplification de la plus faible précondition $\mathcal{ALCQI}$ de $delete(i\ r\ j)$

Instruction	Fait identifié	$wp_{\mathcal{ALCQI}}$	Condition	Précondition
$delete(i\ r\ j)$	$x\ !r\ y$	$(x = i \wedge y = j) \vee x\ !r\ y$	$x \not\approx i \vee y \not\approx j$	$x\ !r\ y$
	$x\ r\ y$	$(x != i \vee y != j) \wedge (x\ r\ y)$	$x \not\approx i$	$x != i \wedge x\ r\ y$
			$y \not\approx j$	$y != j \wedge x\ r\ y$
	$x : (\geq n\ r\ C)$	$(x = i \wedge\ j : C \wedge\ i\ r\ j \wedge\ x : (\geq n+1\ r\ C))$ $\vee\ (x\ != i \wedge\ x : (\geq n\ r\ C))$ $\vee\ (j : !C \wedge\ x : (\geq n\ r\ C))$ $\vee\ (i\ !r\ j \wedge\ x : (\geq n\ r\ C))$ $\vee\ (x : (\geq n+1\ r\ C))$	$x \not\approx i \vee j : !C$	$x : (\geq n\ r\ C)$

Ces tableaux montrent comment des formules complexes en formes normales disjonctives peuvent se réduire à une conjonction en se basant sur les variables possiblement équivalentes ou sur des faits donnés explicitement dans la postcondition.

2.2.3.2.2 L'instruction *select*

Jusqu'à présent, l'analyse statique transforme le prédicat Q en un nouveau prédicat P en se basant sur les plus faibles préconditions des instructions. Cependant, l'analyse opère différemment

lorsqu'il s'agit de l'instruction *select* où $wp(select\ v\ with\ F, Q) = \forall v\ (F \Rightarrow Q)$ est exprimée en $\mathcal{ALCQI}$ par $\neg F \vee Q$ comme mentionné à la section 2.2.3.1. En effet, la plus faible précondition de cette instruction implique deux formules : F donnée par l'instruction *select*, et la postcondition Q.

Dès lors que chaque instruction est analysée vis-à-vis d'une conjonction Q de la postcondition, l'analyse statique repartit Q en deux conjonctions en isolant les faits qui manipulent les variables v de l'instruction *select*. Soit Q_v la formule conjonctive de ces faits identifiés, et $Q_{v'}$ la formule conjonctive des autres faits, de sorte que $Q = Q_v \wedge\ Q_{v'}$. Par exemple, considérant la formule $Q = x\ R\ y\ \wedge\ y : C$ et l'instruction *select x with x* : C, $Q_v = x\ R\ y$ et $Q_{v'} = y : C$.

L'analyse statique vérifie ensuite si $\neg F \vee Q_v$ est valide. Si c'est le cas, la précondition est réduite à $Q_{v'}$ sans affecter la validité du triplet de Hoare. Ceci est justifié par l'implication logique $((Q = Q_v \wedge\ Q_{v'}) \wedge\ (\neg F \vee Q_v)) \Rightarrow (Q_{v'} \Rightarrow (\neg F \vee Q))$ signifiant que $Q_{v'}$ est plus forte que la plus faible précondition $\mathcal{ALCQI}$ $\neg F \vee Q$ et qu'elle constitue ainsi une précondition valide. Inversement, la non-validité de l'implication $F \Rightarrow Q_v$ amène à transformer Q en prédicat $false$ du fait que rien ne peut être conclu sur la validité de la transformation. Cette situation a pour but d'avertir le développeur que des incohérences existent dans sa transformation entre l'instruction *select* et la postcondition Q. Ces cas sont donnés au tableau 2.8.

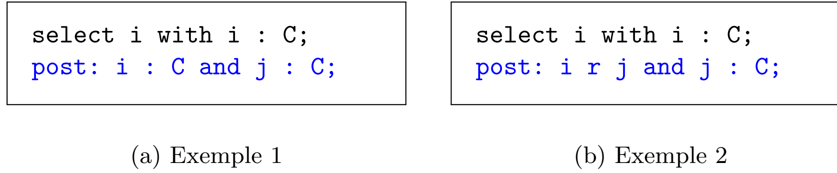
TABLE 2.8 – Condition de vérification de l'instruction *select*

Instruction	Postcondition	$wp_{\mathcal{ALCQI}}$	Condition de vérification	Précondition
select v with F	$Q = Q_v \wedge\ Q_{v'}$	$\neg F \vee Q$	$F \Rightarrow Q_v$	$Q_{v'}$
			$F \not\Rightarrow Q_v$	*false*

Afin de vérifier les conditions mentionnées au tableau 2.8, un évaluateur de formules logiques $\mathcal{ALCQI}$ a été développé en établissant la table de vérité correspondante. Contrairement aux instructions *add* et *delete* qui impliquent directement des nœuds et arcs du graphe, l'instruction *select* met en jeu des formules $\mathcal{ALCQI}$. Dès lors, la simplification de $wp_{\mathcal{ALCQI}}$ se transforme en une simplification de formules, si les conditions de vérification sont établies par l'évaluateur. La table de vérité permet d'établir que la formule considérée est valide.

Considérons l'exemple du code de la figure 2.15a composé de l'instruction *select i with i* : C et de la postcondition $Q = i : C\ and\ j : C$. La première étape consiste à identifier $Q_v = i : C$ et $Q_{v'} = j : C$. En exploitant l'évaluateur de formules logiques Small-t$\mathcal{ALC}$, l'implication $i : C \Rightarrow i : C$ est prouvée correcte. Dès lors, la précondition extraite est $j : C$ fixée par $Q_{v'}$.

Considérons maintenant une autre postcondition $Q = i\ r\ j\ and\ j : C$ pour le même code comme mentionné à la figure 2.15b. Dans ce cas, l'évaluateur de formules logiques Small-t$\mathcal{ALC}$ montre que l'implication entre $F = i : C$ et $Q_v = i\ r\ j$ n'est pas valide. Par conséquent, l'analyseur statique



```
select i with i : C;
post: i : C and j : C;
```

(a) Exemple 1

```
select i with i : C;
post: i r j and j : C;
```

(b) Exemple 2

FIGURE 2.15 – Exemples de codes composés d'une instruction *select* et d'une postcondition

avertit le développeur que la transformation ne peut pas être exécutée car aucun état d'un graphe ne peut satisfaire la précondition *false*.

2.2.3.2.3 Les autres instructions

La procédure de simplification de la plus faible précondition $\mathcal{ALCQI}$ a été présentée pour les instructions atomiques en considérant chaque conjonction $Q_i = \wedge q_j$ de Q avec $Q = \vee Q_i$. La précondition d'une séquence d'instructions $s1; s2$ est calculée conventionnellement en posant $wp(s1; s2, Q) = wp(s1, wp(s2, Q))$. De même, $wp(if\ c\ then\ s1\ else\ s2, Q) = (c \Rightarrow wp(s1, Q)) \wedge\ (\neg c \Rightarrow wp(s2, Q)) = (\neg c \vee wp(s1, Q)) \wedge (c \vee wp(s2, Q))$. Seule la plus faible précondition de l'instruction $if\ c\ then\ s1\ else\ s2$ n'est pas transformée en forme normale disjonctive lors du calcul des préconditions de la règle conformément à la définition classique de sa *wp*.

A part les boucles, les plus faibles préconditions peuvent être calculées automatiquement comme présenté ci-dessus. Cependant, l'instruction *while* exige de la part du développeur, comme mentionné à la section 2.1, la définition d'un invariant faisant office de plus faible précondition. Dès lors, en se basant sur l'invariant fourni par le développeur, notre évaluateur de formules logiques vérifie si la condition de vérification *vc* du prouveur est valide. Si c'est le cas, la précondition de la boucle est réduite à son invariant en le transformant en forme normale disjonctive comme présenté au tableau 2.9. Dans le cas contraire, Q est transformée en *false* vu que l'invariant ne satisfait pas la condition de vérification de la boucle. Cette situation a pour but d'avertir le développeur que l'invariant proposé est incorrect. L'évaluateur de formules est de plus en mesure de préciser éventuellement la conjonction erronée de la condition de vérification lors du calcul régressif des préconditions,

TABLE 2.9 – Condition de vérification de l'instruction *while*

Instruction	Postcondition	Condition de vérification	Précondition
$\{inv\}\ while\ c\ do\ s$	Q	$(inv \wedge\ \neg c \Rightarrow Q)$ $\wedge\ (inv \wedge\ c \Rightarrow wp(s, inv))$ $\wedge\ inv$	inv
		sinon	$false$

au contraire du prouveur qui considère la correction de la règle intégralement et reporte ainsi le diagnostic sur le graphe contre-exemple car la formule $vc(S, Q) \wedge \ (P \Rightarrow wp(S, Q))$ est évaluée en totalité pour vérifier la règle concernée.

Vu que la précondition de l'instruction *while* est associée directement à son invariant fourni par le développeur, une aide à la découverte d'invariants nous semble une perspective intéressante à ces travaux. Ceci permettrait de compléter nos dispositifs d'assistance.

2.2.3.3 Spécificités du transformateur

Usuellement, les formules $\mathcal{ALCQI}$ exprimant la plus faible précondition sont simplifiées en éliminant les clauses conjonctives invalides. Cette simplification conduit à une précondition P plus forte que la plus faible précondition $wp(S, Q)$. En particulier, lorsque deux variables sont non équivalentes, un raisonnement déductif est effectué en s'appuyant sur l'équivalence et les liens d'implication entre P et $wp(S, Q)$. La précondition calculée est présentée sous la forme d'une formule normale disjonctive donnant lieu à différentes alternatives possibles. Chaque alternative P constitue une proposition validant la formule $(P \Rightarrow wp(S, Q)) \wedge \ vc(S, Q)$. A noter qu'un développeur peut proposer une précondition plus faible que la précondition de l'analyseur qui sera validée toutefois par le composant de preuve.

Le choix d'une forme normale disjonctive en tant que résultat de l'analyse peut conduire à des formules complexes dans le cas où la règle présente plusieurs chemins d'exécution possibles. Notre présentation de la précondition est ainsi plus adaptée à des instructions écrites en séquence. D'autre part, les conjonctions résultantes se caractérisent par des redondances de formules logiques [20, 79], notamment dans le cas des restrictions de multiplicité. Les faits seront ainsi recopiés dans différentes conjonctions de la précondition $\mathcal{ALCQI}$. Cette duplication ne favorise pas toujours la lisibilité des conjonctions proposées par l'analyseur.

Il est admis que la logique de Hoare est correcte et complète [67], moyennant que la logique sous-jacente le soit, ce qui a été démontré dans le cadre du projet CLIMT pour les triplets Small-t$\mathcal{ALC}$ [24]. Dériver ce résultat dans le cas de notre analyse impose initialement de démontrer que notre algorithme d'élimination des substitutions établit une formule $wp_{\mathcal{ALCQI}}$ équivalente à sa wp homologue, cette équivalence ayant été établie intuitivement. Puis, à exploiter la réduction d'une formule $wp_{\mathcal{ALCQI}}$ induite par le calcul d'alias sur la base de propriétés générales de la plus faible précondition, indépendantes de tout langage de programmation, comme la distributivité de la disjonction soit $wp(S, P) \ \vee \ wp(S, Q) \Rightarrow wp(S, P \vee Q)$, ou la loi du miracle exclu qui stipule que $wp(S, false) = false$. Cette étude n'a pas été conduite à ce jour et pourrait constituer l'une des perspectives de cette thèse.

2.2.4 Exemple

Considérons comme exemple le code de transformation et la postcondition de la figure 2.16. La première instruction ajoute un nœud x au concept C ; la seconde ajoute un arc r entre des nœuds y et z. La postcondition indique que x possède au plus trois successeurs r vers des nœuds de concept C, et que y appartient à $!C$.

```
add(x : C);
add(y r z);
post: x : (≤ 3 r C) and y : !C;
```

FIGURE 2.16 – Exemple d'un code de transformation et d'une postcondition

Pour atteindre la postcondition donnée, l'analyseur engage un calcul régressif de la plus faible précondition selon la formule $wp(s1; s2, Q) = wp(s1, wp(s2, Q))$ avec $s1 = add(x : C)$ et $s2 = add(y\ r\ z)$. Quatre conjonctions possibles sont ainsi exposées au développeur :

1. *x != y and y : !C and x : (<= 2 r C)*
2. *x != y and y : !C and x : C and x : (<= 3 r C)*
3. *x != y and y : !C and x !r x and x : (<= 3 r C)*
4. *x != y and y : !C and x r x and x : !C and x : (<= 2 r C)*

Les quatre conjonctions extraites par l'analyseur statique sont le résultat de simplifications de la plus faible précondition $\mathcal{ALCQI}$ en se basant sur le calcul d'alias qui certifie, d'après le code et la postcondition donnés, que x et y sont non possiblement équivalentes. En effet, la plus faible précondition de l'instruction *add(y r z)* calculée en premier temps vis-à-vis de la postcondition donnée est la suivante :

$$
\begin{aligned}
wp_{\mathcal{ALCQI}}\big(add(y\ r\ z), x : (\leq 3\ r\ C)\ and\ y : !C\big) = \Big(&\big(x = y\ \wedge\ z : C\ \wedge\ y\ !r\ z\ \wedge\ x : (\leq 2\ r\ C)\big) \\
&\vee \big(x\ != y\ \wedge\ x : (\leq 3\ r\ C)\big) \\
&\vee \big(z : !C\ \wedge\ x : (\leq 3\ r\ C)\big) \\
&\vee \big(y\ r\ z\ \wedge\ x : (\leq 3\ r\ C)\big) \\
&\vee \big(x : (\leq 2\ r\ C)\big)\Big) \\
&\wedge\ y : !C
\end{aligned}
$$

Sachant que $x \not\approx y$ et donc que $x \neq y$, les cinq premières conjonctions se réduisent à $(x : (\leq 3\ r\ C))$. Dès lors, la postcondition calculée après l'instruction *add(y r z)* est *x : (≤ 3 r C) and y : !C*. Cette dernière se transforme après l'instruction $add(x : C)$ en une formule disjoignant les quatre conjonctions finales présentées au développeur.

Chacune de ces propositions constitue une précondition qui permet d'obtenir un triplet de Hoare correct. La première formule *x != y and y : !C and x : (<= 2 r C)* est la plus faible : le nombre de

restrictions est borné par 2 dès lors qu'aucune propriété n'est affirmée sur x. Certaines conjonctions admettent une restriction au plus égale à 3, comme dans la postcondition. Par exemple, la deuxième formule *x != y and y : !C and x : C and x : (<= 3 r C)* indique que si x est préalablement de concept C, la première instruction du code $add(x : C)$ sera sans effet. D'autre part, x et y ne désigneront évidement pas le même nœud du graphe car l'un est de concept C et l'autre de concept $!C$. Le nombre de restrictions du fait $x : (\leq 3\ r\ C)$ de la postcondition est ainsi à reporter sans changement dans la précondition.

Cette même deuxième formule est en contradiction avec la dernière proposition de l'analyseur en considérant l'appartenance de x à C, elle-même plus forte que la première au sens de l'implication :

(2) *x != y and y : !C and x : C and x : (<= 3 r C)* $\not\Rightarrow$ (4) *x != y and y : !C and x r x and x : !C and x : (<= 2 r C)*
(4) *x != y and y : !C and x r x and x : !C and x : (<= 2 r C)* $\not\Rightarrow$ (2) *x != y and y : !C and x : C and x : (<= 3 r C)*
(4) *x != y and y : !C and x r x and x : !C and x : (<= 2 r C)* $\Rightarrow$ (1) *x != y and y : !C and x : (<= 2 r C)*

La variété de ces formules donne le choix au développeur de renforcer les contraintes d'applicabilité de la règle dans la précondition autant qu'il le souhaite.

En ce sens, nous favorisons l'émergence de l'intention d'une règle et aidons ainsi les développeurs à construire et à annoter leur code avec des spécifications afin d'obtenir au final un triplet correct par construction.

2.3 Mise au point d'un triplet d'une règle Small-t$\mathcal{ALC}$

Le prouveur Small-t$\mathcal{ALC}$, comme tous les outils de vérification automatiques, ne nécessite aucune interaction avec les développeurs ; il effectue de manière autonome la vérification formelle d'un programme. En contre-partie, la rétroaction en cas d'échec reste faible. Inversement, une vérification interactive exige qu'un développeur expert soit en mesure de guider l'assistant de preuve à travers des tactiques de preuve.

Afin d'aider un développeur non-expert à construire des règles de transformation correctes, une approche de vérification auto-active [80] se situant entre la vérification automatique et la vérification interactive peut être adoptée en exploitant notre analyseur statique [87]. Cette approche permet généralement au développeur d'annoter progressivement son code avec des spécifications, avant d'engager la vérification proprement dite. Le processus peut être itéré jusqu'à atteindre un programme prouvé.

L'approche auto-active a deux avantages principaux : d'une part, elle favorise un développement incrémental avec un diagnostic riche à chaque itération, d'autre part, elle comble le fossé entre un développeur non-expert et le prouveur.

Étant donné que l'analyseur statique calcule plusieurs conjonctions en tant que préconditions relatives à un code et à une postcondition comme expliqué à la section précédente, le nombre de préconditions extraites peut être important. Ce nombre peut être réduit en permettant aux

développeurs d'exprimer plus finement l'intention de la règle. L'idée est d'alterner l'intervention de l'analyseur statique avec celle du développeur dans un processus itératif et incrémental jusqu'à l'obtention d'un triplet correct. Pour être efficace, l'interactivité entre le développeur et l'analyse doit se traduire par des propositions directement compréhensibles par le développeur.

Un exemple d'un scénario possible est le suivant : tout d'abord, l'analyseur statique suggère, à partir d'un code et d'une postcondition donnée, une précondition en forme normale disjonctive. Ensuite, le développeur sélectionne certaines des propositions suggérées qui reflètent au mieux son intention. Il peut prendre directement en considération une propositions comme précondition de la règle et terminer le processus, ou bien affiner son code ou sa postcondition avant d'interroger à nouveau l'analyseur statique au cours d'une nouvelle itération. En règle générale, le développeur met à jour son code de transformation ou affine sa spécification en y injectant les faits des conjonctions choisies. De cette façon, la règle de transformation est construite progressivement.

Afin d'illustrer l'application de cette approche auto-active, un exemple concret est présenté ci-dessous consistant en un développement d'une règle d'un programme simulant les activités d'un hôpital, inspiré de [24].

Dans ce programme, considérons des patients (*Patient*) et des médecins (*Doctor*) dans un hôpital comprenant plusieurs départements (*Department*). Chaque médecin est rattaché à (*worksIn*) un département, traite (*treats*) des patients, et prend le statut de médecin traitant (*refDr*) pour chacun de ses patients. Chaque département est dirigé par (*directedBy*) un chef (*head*), un des médecins du département. A noter qu'un chef ne peut pas avoir en charge plus de trois patients simultanément.

Le développeur souhaitant construire la règle *assignDoctor* pour affecter un médecin traitant *dr* à un patient *p*, écrit une première version du code de la règle et ajoute ainsi les relations *treats* et *refDr* entre le patient *p* et le médecin *dr* comme le montre la figure 2.17. La postcondition traduit les exigences du programme : le docteur *dr* affecté au patient *p* doit travailler dans le même département où *p* est inscrit, et le chef ne doit pas traiter plus de trois patients, soit *head* : ($\leq$ 3 *treats Patient*).

```
rule assignDoctor {
 add(dr treats p);
 add(p refDr dr);
 post: p : Patient and dr Doctor and dep : Department
       and dep registers p and dr worksIn dep
       and dep directedBy head and head : Doctor
       and dr treats p
       and head : (≤ 3 treats Patient)
       and p : (= 1 refDr Doctor);
}
```

FIGURE 2.17 – Première version de la règle *assignDoctor*

A noter que le fait *p refDr dr*, qui vérifie qu'un médecin est affecté au patient *p*, est assuré par la restriction $p : (= 0\ refDr\ Doctor)$.

Afin de compléter la règle *assignDoctor*, le développeur sollicite l'analyseur statique qui lui propose les douze propositions suivantes comme préconditions possibles :

1. *p* : *Patient and dr* : *Doctor and dep* : *Department and dep registers p*
   and dr worksIn dep and dep directedBy head and head : *Doctor*
   and head : $(\leq 2\ treats\ Patient)$
   and $p : (= 0\ refDr\ Doctor)$

2. *p* : *Patient and dr* : *Doctor and dep* : *Department and dep registers p*
   and dr worksIn dep and dep directedBy head and head : *Doctor*
   and **p !refDr dr**
   and head : $(\leq 2\ treats\ Patient)$
   and $p : (= 0\ refDr\ Doctor)$

3. *p* : *Patient and dr* : *Doctor and dep* : *Department and dep registers p*
   and dr worksIn dep and dep directedBy head and head : *Doctor*
   and **dr != head**
   and head : $(\leq 3\ treats\ Patient)$
   and $p : (= 0\ refDr\ Doctor)$

4. *p* : *Patient and dr* : *Doctor and dep* : *Department and dep registers p*
   and dr worksIn dep and dep directedBy head and head : *Doctor*
   and **p !refDr dr**
   and **dr != head**
   and head : $(\leq 3\ treats\ Patient)$
   and $p : (= 0\ refDr\ Doctor)$

5. *p* : *Patient and dr* : *Doctor and dep* : *Department and dep registers p*
   and dr worksIn dep and dep directedBy head and head : *Doctor*
   and **p refDr dr**
   and head : $(\leq 2\ treats\ Patient)$
   and $p : (= 1\ refDr\ Doctor)$

6. *p* : *Patient and dr* : *Doctor and dep* : *Department and dep registers p*
   and dr worksIn dep and dep directedBy head and head : *Doctor*

and **p refDr dr**
and **dr != head**
and $head : (\leq 3\ treats\ Patient)$
and $p : (= 1\ refDr\ Doctor)$

7. $p : Patient$ *and* $dr : Doctor$ *and* $dep : Department$ *and dep registers p and dr worksIn dep and dep directedBy head and* $head : Doctor$
   and **dr = head**
   and **dr !treats p**
   and $head : (\leq 2\ treats\ Patient)$
   and $p : (= 0\ refDr\ Doctor)$

8. $p : Patient$ *and* $dr : Doctor$ *and* $dep : Department$ *and dep registers p and dr worksIn dep and dep directedBy head and* $head : Doctor$
   and **dr = head**
   and **dr !treats p**
   and **p !refDr dr**
   and $head : (\leq 2\ treats\ Patient)$
   and $p : (= 0\ refDr\ Doctor)$

9. $p : Patient$ *and* $dr : Doctor$ *and* $dep : Department$ *and dep registers p and dr worksIn dep and dep directedBy head and* $head : Doctor$
   and **dr = head**
   and **dr !treats p**
   and **p refDr dr**
   and $head : (\leq 2\ treats\ Patient)$
   and $p : (= 1\ refDr\ Doctor)$

10. $p : Patient$ *and* $dr : Doctor$ *and* $dep : Department$ *and dep registers p and dr worksIn dep and dep directedBy head and* $head : Doctor$
    and **dr treats p**
    and $head : (\leq 3\ treats\ Patient)$
    and $p : (= 0\ refDr\ Doctor)$

11. $p : Patient$ *and* $dr : Doctor$ *and* $dep : Department$ *and dep registers p and dr worksIn dep and dep directedBy head and* $head : Doctor$
    and **dr treats p**

and **p refDr dr**
and $head : (\leq 3\ treats\ Patient)$
and $p : (= 1\ refDr\ Doctor)$

12. $p : Patient$ *and* $dr : Doctor$ *and* $dep : Department$ *and* $dep\ registers\ p$
and $dr\ worksIn\ dep$ *and* $dep\ directedBy\ head$ *and* $head : Doctor$
and **dr treats p**
and **p !refDr dr**
and $head : (\leq 3\ treats\ Patient)$
and $p : (= 0\ refDr\ Doctor)$

Remarquons que le nombre de restrictions dans ces propositions varie selon l'existence des relations *treats* et *refDr* entre *p* et *dr*, et l'égalité entre *dr* et *head*, indiquées en gras dans les douze conjonctions. Par exemple, la présence du fait *p refDr dr* dans les conjonctions (5), (6), (9) et (11), exprimant que le patient *p* a un médecin traitant, rend le nombre de restrictions des arcs *refDr* sortants de *p* vers le concept *Doctor* égal à 1. Dans les conjonctions (7), (8) et (9), le fait $dr = head$ signifiant que le docteur choisi pour traiter le patient est un chef de département, rend le nombre de restrictions des arcs *treats* sortants de *head* vers le concept *Patient* inférieur ou égal à 2.

D'autre part, ces formules présentent différents niveaux de renforcement de la précondition, outre celles qui s'excluent mutuellement : la conjonction (1) est impliquée par les onze autres conjonctions et constitue ainsi la formule la plus faible, la formule (4) implique les deux formules (2) et (3), (6) implique (5), (8) et (9) impliquent (7), et (11) et (12) impliquent (10). Les couples de formules (2, 5), (8, 9) et (11, 12) s'excluent mutuellement : (5, 9, 11) indiquent la présence de la relation *refDr* et (2, 8 et 12) explicitent son absence.

```
rule assignDoctor {
  add(dr treats p);
  add(p refDr dr);
  post: p : Patient and dr Doctor and dep : Department
        and dep registers p and dr worksIn dep
        and dep directedBy head and head : Doctor
        and dr treats p
        and dr != head
        and head : (≤ 3 treats Patient)
        and p : (= 1 refDr Doctor);
}
```

FIGURE 2.18 – Deuxième version de la règle *assignDoctor*

Afin de réduire le nombre de conjonctions extraites, le développeur modifie sa postcondition en y ajoutant le fait *dr* != *head* signifiant que le patient est affecté à un médecin autre que le chef d'un département. La figure 2.18 montre le code modifié de la règle *assignDoctor*.

La spécification de l'inégalité entre les deux variables *dr* et *head* dans la postcondition réduit le nombre de propositions de douze à trois :

1. p : $Patient$ and dr : $Doctor$ and dep : $Department$ and dep $registers$ p
   and dr $worksIn$ dep and dep $directedBy$ $head$ and $head$: $Doctor$
   and dr $!=$ $head$
   and $head$: $(\leq 3\ treats\ Patient)$
   and p : $(= 0\ refDr\ Doctor)$

2. p : $Patient$ and dr : $Doctor$ and dep : $Department$ and dep $registers$ p
   and dr $worksIn$ dep and dep $directedBy$ $head$ and $head$: $Doctor$
   and dr $!=$ $head$
   and **p !refDr dr**
   and $head$: $(\leq 3\ treats\ Patient)$
   and p : $(= 0\ refDr\ Doctor)$

3. p : $Patient$ and dr : $Doctor$ and dep : $Department$ and dep $registers$ p
   and dr $worksIn$ dep and dep $directedBy$ $head$ and $head$: $Doctor$
   and dr $!=$ $head$
   and **p refDr dr**
   and $head$: $(\leq 3\ treats\ Patient)$
   and p : $(= 1\ refDr\ Doctor)$

Dans une nouvelle itération, le développeur ignore la dernière proposition qui spécifie que le patient p a déjà un médecin traitant dr : il atteste ainsi que la règle doit ajouter l'arc typé *refDr* entre p et dr. Notons que la première conjonction est similaire à la deuxième sauf que cette dernière explicite la non-existence de la relation $refDr$ entre p et dr, d'où elle implique la première. Le développeur prend ainsi en compte la deuxième proposition qui permet l'ajout de l'arc *refDr* et l'injecte directement comme précondition de sa règle *assignDoctor* dont la version finale est donnée à la figure 2.19.

En exploitant l'analyseur statique dans une approche auto-active, le développeur rédige progressivement ses transformations en modifiant le code et les spécifications à chaque itération dans l'objectif d'atteindre des règles correctes par construction.

A noter que l'analyseur statique reproduit dans toutes les conjonctions des faits de la postcondition qui ne sont pas modifiés par le code de la règle. Une perspective d'amélioration de l'analyse

```
rule assignDoctor {
 pre: p : Patient and dr : Doctor and dep : Department
      and dep registers p and dr worksIn dep
      and dep directedBy head and head : Doctor
      and dr != head and p !refDr dr
      and head : (≤ 3 treats Patient)
      and p : (= 0 refDr Doctor);
 add(dr treats p);
 add(p refDr dr);
 post: p : Patient and dr Doctor and dep : Department
       and dep registers p and dr worksIn dep
       and dep directedBy head and head : Doctor
       and dr treats p
       and dr != head
       and head : (≤ 3 treats Patient)
       and p : (= 1 refDr Doctor);
}
```

FIGURE 2.19 – Version finale de la règle *assignDoctor*

consisterait à isoler et à inférer des faits qui ne sont pas impactés par le code, et donc de circonscrire les effets de la règle sur le graphe.

2.4 Travaux similaires

Plusieurs travaux considèrent la dérivation d'une précondition à partir d'une postcondition et d'un code source. Plutôt que de se baser sur une configuration initiale du système et d'explorer les états cibles possibles, le raisonnement régressif suppose un certain état cible et calcule le ou les états sources correspondants. Dans le domaine des transformations de modèles, Clarisó et al. [32] présentent une technique pour synthétiser automatiquement les conditions d'application des règles. Ces conditions sont dérivées de postconditions exprimées en OCL de telle sorte que tout modèle qui les satisfait garantira la postcondition à l'issue de toute exécution possible de la règle. Une postcondition peut être une expression qui décrit l'effet de la règle, ou bien une contrainte d'intégrité du méta-modèle qui doit être préservée. Comme illustré à la figure 2.20, leur démarche nécessite d'abord de traduire une règle de transformation en une liste d'actions atomiques, pour ensuite capturer l'effet de chaque action sur l'expression OCL en effectuant des remplacements syntaxiques des sous-expressions affectées par l'action. Small-t$\mathcal{ALC}$ qui vise des transformations de graphes en général ne possède pas la notion de méta-modèle. La postcondition exprime ainsi l'effet de la règle, et le processus d'extraction s'appuie directement sur les instructions.

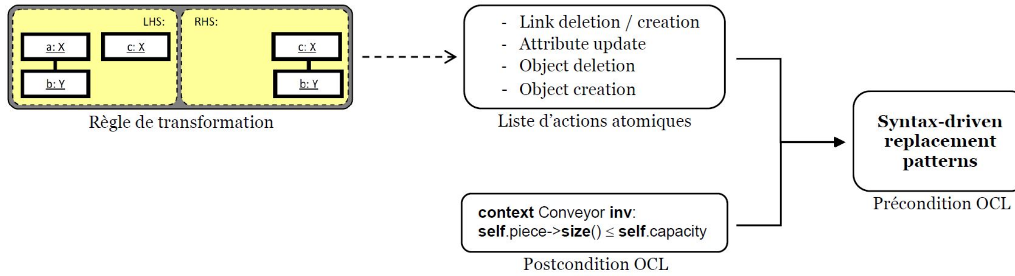


FIGURE 2.20 – Démarche de dérivation de préconditions OCL [32]

D'autres travaux [46, 62] proposent des méthodes d'extraction basées sur des opérations catégoriques, ce qui rend leur application complexe en pratique, à l'opposé de Small-t$\mathcal{ALC}$ qui exploite la même logique pour ses instructions et spécifications. En autorisant la manipulation des attributs d'un graphe, Deckwerth et Varró [46] génèrent des préconditions à partir de contraintes de graphes qui présentent non seulement des propriétés structurelles, mais aussi des formules logiques exprimant des valeurs et des restrictions sur les attributs. Heckel et al. [62] présentent une méthode qui transforme les conditions globales de consistance en précondition pour chaque règle. Ces préconditions, exprimées graphiquement, décrivent des conditions basiques comme l'existence ou l'unicité de certains nœuds et arcs du graphe. Concernant les schémas conceptuels, les travaux dans [34] étudient la génération des plus faibles préconditions pour les opérations atomiques appliquées à l'état du système, par exemple la création d'instance ou la mise à jour des attributs, dans le but de s'assurer que l'ensemble des contraintes d'intégrité sera respecté.

Calcagno et al. [26, 27] proposent un algorithme hybride qui combine l'analyse de forme (*shape analysis*) à une analyse itérative pour calculer les préconditions d'un programme. L'analyse de forme tente d'identifier les formes des structures de données, en mémoire en des points spécifiques d'exécution du programme [48]. L'objectif de leur analyse vise ainsi à générer des préconditions qui couvrent autant d'états fiables que possible, en garantissant que le programme n'entraîne pas d'erreurs de mémoire. La première phase de leur algorithme calcule les préconditions possibles, et la deuxième phase effectue une analyse en mode progressif pour écarter les préconditions qui posent problèmes. L'extraction des postconditions correspondantes aux préconditions validées permet d'aboutir à des triplets de Hoare corrects.

Dans le même contexte, Podelski et al. [102] étudient le problème de la terminaison des programmes manipulant des zones de mémoire. Étant donné un tel programme, leur algorithme HEAPINFER infère un ensemble de préconditions exprimant les états initiaux de la mémoire au point d'entrée d'un fragment de code donné, qui conduisent à la terminaison des calculs. Il s'agit d'appliquer itérativement une analyse de terminaison à une abstraction du programme jusqu'à la découverte d'un contre-exemple sous la forme d'un calcul abstrait non terminé. En exploitant le

contre-exemple, l'analyse de forme produit une précondition décrivant la structure attendue des données en mémoire.

Afin de prouver la fiabilité de la mémoire pour des programmes impératifs comme Java et C, Moy [93] propose une méthode de vérification d'assertion modulaire en combinant l'interprétation abstraite, le calcul de la plus faible précondition et l'élimination des quantificateurs tout en considérant le problème de l'*aliasing* des variables. Ainsi, étant donnée une fonction C décorée par des assertions, une précondition suffisante est inférée permettant d'assurer la validité du code de la fonction.

Nous partageons avec ces différents auteurs l'inférence automatique de préconditions dites suffisantes car elles garantissent l'absence de violations d'assertion au sein de la règle. En référence à l'*aliasing*, ces travaux se focalisent aussi sur la vérification de programmes impératifs manipulant des pointeurs. Certains [26, 48] s'appuient sur la logique de séparation [109] qui offre des mécanismes pour exprimer aisément que différentes structures de données sont disjointes en mémoire.

En ce qui concerne l'assistance à construire des programmes corrects, plusieurs travaux [21, 66] considèrent le développement itératif. Becker et al. [21] proposent une approche pour développer de manière itérative des règles de restructuration du code préservant les fonctionnalités qu'il est censé réaliser (*refactoring*). La figure 2.21 explicite la règle de restructuration *Pull Up Method* qui recopie la méthode m dans la classe ancêtre *super* lorsque toutes ses sous-classes *sub* effectuent le même traitement m. Appliquée à un modèle de code, la règle supprime les éléments qui sont annotés '- -', crée les éléments annotés '++', et conserve les éléments non annotés.

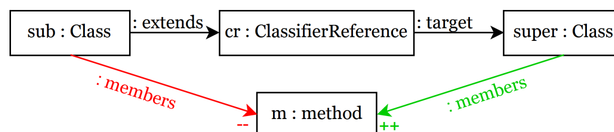


FIGURE 2.21 – Règle de restructuration *Pull up Method* [21]

Doté d'un langage de modélisation avec des contraintes bien définies et d'une spécification de restructuration, leur vérificateur d'invariant est capable de détecter des violations de contraintes sur les restructurations selon cette spécification. Dès lors, il signale les violations en retournant des contre-exemples correspondants à toutes les situations problématiques. Le développeur est alors en mesure d'inspecter ces contre-exemples et de modifier la spécification de restructuration en conséquence. Cette étape peut être répétée jusqu'à ce que le vérificateur d'invariant n'identifie plus de contraintes violées. De cette façon, il est garanti que les restructurations spécifiées préservent la consistance. La figure 2.22 illustre le processus proposé par les auteurs. L'approche nécessite de méta-modéliser le langage analysé, à l'instar des travaux de Clarisó et al. avec OCL [32].

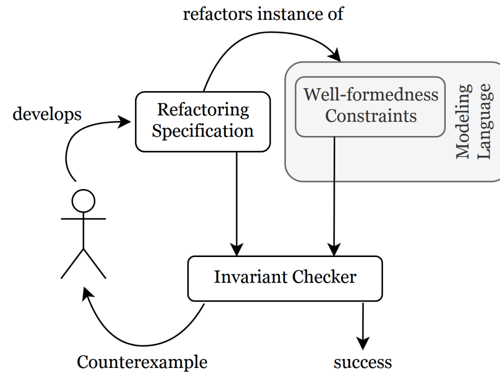


FIGURE 2.22 – Développement itératif d'une règle de restructuration [21]

Jalote et al. [66] proposent une approche de développement itératif qui permet l'exploitation des rétroactions d'une itération afin de prévenir les erreurs dans les itérations qui suivent. Après chaque itération, un processus d'analyse est effectué pour identifier l'origine des erreurs. Le développement itératif dans ces travaux a pour but d'identifier des problèmes, similairement au prouveur Small-t$\mathcal{ALC}$ qui ne fournit qu'un contre-exemple comme rétroaction et contrairement à notre analyseur statique qui suggère, après chaque itération, de nouvelles formules à prendre en compte en tant que préconditions.

2.5 Conclusion

Ce chapitre présente une analyse statique en mode régressif d'une règle Small-t$\mathcal{ALC}$ afin de suggérer, à partir d'un code de transformation et d'une postcondition, des préconditions qui rendent la règle correcte. Cette analyse propage, après chaque instruction, des formules $\mathcal{ALCQI}$, à la base non-closes par substitution et essentiellement universellement quantifiées. Chaque formule est ensuite simplifiée en se basant sur un calcul d'alias qui identifie les variables possiblement équivalentes. Ce calcul est exploité afin d'offrir aux développeurs plusieurs choix de formules, chacune plus forte que la plus faible précondition et conduisant à un triplet correct.

L'interactivité proposée par notre analyseur permet au développeur de construire sa règle d'une manière itérative en la révisant à chaque itération. Le développeur peut ainsi sélectionner une des propositions extraites comme précondition ou modifier sa règle.

Chapitre 3

Vérification *TBox* des règles Small-t$\mathcal{ALC}$

Sommaire

3.1 Visions *ABox* et *TBox* d'un graphe Small-t$\mathcal{ALC}$ 90

3.2 Inférence des propriétés globales Small-t$\mathcal{ALC}$ 92

3.2.1 Interprétation abstraite et règle Small-t$\mathcal{ALC}$ 93

3.2.2 Sémantique abstraite Small-t$\mathcal{ALC}$ 97

3.2.3 Contrats de transformation 98

3.2.4 Formules *TBox* 99

3.3 Analyse statique par interprétation abstraite 100

3.3.1 Processus de l'analyseur statique *TBox* 101

3.3.2 Interprétation abstraite des instructions Small-t$\mathcal{ALC}$ 102

3.3.3 Exemple 109

3.4 Relation entre les niveaux *ABox* et *TBox* 111

3.4.1 Dépendance *ABox* et *TBox* 111

3.4.2 Complémentarité des deux niveaux de vérification 112

3.4.2.1 Formules *TBox* vérifiées, règles *ABox* incorrectes 113

3.4.2.2 Règles *ABox* prouvées correctes, formules *TBox* invalides 114

3.5 Propriétés du second ordre 115

3.6 Travaux similaires 118

3.7 Conclusion 120

La vérification des règles Small-t$\mathcal{ALC}$ avec la logique de Hoare garantit la bonne transformation des sommets et arcs spécifiés par les pré- et postconditions. Les instructions de ces règles expriment une mise à jour de la base de connaissances en ajoutant et supprimant aux concepts et rôles des sommets et arcs respectivement. Ainsi, une règle Small-t$\mathcal{ALC}$ affecte la fonction d'interprétation des concepts *TBox*. Nous souhaitons étudier l'effet des appels des règles sur la *TBox* en exploitant ses axiomes pour exprimer des propriétés sur les ensembles des sommets et arcs. Cela permet de vérifier le graphe en inférant des propriétés globales *TBox* à partir des propriétés locales explicites *ABox* d'une règle. La vérification *TBox* consiste à vérifier des états en des points donnés du programme. Elle se base sur une sémantique abstraite des instructions *ABox*, naturellement présente dans notre langage afin de profiter d'une vérification à un niveau abstrait, en sus de la vérification *ABox* des règles présentée au chapitre précédent. S'appuyant sur la vision *TBox* d'un graphe Small-t$\mathcal{ALC}$, l'analyse statique par interprétation abstraite permet ainsi d'inférer une formule *TBox* révélant la postcondition à partir d'une précondition. Cette étude fait l'objet des trois premières sections 3.1, 3.2 et 3.3 où nous montrons comment complémenter le niveau local de vérification d'une règle *ABox* par un niveau global *TBox* d'un enchaînement de règles concernant la transformation dans son ensemble. Le chapitre se poursuit par la section 3.4 qui montre la relation entre les deux niveaux de vérification *ABox* et *TBox*. Il se conclut, à la section 3.5, par la capacité des formules *TBox* Small-t$\mathcal{ALC}$ à exprimer certaines propriétés qui relèvent de la logique monadique du second ordre.

3.1 Visions *ABox* et *TBox* d'un graphe Small-t$\mathcal{ALC}$

Concrètement, un graphe est constitué d'un ensemble de sommets reliés par des arcs. Ces sommets et arcs sont étiquetés par des concepts et rôles respectivement, conformément au niveau *ABox* des logiques de description. En fait, les assertions *ABox* de la logique permettent de décrire des propriétés des individus ; elles spécifient les concepts d'appartenance de ces individus et les relations binaires entre eux par le biais de rôles. Ces propriétés *ABox* sont spécifiées par les pré- et postconditions des règles Small-t$\mathcal{ALC}$ et manipulées par l'intermédiaire des instructions. La figure 3.1 illustre l'exemple d'un graphe dans le domaine de la zoologie où les sommets (instances) représentent un lion, une girafe, un panda, des feuilles d'acacia et des pousses de bambou. Le lion est de concept *Carnivore* ; la girafe et le panda sont de concept *Herbivores*. Tous les trois appartiennent au concept *Animal*. Les sommets représentant les feuilles d'acacia et les pousses de bambou appartiennent au concept *Végétal*. Le mode alimentaire d'un animal prend ici la forme d'un arc étiqueté du rôle *consomme* : un lion *consomme* une girafe, une girafe *consomme* des feuilles d'acacia et un panda se nourrit exclusivement de pousses de bambou.

Le niveau *TBox* introduit les définitions de concepts et de rôles. Il permet ainsi de décrire les connaissances générales du domaine et d'en avoir une vision plus abstraite que le niveau *ABox*.

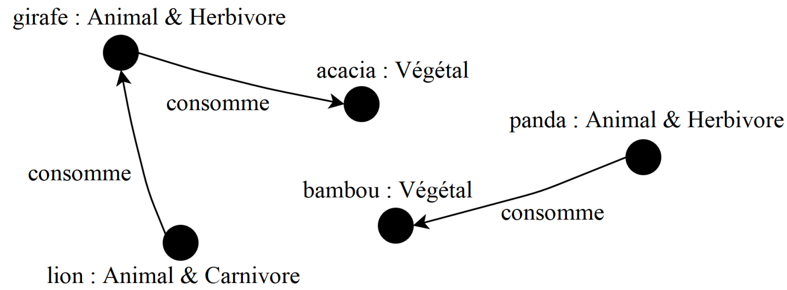


Figure 3.1 – Un graphe *ABox* illustrant un exemple dans le domaine de la zoologie

Ce niveau terminologique axiomatise des propriétés sur les concepts et rôles notamment la relation d'inclusion de concepts. C'est ainsi qu'on considère à la figure 3.2 le même exemple de graphe où sont mentionnés uniquement les ensembles manipulés conformément au niveau terminologique. Ces ensembles représentent les concepts atomiques *Animal, Herbivore, Carnivore* et *Végétal*, et d'autres plus complexes pour exprimer des restrictions sur le rôle *consomme*. Par exemple, on note par $\exists$ *consomme Animal* l'ensemble des animaux dont le régime alimentaire est basé sur la consommation de chair animale. Cet ensemble est bien évidemment équivalent au concept *Carnivore*. D'autre part, $\exists\, consomme^{-1}\ Herbivore$ traduit l'ensemble des plantes consommées par les *Herbivores*. Cet ensemble est ainsi un sous-ensemble du concept *Végétal*.

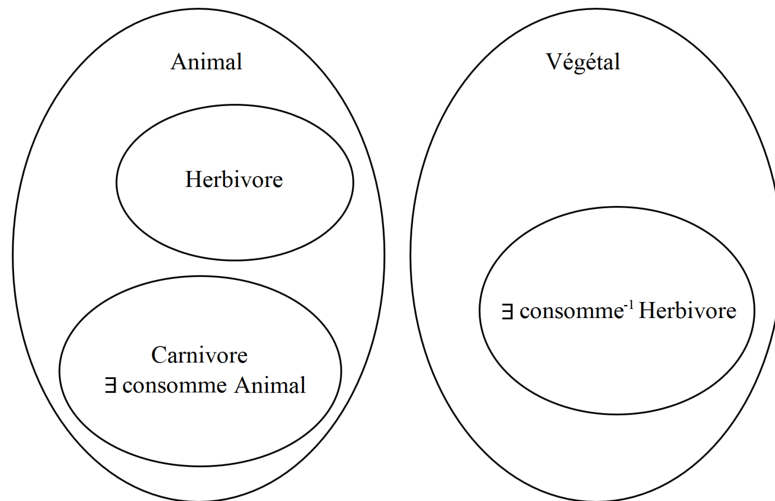


Figure 3.2 – Vision *TBox* du graphe illustrant l'exemple dans le domaine de la zoologie

En Small-t$\mathcal{ALC}$, les concepts sont implicitement introduits dans les préconditions, postconditions et instructions *ABox* des règles. Ces règles ne permettent pas d'exprimer des propriétés terminologiques et donc de décrire l'état global du graphe et son évolution en considérant uniquement les concepts des sommets et les rôles manipulés. Le composant *TBox* des logiques de description est ainsi exploité pour vérifier de telles propriétés sur un graphe, en invoquant les services d'inférence offerts par les logiques de description. Dans notre approche, il s'agit de déduire des propriétés implicites *TBox* à partir des propriétés explicites *ABox*.

Deux types de propriétés peuvent êtres exprimées sur un graphe Small-t$\mathcal{ALC}$ traduisant un domaine : des propriétés locales portant sur les éléments du graphe concret, c'est-à-dire les sommets et les arcs, et des propriétés globales portant sur des ensembles exprimant les concepts d'appartenance des sommets du graphe concret formant ainsi un graphe abstrait. Ce dernier associe chaque sommet de l'*ABox* à un ensemble de la *TBox*. Le graphe abstrait correspondant au graphe concret de la figure 3.1 est donné à la figure 3.3.

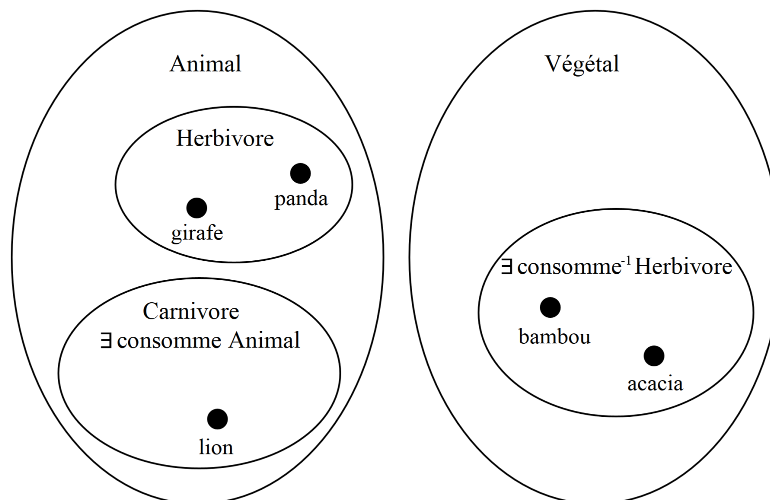


FIGURE 3.3 – Graphe abstrait correspondant au graphe concret de la figure 3.1

Cette interprétation du graphe concret permet de vérifier des propriétés globales exprimant des relations de subsomption sur les ensembles d'un graphe telles que $Herbivore \cup Carnivore \subseteq Animal$ et $Carnivore = \exists\ consomme\ Herbivore$ dans cet exemple.

3.2 Inférence des propriétés globales Small-t$\mathcal{ALC}$

Outre le niveau *ABox* manipulé explicitement par les règles Small-t$\mathcal{ALC}$, la section 3.2.1 montre que le niveau *TBox* des logiques de description naturellement présent dans le langage permet

d'exploiter la notion d'interprétation abstraite afin d'inférer des propriétés globales à partir des propriétés locales d'un programme. Dans ce cadre, la sémantique abstraite, une base essentielle de l'interprétation abstraite, est présentée à la section 3.2.2. La démarche de vérification des propriétés globales s'inspire de la programmation par contrat comme le montre la section 3.2.3. Ce paradigme sert à spécifier une règle par un contrat composé d'une pré- et postcondition dont la syntaxe dans notre contexte est donnée à la section 3.2.4.

3.2.1 Interprétation abstraite et règle Small-t$\mathcal{ALC}$

L'interprétation abstraite est une théorie de l'approximation de la sémantique d'un langage utilisée pour l'analyse et la vérification des programmes [42, 39]. Elle permet de formaliser l'idée qu'une sémantique est plus ou moins précise selon le niveau d'observation auquel on se place. Les sémantiques s'ordonnent ainsi en un treillis complet selon la relation de précision [43, 41]. En définissant une sémantique abstraite plus large que la sémantique concrète [40], une interprétation abstraite est capable de démontrer, par analyse statique, des propriétés données pour un sous-ensemble de programmes vérifiant celles-ci [110]. La plupart de ces propriétés est souvent indécidable à cause de la complexité de la sémantique formelle des langages. La définition d'une sémantique abstraite décidable permet de calculer une sur-approximation de l'ensemble des états atteignables par un programme. En pratique, une interprétation abstraite est constituée : des données concrètes, des opérations sur les données concrètes, des données abstraites et des opérations abstraites [121]. Elle est formalisée par une fonction d'abstraction et offre un cadre théorique pour définir une telle sémantique. La figure 3.4 montre un ensemble R non-calculable des états atteignables par un programme. L'interprétation abstraite permet de sur-approximer cet ensemble R par un ensemble A qui doit être suffisamment précis pour prouver l'absence d'erreurs. C'est ainsi que l'intersection vide entre A et l'ensemble d'états erronés E_2 prouve que les états de E_2 sont inatteignables. Par contre, l'intersection non-vide entre A et E_1 montre que la sur-approximation A de R n'est pas suffisamment précise pour prouver que les états de E_1 sont inatteignables.

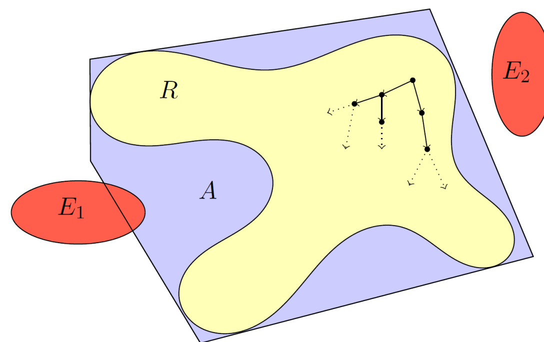


Figure 3.4 – Approximation A de l'ensemble R non-calculable des états atteignables [63]

Notre langage Small-t$\mathcal{ALC}$ respecte naturellement deux niveaux de formalisation : le niveau *ABox* qui permet de manipuler des configurations de sommets et d'arcs correspond à la sémantique concrète, et le niveau *TBox* qui abstrait la sémantique concrète en négligeant ces configurations correspond à la sémantique abstraite. Autrement dit, l'*ABox* définit les données concrètes, la *TBox* définit les données abstraites et les instructions des règles *ABox* constituent les opérations sur les données concrètes. Cependant, dans notre contexte, les opérations abstraites sont définies implicitement au travers des concepts associés aux nœuds ; notre raisonnement infère des données abstraites à partir des données concrètes en considérant l'effet direct des instructions *ABox* sur les données inférées. Ainsi, notre fonction d'abstraction est telle qu'elle associe à tout sommet d'un graphe son concept d'appartenance déclaré implicitement par la règle. La figure 3.5 montre la vision Small-t$\mathcal{ALC}$ des deux niveaux *ABox* et *TBox* : le développeur code des transformations au niveau *ABox* en écrivant des règles Small-t$\mathcal{ALC}$ pour manipuler les sommets et arcs d'un graphe, puis à partir de ces règles, des propriétés globales *TBox* sont déduites pour décrire les relations de subsomption entre les concepts d'appartenance des sommets définis implicitement par les règles. Ces propriétés sont exprimables par défaut dans le langage Small-t$\mathcal{ALC}$ en exploitant le niveau *TBox*, sans avoir besoin de sur-approximer la sémantique concrète explicitement. C'est ainsi que du point de vue de l'interprétation abstraite, l'inférence des propriétés *TBox* s'apparente à une approximation conservative.

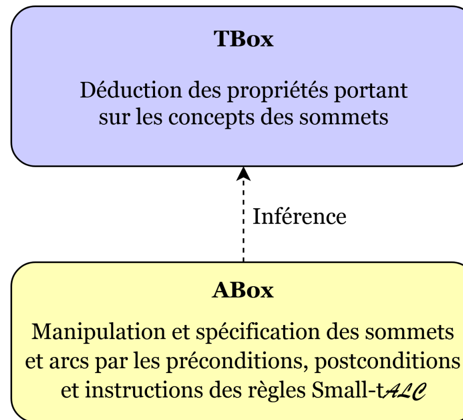


FIGURE 3.5 – Vision *ABox*/*TBox* de Small-t$\mathcal{ALC}$

Dans ce contexte, on cherche à identifier l'état du graphe abstrait avant et après des transformations en négligeant les instances manipulées et en considérant uniquement les ensembles d'instances grâce à une sémantique abstraite basée sur le niveau terminologique. Cela nous permet de vérifier des propriétés relatives à ces ensembles affectés par les instructions Small-t$\mathcal{ALC}$ manipulant des sommets y appartenant. Il en résulte un affaiblissement de la sémantique concrète afin de procéder

à la vérification au niveau abstrait *TBox*, après avoir effectué une vérification plus fine au niveau concret *ABox* grâce à la logique de Hoare. Par exemple, en affirmant que toutes les relations r sortantes de toutes les instances de concept A conduisent vers les instances de concept B, on ne peut pas déterminer quelle instance de A est reliée par r à celle de B. On exploite ainsi cette théorie d'interprétation abstraite pour analyser les programmes Small-t$\mathcal{ALC}$ afin d'inférer des propriétés complémentaires du graphe.

Dès lors, on classifie les sommets d'un graphe selon les étiquettes associées à tout sommet et à tout arc d'un graphe : l'étiquette d'un sommet désigne son concept d'appartenance ; l'étiquette d'un arc spécifie la nature du lien entre son sommet source et son sommet cible. De plus, le graphe étant orienté, les arcs donnent lieu à deux catégories distinctes d'ensembles : celle des sommets sources et celle des sommets cibles. Ces ensembles sont atteignables en utilisant les restrictions universelles et existentielles ainsi que les rôles inverses des constructeurs *TBox*. L'ensemble des sommets d'un graphe peut dès lors être partitionné en ensembles de sommets ayant même étiquette, et des propriétés portant sur ces ensembles peuvent ainsi être vérifiées par une infinité de programmes qui transforment différentes configurations de graphes.

Considérons les deux programmes $P1$ et $P2$ de la figure 3.6. Le premier programme redirige les arcs r sortants de l'instance a vers des instances de concept C à des instances de concept B. Le deuxième ajoute un arc r entre l'instance a de concept A et une instance b de concept B. On suppose que les graphes sources donnés à la figure 3.7 sont des modèles de la précondition de chaque triplet : le graphe de la première règle est composé d'une seule instance a de concept A liée par r à trois instances de concept C, et le graphe de la deuxième est composé d'une instance a de concept A qui n'a aucun arc sortant. En exécutant chaque programme sur le graphe source qui correspond à sa précondition, on constate qu'ils donnent le même résultat au niveau $TBox$: les arcs r sortants des instances de concept A se dirigent uniquement vers des instances de concept B. Les graphes cibles *ABox* des figures 3.8a et 3.8b respectent $A \subseteq (only\ r\ B)$ la propriété commune : A est un

```
pre: a : A and a : (≥ 3 r C);
inv : a : A;
while (a : (≥ 1 r C)) do {
    select c with a r c and c : C;
    delete(a r c);
    select b with b : B;
    add(a r b);
}
post: a : A and a : (≥ 1 r B);
```

P1

```
pre: a : A;
select b with b : B;
add(a r b);
post: a : A and b : B and a r b;
```

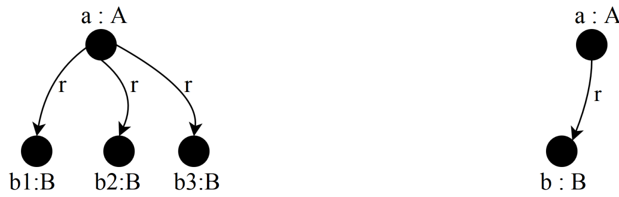
P2

FIGURE 3.6 – Deux programmes de transformation distincts vérifiant $A \subseteq (only\ r\ B)$

(a) Graphe source du programme $P1$

(b) Graphe source du programme $P2$

FIGURE 3.7 – Modèles de graphes sources des programmes $P1$ et $P2$



(a) Graphe cible du programme $P1$

(b) Graphe cible du programme $P2$

FIGURE 3.8 – Modèles de graphes cibles des programmes $P1$ et $P2$

sous-ensemble de l'ensemble des sommets ayant des arcs sortants r se dirigeant vers des sommets de concept B uniquement. Le graphe traduisant cette propriété est illustré à la figure 3.9. Dans ce graphe, les deux ensembles B et ($only\ r\ B$) peuvent avoir une intersection non vide dans le cas où B contient au moins un sommet ayant tous les arcs r se dirigeant vers des sommets de B.

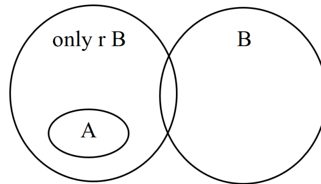


FIGURE 3.9 – Le graphe *TBox* traduisant la propriété $A \subseteq (only\ r\ B)$

Ces propriétés ensemblistes permettent de raisonner sur des transformations en négligeant la notion d'instance et en s'intéressant uniquement à des ensembles de sommets d'un graphe. Ce mode de raisonnement s'applique ainsi à une famille de programmes partageant les mêmes propriétés ensemblistes.

3.2.2 Sémantique abstraite Small-t$\mathcal{ALC}$

A l'instar des logiques de description, la sémantique du langage Small-t$\mathcal{ALC}$ fait intervenir une interprétation $\mathcal{I}$ où les sommets d'un graphe sont des éléments du domaine $\Delta^{\mathcal{I}}$. Les instructions Small-t$\mathcal{ALC}$ permettent ainsi concrètement d'ajouter et de supprimer des sommets et des arcs appartenant à $\Delta^{\mathcal{I}}$ et $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$ respectivement. Les assertions d'appartenance à une *ABox* exploitées dans les pré- et postconditions sont $a : C$ pour exprimer que a appartient à un concept C de la $TBox$ associée, et $a\ r\ b$ pour exprimer que (a, b) appartient au rôle r de la $TBox$. L'interprétation $\mathcal{I}$ satisfait ces deux assertions si et seulement si $a^{\mathcal{I}} \in C^{\mathcal{I}}$ et $(a^{\mathcal{I}}, b^{\mathcal{I}}) \in r^{\mathcal{I}}$ respectivement.

$\mathcal{I}$ associe également à chaque concept un sous-ensemble de $\Delta^{\mathcal{I}}$: le concept C est associé à $C^{\mathcal{I}}$ avec $C^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$ et r est associé à $r^{\mathcal{I}}$, l'ensemble des couples connectés par la relation r, avec $r^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$. Les autres constructeurs *TBox* s'interprètent classiquement comme suit :

$$
\begin{aligned}
&(\top)^{\mathcal{I}} = \Delta^{\mathcal{I}} \\
&(\bot)^{\mathcal{I}} = \emptyset \\
&(\neg C)^{\mathcal{I}} = \Delta^{\mathcal{I}} \backslash C^{\mathcal{I}} \\
&(C1 \cup C2)^{\mathcal{I}} = (C1)^{\mathcal{I}} \cup (C2)^{\mathcal{I}} \\
&(C1 \cap C2)^{\mathcal{I}} = (C1)^{\mathcal{I}} \cap (C2)^{\mathcal{I}} \\
&(\exists\ r\ C)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} |\ \{\exists y.(x, y) \in r^{\mathcal{I}}\} \wedge y \in C^{\mathcal{I}}\} \\
&(\forall\ r\ C)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} |\ \forall y.(x, y) \in r^{\mathcal{I}} \Rightarrow y \in C^{\mathcal{I}}\} \\
&(\leq n\ r\ C)^{\mathcal{I}} = \{x, y \in \Delta^{\mathcal{I}} |\ |(x, y) \in r^{\mathcal{I}}| \leq n\} \\
&(\geq n\ r\ C)^{\mathcal{I}} = \{x, y \in \Delta^{\mathcal{I}} |\ |(x, y) \in r^{\mathcal{I}}| \geq n\} \\
&(r^{-1})^{\mathcal{I}} = \{(y, x) |\ (x, y) \in r^{\mathcal{I}}\}
\end{aligned}
$$

La $TBox$ définit des axiomes terminologiques exprimant des relations de subsomption entre les concepts de la forme $C \subseteq D$. L'interprétation $\mathcal{I}$ satisfait $C \subseteq D$ si et seulement si $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$. Une interprétation $\mathcal{I}$ satisfait une $TBox$ si et seulement si $\mathcal{I}$ satisfait tous les axiomes de la $TBox$. La *TBox* comporte ainsi l'abstraction et la généralisation des instances. Dans ce sens, l'ajout (la suppression) d'un sommet à un (d'un) concept C affecte directement le sous-ensemble $C^{\mathcal{I}}$ de $\Delta^{\mathcal{I}}$, ainsi que les axiomes portant sur C. Également, l'ajout (la suppression) d'un arc étiqueté r affecte directement le sous-ensemble $r^{\mathcal{I}}$ de $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$ ainsi que les axiomes portant sur r.

D'un point de vue pragmatique, la sémantique abstraite Small-t$\mathcal{ALC}$ est celle des constructeurs *TBox*. Les pré- et postconditions des règles Small-t$\mathcal{ALC}$ expriment des assertions *ABox* sur les sommets et arcs d'un graphe et les instructions les manipulent directement. Dans ce contexte, les axiomes *TBox* sont exploités pour établir une vérification à un niveau abstrait en exprimant des propriétés ensemblistes. Ces propriétés décrivent des relations entre des concepts en se basant sur les concepts et les rôles manipulés implicitement par les corps des règles Small-t$\mathcal{ALC}$.

Dans ce cadre, outre la vérification fine des transformations *ABox* par la logique de Hoare, la vérification *TBox* s'intéresse aux propriétés ensemblistes des sommets d'un graphe en formulant des contrats avant et après les appels des règles dans le point d'entrée d'un programme Small-t$\mathcal{ALC}$.

3.2.3 Contrats de transformation

La programmation par contrat a été introduite par Bertrand Meyer [89] avec le langage Eiffel [90]. C'est un paradigme de programmation établi entre un client et son fournisseur visant à s'assurer que les clauses d'utilisation d'un service sont respectées. La notion de contrat est ainsi une métaphore conceptuelle des contrats industriels : si le client respecte la précondition avant l'appel du service, alors la postcondition sera satisfaite par le fournisseur après l'appel. Considérons l'exemple d'un voyage en train : si le client achète son billet, le composte et monte dans le train correspondant avant l'heure de départ, alors il est assuré d'arriver à destination à l'heure prévue. Dans un programme, ces contrats de transformation prennent la forme d'annotations traduisant des assertions.
En s'inspirant de cette notion de contrat, les assertions *TBox* Small-t$\mathcal{ALC}$ sont exploitées pour :

1. exprimer, conformément au paradigme traditionnel, un contrat attaché à une règle et spécifié par des pré- et postconditions *TBox*,
2. refléter des états de calcul en des points donnés du programme afin d'assurer le bon enchaînement des règles et vérifier leur effet global *TBox* sur le graphe abstrait,
3. traduire des invariants du programme prenant la forme d'une même propriété *TBox* valide entre deux points d'exécution du code.

Ces assertions sont attachées aux appels des règles dans le point d'entrée d'un programme Small-t$\mathcal{ALC}$; la première forme est associée à un appel simple ou itératif, et les deux autres à toute séquence d'appels.

```
rule delete_r {
  while(a : (≥ 1 r A)) do {
    select n with (a r n) and (n : A);
    delete(a r n);
  }
}

main {
 assert: (only r- A) ⊆ A|B;
 delete_r!;
 assert: (only r- A) ⊆ B;
}
```

Figure 3.10 – Contrat *TBox* de la règle *delete_r*

Considérons à la figure 3.10 la règle *delete_r* qui supprime tous les arcs r sortants d'un sommet a de concept A vers d'autres sommets de concept A. L'appel itératif de cette règle dans le point d'entrée du programme entraîne la suppression de tous les arcs reliant les sommets de concept A. Les assertions *TBox* de cet exemple respectent la forme d'un contrat traditionnel. La précondition *TBox* de l'appel *delete_r*! dans le *main* spécifie que tous les arcs sortants des sommets de concept A se dirigent vers des sommets de concept A ou de concept B, soit (*only* r- A) $\subseteq A|B$. Cette propriété est relative au graphe source avant transformation. Après l'application itérative de la règle *delete_r*, le graphe cible est tel que tous les arcs sortants des sommets de concept A se dirigent vers des sommets de concept B uniquement, soit (*only* r- A) $\subseteq B$. Ces états de calcul expriment un contrat de transformation : si l'assertion *TBox* avant l'appel itératif de la règle *delete_r* (la précondition) est respectée par le graphe source, alors l'assertion *TBox* après l'appel (la postcondition) doit l'être. Dans ce qui suit, on appelle *TFormules* ces assertions *TBox*.

3.2.4 Formules *TBox*

En Small-t$\mathcal{ALC}$, les axiomes de la *TBox* sont dénommés *TFaits* et ont pour syntaxe :

TFait := *Concept* '=' *Concept*
| *Concept* '⊆' *Concept*

A ce niveau de vérification *TBox*, au-delà des constructeurs faisant partie de Small-t$\mathcal{ALC}$, le caractère impératif du langage nécessite de référencer les ensembles avant leur mise à jour par les appels des règles. On introduit ainsi *init.C* pour désigner l'ensemble initial des instances d'un concept C avant tout appel de règle dans le *main* et *old.C* pour désigner l'ensemble C avant sa dernière mise à jour. Ce dernier est équivalent à l'expression *old* du langage *Eiffel* [90]. A noter qu'il n'est pas autorisé syntaxiquement à utiliser ces deux constructeurs dans la première *TFormule* d'une transformation.

Une *TFormule* exprimant une assertion dans le point d'entrée du programme est une combinaison booléenne de *TFaits* :

TFormule := *TFait*
| '(' *TFormule* ')'
| '!' '(' *TFormule* ')'
| *TFormule* "and" *TFormule*
| *TFormule* "or" *TFormule*

Pour les identifier, les *TFormules* sont précédées du mot-clé *assert*. Le point d'entrée *main* d'un programme Small-t$\mathcal{ALC}$ respecte ainsi la syntaxe :

main := ["assert" ' :' *TFormule* ' ;'] {*nom_règle* [!] ' ;' ["assert" ' :' *TFormule* ' ;']}*

```
main {
  assert : A&B = empty and C = D;
  r1;
  r2;
  assert : A&B ⊆ C and C = D;
  r3;
  assert : A&B ⊆ C and D = empty;
}
```

FIGURE 3.11 – Séquence d'appels de règles avec des contrats *TBox*

Ces *TFormules* permettent d'une part de vérifier l'effet d'une règle sur le graphe abstrait et d'autre part de vérifier que l'état après l'appel d'une première règle correspond à l'état requis avant l'appel de la règle qui suit. Par exemple, soit le point d'entrée Small-t$\mathcal{ALC}$ de la figure 3.11 composé d'une séquence d'appels de règles avec des contrats *TBox*. Cette séquence comporte deux contrats *TBox* : le premier où $F1 = (A\&B = empty\ and\ C = D)$ constitue la précondition et $F2$ la postcondition de la séquence d'appels $(r1; r2)$ et le deuxième conforme à la notion conventionnelle des contrats où $F2 = (A\&B \subseteq C\ and\ C = D)$ constitue la précondition et $F3 = (A\&B \subseteq C\ and\ D = empty)$ la postcondition de la règle $r3$. Soulignons que $F2$ joue à la fois le rôle d'une pré- et postcondition de deux contrats distincts. $F1$ indique que les deux concepts A et B sont disjoints et que C et D ont exactement les mêmes nœuds. $F2$ affirme que l'intersection entre A et B constitue désormais un sous-ensemble de C. $F3$ exprime que le concept D est devenu vide.

Afin d'étudier l'effet des appels sur les *TFormules*, une analyse statique est effectuée permettant d'exécuter symboliquement les instructions des règles afin d'inférer des propriétés *TBox* consistantes avec les propriétés *ABox* des instances manipulées par les règles appelées.

3.3 Analyse statique par interprétation abstraite

Les instructions d'une règle Small-t$\mathcal{ALC}$ manipulent les sommets et arcs d'un graphe, c'est-à-dire le niveau *ABox* de la logique. Opérationnellement, une règle Small-t$\mathcal{ALC}$ définit une nouvelle interprétation en ajoutant et supprimant des éléments dans les sous-ensembles désignant les concepts et les rôles. Ces mises à jour *ABox* impactent directement les relations de subsomption *TBox*. L'analyseur statique *TBox* que nous proposons permet, à partir d'une *TFormule* donnée par le développeur, de raisonner sur un programme pour déduire des propriétés *TBox* de la transformation. L'analyseur infère ces propriétés à partir des propriétés *ABox* explicites d'une règle.

Si une assertion *TBox* est valide à l'issue de l'exécution d'une règle *ABox*, alors cette assertion est consistante avec le niveau *ABox* de la règle (principe de réflection). Inversement, si l'exécution d'une règle *ABox* valide localement ses prémisses et sa conclusion, alors l'état abstrait du graphe cible peut se caractériser par des assertions *TBox* (principe de préservation). Ces propriétés de

réflection et de préservation sont telles que nous respectons le caractère monotone du raisonnement logique. En outre, la véracité d'une inclusion cible *TBox* n'est établie que si elle n'implique pas de contradiction avec la base de connaissances, conformément à l'hypothèse du monde ouvert des logiques de description. Dès lors, l'inférence de telles propriétés après une séquence de mises à jour entraîne la vérification qu'une interprétation exprimée par un graphe est un modèle de l'*ABox* et la *TBox*.

La section 3.3.1 présente le processus de l'analyseur statique *TBox* de Small-t$\mathcal{ALC}$ qui exécute symboliquement les instructions des règles appelées. L'effet de cette exécution sur les faits d'une formule est détaillé à la section 3.3.2. Il s'agit d'interpréter abstraitement les instructions Small-t$\mathcal{ALC}$ au regard de la sémantique des constructeurs *TBox*. Un exemple illustrant l'évolution d'une précondition *TBox* après la qualification d'un sommet et l'ajout d'un arc au graphe source est proposé à la section 3.3.3 pour terminer cette étude ; la formule *TBox* inférée, qui caractérise le graphe cible, peut ainsi être confrontée à la postcondition du développeur.

3.3.1 Processus de l'analyseur statique *TBox*

Afin de déduire des propriétés $TBox$, un analyseur statique est mis en place pour inférer une *TFormule* à l'issue de l'exécution d'une ou plusieurs règles en se basant sur une *TFormule* fournie par le développeur avant l'appel. Par exemple, étant donnée une *TFormule* composée du *TFait* $C = D$, l'ajout d'une instance i au concept C par l'instruction $add(i : C)$ peut affecter la validité de $C = D$. En effet, si i appartenait préalablement au concept C alors $C = D$ reste valide d'après la théorie des ensembles. Sinon, l'instruction $add(i : C)$ entraîne l'ajout d'un élément de plus au concept C, d'où C devient $C \cup \{i\}$ et $C = D$ se réduit à $C \supseteq D$. L'appartenance de l'individu i au concept C est identifiée dans la précondition de l'instruction $add(i : C)$. Dans cette optique, la précondition *ABox* d'une instruction joue un rôle essentiel pour définir de l'effet de l'instruction sur une *TFormule*.

Rappelons que dans une inférence déductive, la conclusion est une conséquence logique de la prémisse si à chaque fois que la prémisse est vraie, alors la conclusion est vraie. Dans ce cas, l'inférence est dite valide. Inversement, si la prémisse est fausse, la conclusion n'a pas à être considérée. C'est ainsi que l'analyseur statique *TBox* assume la validité de la *TFormule* fournie avant les appels de règles. L'inférence garantit alors la consistance de la base de connaissances transformée $ABox$ au regard de la formule *TBox* inférée. Dans le cas où la *TFormule* initiale n'est pas respectée par un graphe source, la *TFormule* inférée est considérée insatisfiable vis-à-vis du graphe cible. Cette situation est détectée par l'environnement Small-t$\mathcal{ALC}$ qui s'assure, en amont de la transformation du graphe source, que la précondition du contrat de transformation est bien respectée ; ce point sera discuté au dernier chapitre de la thèse.

Ayant une *TFormule* donnée traduisant des égalités et inclusions entre des ensembles et des instructions qui ajoutent et suppriment des éléments appartenant à ces ensembles, l'inférence d'une

nouvelle *TFormule* parcourt symboliquement les instructions d'une règle et analyse l'effet de chacune sur les *TFaits* de la *TFormule* selon sa précondition *ABox*. A cet égard, la *TFormule* évolue éventuellement après chaque instruction.

Considérons l'exemple 3.11 de la section 3.2.4 exprimant une séquence donnée dans un point d'entrée. L'analyseur statique étudie l'effet de la séquence d'appels $(r1; r2)$ sur la formule $F1$ ainsi que l'effet de l'appel $r3$ sur la formule $F2$. Il produit ainsi deux autres formules $F2'$ et $F3'$ qui rendent les propriétés *TBox* exprimées par ces formules et les propriétés *ABox* consistantes. Pour vérifier les formules $F2$ et $F3$ données après les appels des règles, l'analyseur propose les formules extraites au développeur. Les éventuelles contradictions entre $F2$ et $F2'$ ou entre $F3$ et $F3'$ invitent le développeur à revoir le code de ses règles ou ses *TFormules*. Ce cas est détaillé ultérieurement à la section 3.4.2.

3.3.2 Interprétation abstraite des instructions Small-t$\mathcal{ALC}$

Cette section s'intéresse à l'effet des instructions Small-t$\mathcal{ALC}$ sur les *TFaits* pouvant être affectés dans les *TFormules* en prenant en compte les propriétés des instances manipulées. Rappelons qu'une règle Small-t$\mathcal{ALC}$ comporte la précondition *ABox* de la règle. De ce fait, l'analyseur statique *TBox* s'appuie sur l'analyseur statique *ABox* qui permet d'extraire la précondition de chaque instruction révélant les propriétés des instances manipulées. Cela se fait lors du calcul en mode progressif de la plus forte postcondition d'une règle à partir de sa précondition. En effet, la postcondition calculée d'une instruction constitue une précondition de l'instruction qui suit.

L'interprétation des instructions vis-à-vis des *TFaits* est donnée sous la forme de règles d'inférence établies par des arbres de preuve. Ces règles permettent de déduire une formule à partir de n-uplets de formules appelées prémisses. La formule déduite est appelée conclusion. On la représente sous la forme d'un triplet de Hoare $\{P\}S\{Q\}$ où P est le *TFait* donné et Q le *TFait* inféré suite à l'instruction S. Les règles d'inférence ci-dessous correspondent à l'effet de l'instruction $add(i : C)$ sur les *TFaits* $C \subseteq D$, $E = C \cap D$ et $C \cap D = \phi$ respectivement, supposant l'hypothèse $i \in D$ exprimant une propriété *ABox* :

$$\frac{i \in D}{\{C \subseteq D\}\ add(i : C)\ \{C \subseteq D\}}(\text{ADD}_i_1)$$

$$\frac{i \in D}{\{E = C \cap D\}\ add(i : C)\ \{E \subseteq C \cap D\}}(\text{ADD}_i_2)$$

$$\frac{i \in D}{\{C \cap D = \phi\}\ add(i : C)\ \{C \cap D \neq \phi\}}(\text{ADD}_i_3)$$

Considérant la condition *ABox* $i \in D$, la première règle ADD_i_1 montre que $C \subseteq D$ est préservé après l'application de l'instruction $add(i : C)$. ADD_i_2 montre l'inférence de l'inclusion $E \subseteq C \cap D$

à partir de l'égalité $E = C \cap D$. ADD$_i$_3 fixe les conditions pour inférer l'inégalité $C \cap D \neq \phi$ à partir de l'égalité $C \cap D = \phi$. Cette instruction $add(i : C)$ a différents effets sur les *TFaits* présentés si d'autres hypothèses *ABox* sont considérées ; elle peut aussi avoir d'autres effets sur d'autres *TFaits*.

Un arbre de preuve a pour racine une conclusion et pour feuilles des hypothèses. A chacune des instructions de l'analyseur correspond un arbre de preuve qui se caractérise par une suite d'applications de règles d'inférence donnant lieu à une formule qui se représente sous la forme d'une conséquence syntaxique $\vdash$ signifiant *infère*. Pour établir ces preuves, d'autres règles, plus basiques, concernant la théorie des ensembles (ENS), la plus faible précondition (WP) de chaque instruction Small-t$\mathcal{ALC}$, et le renforcement d'une précondition en logique de Hoare (CSQ) sont considérées. Par exemple :

$$\frac{i \in D}{D \cup \{i\} = D}(\text{ENS}_1) \qquad \frac{C \subseteq D}{C \cup \{i\} \subseteq D \cup \{i\}}(\text{ENS}_2)$$

$$\frac{C = D}{C \subseteq D \cup \{i\}}(\text{ENS}_3) \qquad \frac{}{C \cup \{i\} \neq \phi}(\text{ENS}_4)$$

$$\frac{}{\{P[C + i \backslash C]\}\ add(i : C)\ \{P\}}(\text{WP}_{\text{ADD}i})$$

$$\frac{P' \Rightarrow P \qquad \{P\}S\{Q\}}{\{P'\}\ S\ \{Q\}}(\text{CSQ})$$

En se basant sur ces hypothèses, les arbres de preuve en déduction naturelle associées aux règles d'inférence ADD$_i$_1, ADD$_i$_2 et ADD$_i$_3 s'établissent comme suit :

$$\frac{\dfrac{}{C \subseteq D \Rightarrow C \cup \{i\} \subseteq D \cup \{i\}}(\text{ENS}_2) \quad \dfrac{\dfrac{}{i \in D \vdash D \cup \{i\} = D}(\text{ENS}_1) \quad \dfrac{}{\{C \cup \{i\} \subseteq D\}add(i : C)\{C \subseteq D\}}(\text{WP}_{\text{ADD}i})}{i \in D \vdash \{C \cup \{i\} \subseteq D \cup \{i\}\}add(i : C)\{C \subseteq D\}}}{\dfrac{i \in D \vdash \{C \subseteq D\}\ add(i : C)\ \{C \subseteq D\}}{\vdash i \in D \Rightarrow \{C \subseteq D\}\ add(i : C)\ \{C \subseteq D\}}}(\text{CSQ})$$

$$\frac{\dfrac{}{E = C \cap D \Rightarrow E \subseteq (C \cap D) \cup \{i\}}(\text{ENS}_3) \quad \dfrac{\dfrac{\dfrac{}{i \in D \vdash D \cup \{i\} = D}(\text{ENS}_1) \quad \dfrac{}{\{E \subseteq C \cup \{i\} \cap D\}add(i : C)\{E \subseteq C \cap D\}}(\text{WP}_{\text{ADD}i})}{i \in D \vdash \{E \subseteq C \cup \{i\} \cap D \cup \{i\}\}add(i : C)\{E \subseteq C \cap D\}}}{i \in D \vdash \{E \subseteq (C \cap D) \cup \{i\}\}add(i : C)\{E \subseteq C \cap D\}}}{\dfrac{i \in D \vdash \{E = C \cap D\}\ add(i : C)\ \{E \subseteq C \cap D\}}{\vdash i \in D \Rightarrow \{E = C \cap D\}\ add(i : C)\ \{E \subseteq C \cap D\}}}(\text{CSQ})$$

$$\frac{\dfrac{}{(C \cap D) \cup \{i\} \neq \phi}(\text{ENS}_4) \quad \dfrac{\dfrac{\dfrac{}{i \in D \vdash D \cup \{i\} = D}(\text{ENS}_1) \quad \dfrac{}{\{C \cup \{i\} \cap D \neq \phi\}add(i : C)\{C \cap D \neq \phi\}}(\text{WP}_{\text{ADD}i})}{i \in D \vdash \{C \cup \{i\} \cap D \cup \{i\} \neq \phi\}add(i : C)\{C \cap D \neq \phi\}}}{i \in D \vdash \{(C \cap D) \cup \{i\} \neq \phi\}add(i : C)\{C \cap D \neq \phi\}}}{\dfrac{i \in D \vdash \{C \cap D = \phi\}\ add(i : C)\ \{C \cap D \neq \phi\}}{\vdash i \in D \Rightarrow \{C \cap D = \phi\}\ add(i : C)\ \{C \cap D \neq \phi\}}}(\text{CSQ})$$

Soulignons que les conditions portant sur les propriétés *ABox* des instances manipulées sont nécessaires pour appliquer les dérivations. De telle manière est illustrée l'interdépendance des deux niveaux de vérification. Les tableaux de l'analyseur qui suivent implémentent les conditions nécessaires permettant d'inférer les assertions *TBox* à partir des instructions *ABox*. Chacune des décisions de l'analyseur statique faisant l'objet d'un arbre de preuve visant à prouver sa correction est traduite en une ligne des tableaux.

Le tableau 3.1 montre l'inférence des *TFaits* dits *pré-TFaits* après l'instruction $add(i : C)$ en prenant en compte l'union et l'intersection des concepts, ainsi que les restrictions existentielles. La troisième colonne représente les propriétés nécessaires portant sur l'instance i manipulée dans $add(i : C)$. La mention "sinon" implique le cas où les conditions indiquées dans la ligne précédente ne sont pas satisfaites. Le caractère "-" signifie qu'aucune condition n'est requise pour inférer un *TFait* à partir du *pré-TFait* donné. Un *TFait* inféré est désigné par *post-TFait* et il est présenté dans la quatrième colonne. Il existe des cas où aucun *TFait* ne peut être inféré. Ces cas sont ainsi marqués par le symbole X signifiant que le *pré-TFait* en question est supprimé de la *TFormule* lorsqu'on ne peut décider s'il demeure valide suite à l'instruction interprétée. Par exemple, l'interprétation de l'effet de l'instruction $add(i : C)$ sur le *TFait* $C \subseteq D$ en ignorant le concept d'appartenance de l'instance i entraîne la suppression de ce $TFait$. C'est ainsi que l'ignorance des propriétés des instances manipulées aboutit à affaiblir ou éventuellement supprimer les *TFaits* pouvant être affectés par les instructions correspondantes. De telle manière, la base de connaissances composée de l'*ABox* et la *TBox* est toujours consistante.

L'interprétation de l'instruction $delete(i : C)$ est symétrique à celle de l'instruction $add(i : C)$ comme le montre le tableau 3.2. Par contre, ce n'est pas le cas pour l'instruction $add(i\ r\ j)$. En effet, l'ajout d'une relation r entre deux instances affecte uniquement les restrictions existentielles et universelles portant sur le rôle r manipulé comme le montre le tableau 3.3 relatif à l'instruction $add(i\ r\ j)$. Le tableau 3.4 décrit à son tour l'interprétation de l'instruction $delete(i\ r\ j)$, symétrique à $add(i\ r\ j)$.

Conformément aux travaux de Kleymann [67], le fait de dériver une propriété d'une instruction ou d'un programme ne garantit pas la correction de l'analyseur statique Small-t$\mathcal{ALC}$. En fait pour prouver chaque ligne des tableaux présentés, il faut démontrer, à la suite de la dérivation syntaxique $\vdash \{P\}S\{Q\}$, que la proposition $\models \{P\}S\{Q\}$ qui représente la conséquence sémantique est valide. Cela nous conduit à considérer que $\models \{P\}S\{Q\}$ est un nouveau jugement basé sur la mise à jour des états du système. Autrement dit, un programme S appelé dans l'état σ doit se terminer dans l'état τ. On désigne cette relation par $S(\sigma, \tau)$, et on définit $\models \{P\}S\{Q\}$ par :

$$\forall\ \sigma.P(\sigma) \Rightarrow (\exists\ \tau.S(\sigma, \tau) \wedge Q(\tau))$$

La preuve de cette formule est établie par induction en considérant la sémantique opérationnelle de Small-t$\mathcal{ALC}$. Ainsi, en exécutant un programme S dans un état σ, l'état τ sera satisfait à la fin de l'exécution. La réciproque de la correction, appelée complétude, n'a pas été étudiée. Elle vise à montrer que toute conséquence sémantique peut se dériver.

Table 3.1 – Interprétation de l'instruction $add(i : C)$

Instruction	pré-TFait	Condition	post-TFait
$add(i : C)$	$C = empty$	-	$!(C = empty)$
	$!(C = all)$	$i : C$	$!(C = all)$
		sinon	X
	$C = D$	$i : C$	$C = D$
		sinon	$D \subseteq C$
	$C \subseteq D$	$i : C$ *ou* $i : D$	$C \subseteq D$
		sinon	X
	$C \cup D = empty$	-	$!(C \cup D = empty)$
	$!(C \cup D = all)$	$i : C$ *ou* $i : D$	$!(C \cup D = all)$
		sinon	X
	$C \cup D = E$	$i : C$ *ou* $i : D$ *ou* $i : E$	$C \cup E = E$
		sinon	$E \subseteq C \cup D$
	$C \cup D \subseteq E$	$i : C$ *ou* $i : D$ *ou* $i : E$	$C \cup D \subseteq E$
		sinon	X
	$C \cap D = empty$	$i : C$ *ou* $i :!D$	$C \cap D = empty$
		sinon	$!(C \cap D = empty)$
	$!(C \cap D = all)$	$i : C$ *ou* $i :!D$	$!(C \cap D = all)$
		sinon	X
	$C \cap D = E$	$i : C$ *ou* $i : E$ *ou* $i :!D$	$C \cap D = E$
		sinon	$E \subseteq C \cap D$
	$C \cap D \subseteq E$	$i : C$ *ou* $i : E$ *ou* $i :!D$	$C \cap D \subseteq E$
		sinon	X
	$(some\ r\ C) = D$	$i : (= 0\ r\text{-}\ !D)$	$(some\ r\ C) = D$
		sinon	$D \subseteq (some\ r\ C)$
	$(some\ r\ C) \subseteq D$	$i : (= 0\ r\text{-}\ !D)$	$(some\ r\ C) \subseteq D$
		sinon	X
	$(some\ r\text{-}\ C) = D$	$i : (= 0\ r\ !D)$	$(some\ r\text{-}\ C) = D$
		sinon	$D \subseteq (some\ r\text{-}\ C)$
	$(some\ r\text{-}\ C) \subseteq D$	$i : (= 0\ r\ !D)$	$(some\ r\text{-}\ C) \subseteq D$
		sinon	X

TABLE 3.2 – Interprétation de l'instruction $delete(i : C)$

Instructions	pré-TFait	Condition	post-TFait
$delete(i : C)$	$C = all$	-	$!(C = all)$
	$!(C = empty)$	$i : !C$	$!(C = empty)$
		sinon	X
	$C = D$	$i : !C$	$C = D$
		sinon	$C \subseteq D$
	$D \subseteq C$	$i : !C$ *ou* $i : !D$	$D \subseteq C$
		sinon	X
	$C \cup D = all$	$i : D$	$C \cup D = all$
		sinon	X
	$!(C \cup D = empty)$	$i : D$	$!(C \cup D = empty)$
		sinon	X
	$C \cup D = E$	$i : !C$ *ou* $i : !E$ *ou* $i : D$	$C \cup D = E$
		sinon	$C \cup D \subseteq E$
	$E \subseteq C \cup D$	$i : !C$ *ou* $i : !E$ *ou* $i : D$	$E \subset C \cup D$
		sinon	X
	$C \cap D = all$	$i : !D$	$C \cap D = all$
		sinon	X
	$!(C \cap D = empty)$	$i : !D$	$!(C \cap D = empty)$
		sinon	X
	$C \cap D = E$	$i : !C$ *ou* $i : !D$ *ou* $i : !E$	$C \cap D = E$
		sinon	$C \cap D \subseteq E$
	$E \subseteq C \cap D$	$i : !C$ *ou* $i : !D$ *ou* $i : !E$	$E \subseteq C \cap D$
		sinon	X
	$(some\ r\ C) = D$	$i : (=\ 0\ r\text{-}\ D)$	$(some\ r\ C) = D$
		sinon	$(some\ r\ C) \subseteq D$
	$D \subseteq (some\ r\ C)$	$i : (=\ 0\ r\text{-}\ D)$	$D \subseteq (some\ r\ C)$
		sinon	X
	$(some\ r\text{-}\ C) = D$	$i : (=\ 0\ r\ D)$	$(some\ r\text{-}\ C) = D$
		sinon	$(some\ r\text{-}\ C) \subseteq D$
	$D \subseteq (some\ r\text{-}\ C)$	$i : (=\ 0\ r\ D)$	$D \subseteq (some\ r\text{-}\ C)$
		sinon	X

Table 3.3 – Interprétation de l'instruction $add(i\ r\ j)$

Instruction	pré-TFait	Condition	post-TFait
$add(i\ r\ j)$	$(some\ r\ D) = C$	$i\ r\ j\ ou\ i : C\ ou\ j : !D$	$(some\ r\ D) = C$
		$j : D\ et\ i : E$	$(some\ r\ D) \subseteq C \cup E$
		sinon	$C \subseteq (some\ r\ D)$
	$(some\ r\ D) \subseteq C$	$i\ r\ j\ ou\ i : C\ ou\ j : !D$	$(some\ r\ D) \subseteq C$
		$j : D\ et\ i : E$	$(some\ r\ D) \subseteq C \cup E$
		sinon	X
	$(some\ r\text{-}\ C) = D$	$i\ r\ j\ ou\ i : !C\ ou\ j : D$	$(some\ r\text{-}\ C) = D$
		$i : C\ et\ j : E$	$(some\ r\text{-}\ C) \subseteq D \cup E$
		sinon	$D \subseteq (some\ r\text{-}\ C)$
	$(some\ r\text{-}\ C) \subseteq D$	$i\ r\ j\ ou\ i : !C\ ou\ j : D$	$(some\ r\text{-}\ C) \subseteq D$
		$i : C\ et\ j : E$	$(some\ r\text{-}\ C) \subseteq D \cup E$
		sinon	X
	$(only\ r\ D) = C$	$(i\ r\ j\ ou\ i : C)\ et\ j : D$	$(only\ r\ D) = C$
		$i : E\ et\ j : D$	$(only\ r\ D) \subseteq C \cup E$
		$j : D$	$C \subseteq (only\ r\ D)$
		sinon	X
	$(only\ r\ D) \subseteq C$	$(i\ r\ j\ ou\ i : C)\ et\ j : D$	$(only\ r\ D) \subseteq C$
		$i : E\ et\ j : D$	$(only\ r\ D) \subseteq C \cup E$
		sinon	X
	$(only\ r\text{-}\ C) = D$	$(i\ r\ j\ ou\ j : D)\ et\ i : C$	$(only\ r\text{-}\ C) = D$
		$j : E\ et\ i : C$	$(only\ r\text{-}\ C) \subseteq D \cup E$
		$i : C$	$D \subseteq (only\ r\text{-}\ C)$
		sinon	X
	$(only\ r\text{-}\ C) \subseteq D$	$(i\ r\ j\ ou\ j : D)\ et\ i : C$	$(only\ r\text{-}\ C) \subseteq D$
		$j : E\ et\ i : C$	$(only\ r\text{-}\ C) \subseteq D \cup E$
		sinon	X

Vu que l'instruction *select* du langage ne présente qu'une affectation des instances *ABox*, elle n'affecte la validité d'aucun *TFait* de la *TFormule*. L'interprétation de l'instruction if *condition then* $s1$ *else* $s2$ nécessite d'interpréter $s1$ d'une part, et $s2$ d'autre part, puis de bâtir l'union des deux *TFormules* résultantes.

TABLE 3.4 – Interprétation de l'instruction $delete(i\ r\ j)$

Instruction	pré-TFait	Condition	post-TFait
$delete(i\ r\ j)$	$(some\ r\ D) = C$	$i\ !r\ j\ ou\ i : !C\ ou\ j : !D$	$(some\ r\ D) = C$
		sinon	$(some\ r\ D) \subseteq C$
	$C \subseteq (some\ r\ D)$	$i\ !r\ j\ ou\ j\ : !D\ ou\ (i : !C\ et\ j : D)$	$C \subseteq (some\ r\ D)$
		sinon	X
	$(some\ r\text{-}\ C) = D$	$i\ !r\ j\ ou\ i : !C\ ou\ j : !D$	$(some\ r\text{-}\ C) = D$
		sinon	$(some\ r\text{-}\ C) \subseteq D$
	$D \subseteq (some\ r\text{-}\ C)$	$i\ !r\ j\ ou\ i :\ C\ ou\ j : !D$	$D \subseteq (some\ r\text{-}\ C)$
		sinon	X
	$(only\ r\ D) = C$	$i\ !r\ j\ ou\ i : (\geq 2\ r\ D)$	$(only\ r\ D) = C$
		sinon	$(only\ r\ D) \subseteq C$
	$C \subseteq (only\ r\ D)$	$i\ !r\ j\ ou\ i : (\geq 2\ r\ D)$	$C \subseteq (only\ r\ D)$
		sinon	X
	$(only\ r\text{-}\ C) = D$	$i\ !r\ j\ ou\ i : (\geq 2\ r\text{-}\ C)$	$(only\ r\text{-}\ C) = D$
		sinon	$(only\ r\text{-}\ C) \subseteq D$
	$D \subseteq (only\ r\text{-}\ C)$	$i\ !r\ j\ ou\ i : (\geq 2\ r\text{-}\ C)$	$D \subseteq (only\ r\text{-}\ C)$
		sinon	X

Habituellement, lors d'une analyse d'une boucle par interprétation abstraite, un opérateur d'élargissement est défini et appliqué pour assurer la convergence de la boucle en un nombre fini d'étapes [40, 110]. Cet opérateur permet de généraliser le comportement de la première itération de la boucle afin de calculer un point fixe qui désigne une sur-approximation après un nombre d'itérations. Dans notre analyse, l'interprétation de l'instruction *while condition do s* analyse l'effet en une seule itération de s sur la *TFormule* initiale. Pareillement, le corps d'une règle appelée itérativement dans le point d'entrée *main* (avec l'opérateur !) est interprété une seule fois. Vu que les instructions du langage sont limitées par l'ajout d'un élément à un ensemble ou sa suppression, l'analyse des instructions conduit au même constat qu'elles s'appliquent une ou plusieurs fois sur les *TFaits*. Considérons comme exemple simple le *TFait* $A = B$, et une boucle *while* :

```
while (x : (≥ 3 r C)) do {
    select c with x r c;
    add(c : B);
}
```

L'instruction *select* n'affecte pas la validité de ce *TFait*. Par contre, en interprétant l'instruction $add(c : B)$ une première fois, le *TFait* $A = B$ se transforme en $A \subseteq B$. L'ajout de la nouvelle

instance sélectionnée c au concept B dans le corps de la boucle n'affecte pas le *TFait* inféré $A \subseteq B$: l'ensemble d'instances A restera toujours un sous-ensemble de B. Par conséquent, la notion d'opérateur d'élargissement qui assure la terminaison du calcul de la sémantique au prix d'une approximation supplémentaire n'est pas significatif pas dans notre contexte.

L'analyse par interprétation abstraite conventionnelle a pour but d'améliorer la qualité d'un code et minimiser les risques d'erreurs [40, 110], sans exiger aucune spécification ou annotation de la part d'un développeur contrairement à notre approche. De plus, vu que l'interprétation abstraite au sens conventionnel calcule une abstraction des propriétés en sur-approximant la sémantique afin de considérer toutes les exécutions concrètes possibles, des alarmes peuvent être émises signifiant qu'une propriété ne peut être ni confirmée ni infirmée. L'alarme correspondante à une situation qui ne peut jamais apparaître à l'exécution est une fausse alarme qui peut être supprimée manuellement par le développeur. Ces notions d'alarme et de sur-approximation de la sémantique n'existent pas dans notre contexte ; cependant, des incohérences entre le niveaux concret *ABox* et le niveau abstrait *TBox* peuvent se révéler et sont détectées par notre analyse.

3.3.3 Exemple

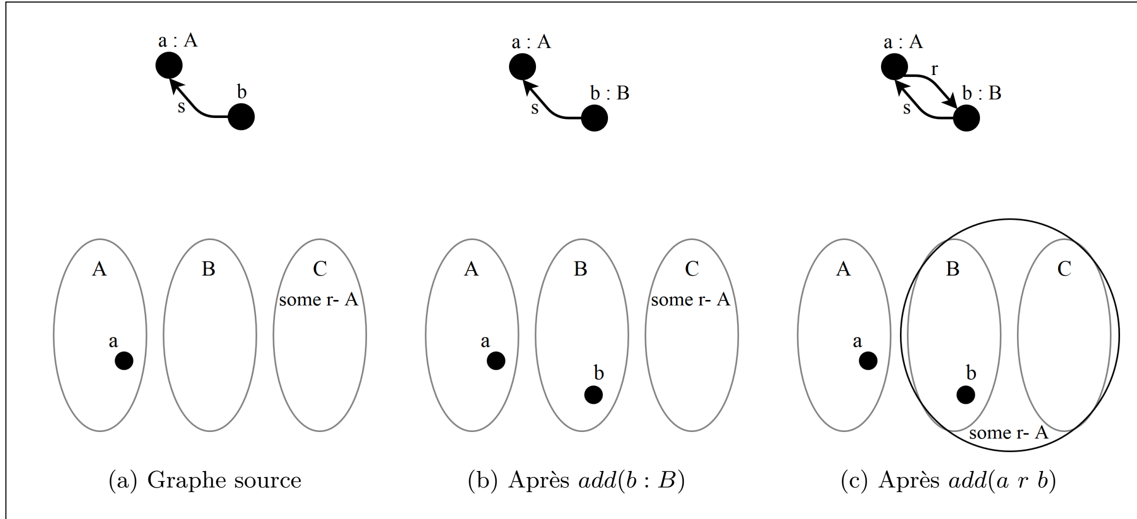
Pour clarifier le processus de l'analyseur statique, considérons à la figure 3.12 la règle *add_r* et son appel dans le point d'entrée *main* d'un programme. Cette règle ajoute l'instance b au concept B et un arc r entre a et b.

```
rule add_r {
 pre: a : A and b s a;
 add(b : B);
 add(a r b);
 post: a : A and b : B and b s a and a r b;
}

main {
  assert: B = empty and (some r- A) = C;
  add_r ;
  assert: !(B = empty) and (some r- A) ⊆ C|B;
}
```

FIGURE 3.12 – Une règle *add_r* et son appel dans le *main*

La figure 3.13 illustre un graphe source conforme à la précondition *ABox* ainsi que son évolution après chaque instruction de la règle *add_r* pour atteindre un graphe conforme à la postcondition *TBox*. L'hypothèse *TBox* donnée sous la forme d'une *TFormule* avant l'appel de la règle *add_r* dans le point d'entrée *main* du programme est telle que le concept B est vide et que tous les arcs r



(a) Graphe source (b) Après $add(b : B)$ (c) Après $add(a\ r\ b)$

FIGURE 3.13 – Évolution des graphes *ABox* et *TBox*

sortants des instances de concept A se dirigent vers les instances du concept C. Le graphe abstrait respectant cette *TFormule* est exposé au-dessous du graphe concret initial associé. L'évolution de ce graphe abstrait après chaque instruction est également présenté dans la même figure.

Afin d'étudier l'effet de la règle *add_r* sur la *TFormule* initiale, ses instructions sont parcourues en prenant en considération la précondition *ABox* de chacune. Considérant la *TFormule* $B = empty\ and\ (some\ r\text{-}\ A) = C$ valide, la figure 3.14 décline la *TFormule* inférée après l'exécution symbolique de chacune des instructions de la règle selon sa précondition *ABox*.

L'effet de la première instruction $add(b : B)$ par rapport au premier *TFait* $B = empty$ de la *TFormule* initiale correspond à la première ligne du tableau 3.1 : ce *TFait* se transforme en

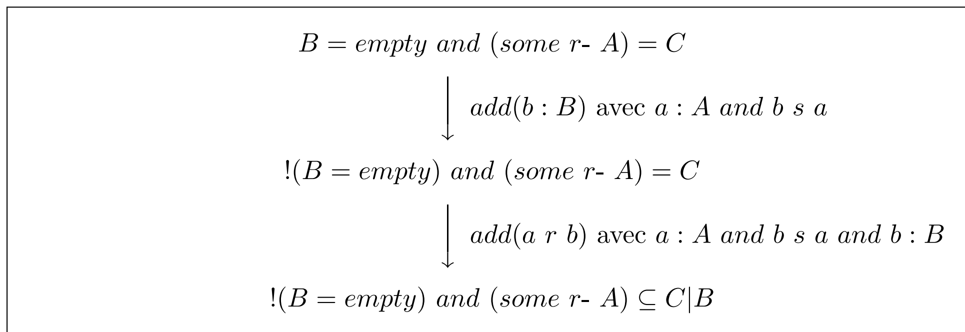


FIGURE 3.14 – Évolution de la *TFormule* après chaque instruction de la règle *add_r*

$!(B = empty)$. Par contre, $add(b : B)$ n'affecte pas la validité du deuxième *TFait* $(some\ r\text{-}\ A) = C$ de la *TFormule* car modifier en extension B ne modifie pas les antécédents de A par r et C. Après la première instruction, la *TFormule* devient $!(B = empty)\ and\ (some\ r\text{-}\ A) = C$.

La deuxième instruction $add(a\ r\ b)$ n'affecte pas le premier *TFait* $!(B = empty)$ de la nouvelle *TFormule*, mais affecte le deuxième *TFait* $(some\ r\text{-}\ A) = C$. Ce dernier cas correspond à la troisième ligne du tableau 3.3 : on étudie l'effet de l'instruction $add(a\ r\ b)$ sur $(some\ r\text{-}\ A) = C$ en ayant $a : A$ et $b : B$. Donc $(some\ r\text{-}\ A) = C$ se transforme en $(some\ r\text{-}\ A) \subseteq C \cup B$. Dès lors, la nouvelle formule obtenue après l'interprétation de la dernière instruction sera $!(B = empty)\ and\ (some\ r\text{-}\ A) \subseteq C \cup B$. Cette *TFormule* est conforme à celle donnée par le développeur ; elle est consistante par rapport aux propriétés *ABox* du programme.

3.4 Relation entre les niveaux *ABox* et *TBox*

La section précédente s'est attachée à décrire le processus de l'analyseur statique *TBox* basé fondamentalement sur une analyse *ABox*. Cela prouve la dépendance entre les deux niveaux de vérification *ABox* et *TBox* comme le montre la section 3.4.1. La vérification *ABox* permet de vérifier localement les transformations des instances manipulées par une règle. Par contre la vérification *TBox* vérifie globalement l'effet des appels des règles sur le graphe abstrait en considérant uniquement des ensembles d'instances du graphe. Ces deux niveaux de vérification sont ainsi complémentaires comme discuté à la section 3.4.2.

3.4.1 Dépendance *ABox* et *TBox*

Le choix d'un raisonnement sur les propriétés *TBox* est tel que l'analyse statique concerne non seulement les instructions de transformation mais aussi sa spécification. En particulier, l'analyseur *TBox* se fonde sur l'analyseur statique *ABox* qui raisonne sur les propriétés *ABox* d'une règle pour identifier la précondition de chaque instruction. De ce fait, l'affaiblissement des spécifications, notamment la précondition d'une règle, entraîne l'impossibilité de prouver la validité de certains *TFaits*. En effet, d'après les tableaux présentés à la section précédente, des conditions de la précondition sont nécessaires à l'inférence des *TFaits* après exécution symbolique de chaque instruction. Dans le cas où ces conditions ne sont pas remplies, le *TFait* est supprimé de la *TFormule* résultante car il devient indécidable.

Considérons la règle *replace_r_s* de la figure 3.15 qui remplace un arc r entre deux instances a et b par s. La précondition indique que le sommet source a de cet arc est de concept A, mais ne spécifie pas le concept du sommet cible b. Le contrat *TBox* de cette règle est formé de la précondition $(some\ s\ B) = empty$ indiquant qu'il n'y a pas d'arcs s se dirigeant vers des sommets de concept B et de la postcondition $(some\ s\ B) \subseteq A$ traduisant que l'ensemble des sommets cibles du rôle s se dirigeant vers des sommets du concept B est un sous-ensemble du concept A.

```
rule replace_r_s {
 pre: a : A and a r b;
 delete(a r b);
 add(a s b);
 post: a : A and a s b;
}

main {
 assert: (some s B) = empty;
 replace_r_s;
 assert: (some s B) ⊆ A;
}
```

FIGURE 3.15 – Exemple de spécifications *ABox* affaiblies

L'analyse statique de la règle *replace_r_s* montre que la première instruction $delete(a\ r\ b)$ n'affecte pas ce *TFait* dès lors qu'il ne concerne pas le rôle r manipulé par l'instruction. L'instruction suivante $add(a\ s\ b)$ est plus problématique. Étant donné que le concept de b n'est pas spécifié par la précondition de la règle, l'analyseur statique supprime ce *TFait* de la *TFormule* car il ne peut ni affirmer sa validité après la règle, ni indiquer son évolution dans le cas contraire. A noter que la règle *replace_r_s* est prouvée correcte au sens de Hoare.

Si le développeur avait indiqué dans sa précondition que le sommet b est de concept B, l'analyseur statique aurait déduit que l'ensemble des sommets cibles du rôle s se dirigeant vers des sommets du concept B est un sous-ensemble du concept A, soit $(some\ s\ B) \subseteq A$, et par suite validé ce *TFait* donné par le développeur sous forme d'une postcondition *TBox*.

Par conséquent, le renforcement de la précondition d'une règle conduit à un diagnostic d'analyse statique *TBox* plus précis, et permet ainsi de prouver plus des propriétés ensemblistes du graphe cible en considérant les propriétés données sur le graphe source.

3.4.2 Complémentarité des deux niveaux de vérification

La vérification d'une règle *ABox* par la logique de Hoare garantit une transformation correcte des instances manipulées par la règle. À un niveau plus abstrait, la vérification *TBox* vérifie l'effet global souhaité d'une ou plusieurs règles sur le graphe abstrait. Ces deux niveaux sont complémentaires. Chacun des deux niveaux vérifie des propriétés non exprimées par l'autre. C'est ainsi qu'on peut identifier deux scénarios : le premier où les formules *TBox* sont vérifiées, mais la preuve *ABox* d'une ou plusieurs règles Small-t$\mathcal{ALC}$ échoue. Le deuxième où toutes les règles *ABox* d'un programme Small-t$\mathcal{ALC}$ sont prouvées correctes, alors que les formules *TBox* ne le sont pas.

3.4.2.1 Formules *TBox* vérifiées, règles *ABox* incorrectes

Vérifier exclusivement que les *TFormules* sont consistantes avec les propriétés *ABox* ne garantit pas que les règles *ABox* sont proprement écrites et correctes. Comme on l'a déjà mentionné, les propriétés ensemblistes exprimées par les *TFormules* permettent de vérifier une famille de programmes et donc ne sont pas limitées à un programme précis. D'où la nécessité de vérifier chaque règle Small-t$\mathcal{ALC}$ séparément en renforçant les spécifications *ABox* pour garantir des résultats tangibles.

Par exemple, considérons la règle *delete_r_nonB* de la figure 3.16 qui supprime l'arc r entre les deux sommets a et x, ce dernier vérifiant $x : !B$. Un invariant *TBox* composé du *TFait* $A = (\textit{some } r\ B)$ est donné pour ce programme, signifiant que tous les sommets de A ont au moins un arc r sortant vers un sommet de B avant et après l'appel de la règle. En pratique, cet invariant considère la même pré- et postcondition *TBox* avant et après l'appel de la règle *delete_r_nonB* dans le point d'entrée du programme.

```
rule delete_r_nonB {
  pre: a r x and x : !B;
  delete(a r x);
  post: a : A and a r x and x : !B;
}
```

FIGURE 3.16 – Une règle *ABox* incorrecte

Dans cet exemple, même en ignorant la vérification au niveau *ABox*, les *TFormules* sont vérifiées sur le graphe abstrait. Par contre, le prouveur Small-t$\mathcal{ALC}$ indique que la règle *delete_r_nonB* est incorrecte et produit le contre-exemple de la figure 3.17. La postcondition n'est pas satisfaite après l'exécution de la règle car le sommet a n'est déclaré de concept A que dans la postcondition. Dès lors, l'application de cette règle entraîne la suppression d'un arc r ayant un sommet source quelconque et non pas un sommet de concept A comme c'est l'intention de la règle.

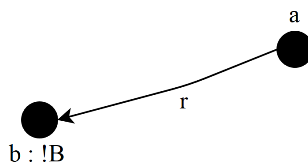


FIGURE 3.17 – Contre-exemple du prouveur indiquant que *delete_r_nonB* est incorrecte

Malgré l'écriture incorrecte de la règle *ABox* *delete_r_nonB*, l'état global du graphe décrit par la propriété *TBox* $A = (\textit{some } r\ B)$ n'est pas affecté. Le *TFait* $A = (\textit{some } r\ B)$ reste toujours

valide après l'appel de *delete_r_nonB*. D'où la nécessité de vérifier, outre le graphe global avec les formules *TBox*, les règles *ABox* pour assurer que la transformation locale des instances est correcte.

3.4.2.2 Règles *ABox* prouvées correctes, formules *TBox* invalides

Étant donné qu'on peut relâcher la spécification d'un triplet de Hoare tout en conservant sa validité, un développeur peut affaiblir une postcondition *ABox* d'un triplet et exprimer son résultat souhaité en termes de propriétés ensemblistes *TBox* pour le vérifier. Dans ce cas, un échec de la vérification au niveau terminologique peut avoir lieu avec un triplet de Hoare cependant valide.

Pour illustrer ce cas, considérons par exemple qu'on souhaite inverser les arcs r sortants des sommets de concept A vers les sommets de concept B. Cette transformation consiste en deux règles : une première règle qui renomme ces arcs en s pour avoir $a\ s\ b$ au lieu de $a\ r\ b$ pour chaque

```
rule rename {
  pre: a : A and a : (≥ 1 r B);
  inv : a : A;
  while (a : (≥ 1 r B)) do {
   select b with b : B and a r b;
   delete(a r b);
   add(a s b);
  }
  post: a : A and a : (≥ 1 s B);
}

rule reverse {
  pre: a : A and a : (≥ 1 s B);
  inv : a : A;
  while (a : (≥ 1 s B)) do {
   select b with b : B and a s b;
   delete(a s b);
   add(b r a);
  }
  post: a : A and a : (= 0 s B);
}

main {
  assert: (some r- A) ⊆ B and (some r A) = empty;
  rename! ;
  reverse! ;
  assert: (some r- A) = empty and (some r A) ⊆ B;
}
```

FIGURE 3.18 – Affaiblissement des spécifications *ABox*

instance a de A et b de B, et une deuxième qui supprime ces arcs $a\ s\ b$ et ajoute $b\ r\ a$. Ces deux règles, respectivement *rename* et $reverse$, sont décrites à la figure 3.18. Elles agissent sur un sommet a sélectionné de A. Pour appliquer ces règles sur tous les sommets de A, un appel itératif de chacun est nécessaire. Dès lors, la séquence ($rename!$; $reverse!$) met en œuvre la transformation souhaitée, tout en ayant une postcondition du premier appel conforme à la précondition du deuxième.

Supposons maintenant l'action *add*($a\ r\ b$) dans la boucle *while* de la règle $reverse$ à la place de *add*($b\ r\ a$). Le résultat de la règle $reverse$ sera donc un renommage des arcs s en r. Dans ce cas, en appliquant la séquence ($rename!$, $reverse!$) sur un graphe, on obtient un graphe cible identique au graphe source, ce qui rend la transformation sans effet. Cette erreur n'est pas identifiable au niveau de la règle $reverse$ du fait que le développeur a affaibli sa postcondition $ABox$ tout en conservant la validité de son triplet. En fait, il n'a pas vérifié qu'il n'y a pas d'arcs r sortants de l'instance a et cela en ignorant le fait $a : (= 0\ r\ B)$ de la postcondition. La vérification $TBox$, comme le montre la figure 3.18, exploite les *TFormules* exprimant les propriétés ensemblistes avant et après l'appel des règles pour vérifier le graphe abstrait. Le graphe initial est ainsi décrit par la *TFormule* (*some* r- $A) \subseteq B$ and (*some* $r\ A$) = *empty* signifiant qu'il existe des arcs r sortants des sommets de concept A vers des sommets de concept B, mais qu'il n'existe pas d'arcs r se dirigeant vers A. D'autre part, le résultat attendu est exprimé par la *TFormule* (*some* r- A) = *empty and* (*some* $r\ A) \subseteq B$ pour vérifier qu'il n'y a pas d'arcs r issus de A et que les arcs r de cible A sont tels que leurs sommets appartiennent à B. Le *TFait* (*some* r- A) = *empty* sera évidement insatisfiable vue l'erreur commise dans le code qui a rendu la transformation exprimée par la séquence sans effet.

Malgré la vérification *ABox* des règles par la logique de Hoare, l'incohérence entre le résultat obtenu et le résultat attendu se traduit soit par une écriture incorrecte des formules $TBox$, soit par un affaiblissement des spécifications *ABox* des triplets de Hoare. Dès lors que le développeur renforce la postcondition de la règle $reverse$ de l'exemple donné par le fait $a : (= 0\ r\ B)$ pour vérifier qu'il n'y a pas d'arcs r sortants de l'instance a, la preuve de cette règle échoue et le développeur se rend compte par conséquent qu'il s'est trompé dans l'instruction *add*($a\ r\ b$). D'où l'importance de la vérification *TBox* permettant de compléter la vérification *ABox* pour assurer la transformation correcte du graphe.

3.5 Propriétés du second ordre

La logique monadique du second ordre (MSO) [36] est une extension de la logique du premier ordre avec des variables dénotant des ensembles. Cette logique autorise ainsi les quantifications portant sur des relations unaires, c'est-à-dire sur des ensembles et pas seulement sur des objets individuels. C'est ainsi que les propriétés de cette logique ont la capacité d'exprimer des propriétés d'un graphe. Par exemple, savoir si un graphe est connexe, acyclique, biparti, etc.. [105].

De ce fait, les assertions *TBox* exprimant des propriétés sur un graphe peuvent être perçues comme des formules MSO. Considérons le problème de vérifier si un graphe est biparti : un graphe dont les sommets sont partitionnés en deux ensembles exprimant deux couleurs différentes par exemple A et B, et dans lequel chaque arc relie un sommet de A à un sommet de B. La figure 3.19 montre un exemple d'une règle Small-t$\mathcal{ALC}$ nommée *grow* qui permet de relier un sommet du graphe à un autre ayant un concept différent avec un arc étiqueté r. Les propriétés décrivant ce graphe peuvent être exprimées par un invariant *TBox* Small-t$\mathcal{ALC}$ composé de deux *TFaits* : $(some\ r\ A) \cap A = empty$ pour vérifier que l'ensemble A et l'ensemble des sommets sources des arcs r se dirigeant vers A sont disjoints, et $(some\ r\ B)\ \cap\ B = empty$ pour vérifier que l'ensemble B et l'ensemble des sommets sources des arcs r se dirigeant vers B sont disjoints. Afin d'exclure la possibilité d'ajouter un arc r entre des sommets ayant des concepts différents de A ou B, des axiomes de fermeture sont nécessaires : $A \cup B = all$ et $A \cap B = empty$ signifiant que tous les sommets du graphe sont soit de concept A, soit de concept B exclusivement. En traduisant cet invariant en tant que pré- et postcondition *TBox* identiques relatives à l'appel itératif de la règle *grow* et en assumant la validité de la *TFormule* associée avant l'appel, cette *TFormule*, d'après l'analyseur statique *TBox* de Small-t$\mathcal{ALC}$, sera toujours valide après l'appel. De ce fait, elle s'apparente à un invariant de construction du graphe.

```
rule grow {
 pre: x : (= 0 r A|B);
 if (x : A) then
  select y with y : B;
 else
  select y with y : A;
 add(x r y);
 post: x : (= 1 r A|B);
}

main {
 assert: (some r A) & A = empty and (some r B) & B = empty
        and A|B = all and A & B = empty;
 grow!;
 assert: (some r A) & A = empty and (some r B) & B = empty
        and A|B = all and A & B = empty;
}
```

FIGURE 3.19 – Programme Small-t$\mathcal{ALC}$ permettant de construire un graphe biparti

D'autres propriétés MSO peuvent se traduire par des *TFormules* Small-t$\mathcal{ALC}$. Par exemple, le *TFait* $C \subseteq (some\ succ\ C)$ signifiant que les sommets de concept C forment un sous-ensemble de

l'ensemble des sommets sources des arcs *succ* se dirigeant vers des sommets de C, contraint tous les sommets du concept C d'être connectés via des arcs *succ* à au moins un sommet de C. Par conséquent, ce *TFait* vérifie s'il existe au moins un chemin étiqueté *succ* entre tous les nœuds du concept C. Cette propriété peut être vérifiée après l'application itérative du code de la règle de la figure 3.20 qui ajoute un arc sortant *succ* à chaque nœud de concept C n'ayant pas préalablement d'arcs sortants.

```
pre: x : C and x : (= 0 succ C);
select y with y : C;
add(x succ y);
post: x : C and x : (= 1 succ C);
```

FIGURE 3.20 – Une règle assurant le *TFait* $C \subseteq$ (*some succ* C)

D'autre part, en considérant les restrictions de cardinalité, Small-t$\mathcal{ALC}$ peut restreindre le nombre de sommets manipulés. Par exemple, le *TFait* $all = (\leq 1\ r\ all)$, qui signifie que tous les sommets d'un graphe ont au plus un seul arc r sortant, impose pour chaque sommet du graphe d'avoir au plus un sommet cible unique d'un rôle r. Ce *TFait* traduit que r est fonctionnel, c'est-à-dire $\neg(\exists x\ \exists y\ \exists z \mid (x\ r\ y) \wedge (x\ r\ z) \wedge (y \neq z))$. Autrement dit, l'occurrence illustrée à la figure 3.21 est interdite dans un graphe vérifiant ce *TFait*.

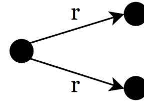


FIGURE 3.21 – Une occurrence interdite dans un graphe vérifiant le *TFait* $all = (\leq 1\ r\ all)$

Les assertions *TBox* Small-t$\mathcal{ALC}$ expriment des propriétés sur des ensembles de sommets dans le graphe ne pouvant pas être vérifiées par les assertions *ABox* portant sur des instances sélectionnées au moment de la transformation. Ces assertions *TBox* sont exploitées notamment après les appels itératifs des règles qui permettent de transformer des ensembles de sommets vérifiant une propriété spécifique.

D'autres propriétés MSO ne sont pas actuellement exprimables par nos assertions *TBox* qui sont limitées à décrire des propriétés sur des partitions de sommets et non pas sur des partitions d'arcs du graphe. Tel est le cas de la vérification qu'un graphe est acyclique, c'est-à-dire ne contenant aucun cycle. Cependant, la vérification de cette propriété peut se faire en appliquant, itérativement sur le graphe, la règle *ABox delete* qui supprime un arc r du graphe entre x et y s'il n'existe pas un arc entrant vers x ou un arc sortant de y [105], comme le montre la figure 3.22. Après la suppression

```
rule delete {
 pre: x r y and (x : (= 0 r- all) or y : (= 0 r all));
 delete(x r y);
 post: x !r y and (x : (= 0 r- all) or y : (= 0 r all));
}

main {
 assert: !((some r all) = empty);
 delete!;
 assert: (some r all) = empty;
}
```

FIGURE 3.22 – Programme Small-t$\mathcal{ALC}$ vérifiant qu'un graphe est acyclique

de tous les arcs satisfaisant cette condition, le graphe source est démontré acyclique si le graphe résultant ne contient aucun arc, c'est-à-dire s'il satisfait l'assertion *TBox* $(some\ r\ all) = empty$. Par conséquent, la vérification de cette propriété est établie par un algorithme traduit par une règle *ABox* et non directement par une formule *TBox*. Un des axes de travaux futurs pourrait concerner l'expressivité des formules *TBox* par rapport à la réécriture de graphes dans le cadre des logiques de description.

A noter de plus que nos propriétés *TBox* ne servent pas à filtrer un graphe, contrairement au langage de transformation de graphe GP qui exploite des conditions imbriquées (*Nested graph conditions*) de second ordre en membre gauche d'une règle [105].

3.6 Travaux similaires

Dans le contexte des approches algébriques des transformations de graphe, Poskitt et al. [105] étendent les conditions imbriquées des transformations écrites en GP [100] à des conditions en logique monadique du second ordre [38]. Leur proposition ajoute des quantifications sur les ensembles de nœuds et d'arcs et des assertions d'appartenance à ces ensembles. Ces propriétés sont prouvées avec le calcul de Hoare. Cela permet de vérifier des propriétés non-locales sur le graphe en plus des propriétés locales exprimées en logique du premier ordre par les contraintes et les conditions d'application [58]. Leurs conditions imbriquées sont exploitées également pour filtrer des parties d'un graphe en dotant les morphismes de contraintes sur les éléments appartenant à des ensembles. C'est ainsi qu'avec ces conditions, le filtrage d'un chemin entre deux nœuds, par exemple, est réalisable. En Small-t$\mathcal{ALC}$, le filtrage d'un graphe ayant de telles propriétés globales n'est pas assuré par les assertions *TBox*.

Plusieurs travaux de recherche ont mit l'accent sur la vérification des programmes en exploitant les deux composants *ABox* et *TBox* des logiques de description. Les travaux les plus proches sont

ceux de Ahmetaj et al. [9] qui exploitent les logiques de description pour décrire des propriétés d'évolution de bases de données exprimées sous la forme de graphes [28]. Plus précisément, la logique utilisée dans ces travaux est la logique $\mathcal{ALCHOIQ}$, une extension de la logique $\mathcal{ALC}$ enrichie par la hiérarchie des rôles, les concepts nominaux, les inverses et les restrictions des cardinalités. L'objectif est d'une part, de décider si une base de connaissances $\mathcal{K}$ traduisant des contraintes préserve sa satisfiabilité après l'application d'une suite d'actions, et d'autre part, de décider s'il existe ou non une suite d'actions qui conduit à un état indésirable à partir d'une instance spécifique ou d'une description incomplète. Leur langage consiste à ajouter et supprimer des instances et paires d'instances aux concepts et rôles respectivement via les actions $A \oplus C$, $A \ominus C$, $p \oplus r$ et $p \ominus r$. Les actions $A \oplus C$ et $A \ominus C$ ajoutent et suppriment respectivement le contenu d'un concept arbitraire C au (du) concept A. Les actions $p \oplus r$ et $p \ominus r$ ajoutent et suppriment également le contenu d'un rôle arbitraire r au (du) rôle p. Une instruction de sélection ($\mathcal{K}$? α_1 $[\![\alpha_2]\!]$) est également définie, traduisant que l'action α_1 est exécutée si le graphe est un modèle de la formule logique $\mathcal{K}$; dans le cas contraire, α_2 sera exécutée. Par contre, l'instruction *while* est remplacée par une formule en exploitant dans les actions basiques des concepts complexes comprenant des quantificateurs. Par exemple, l'action ($\mathit{Empl} \ominus \forall worksFor.\{p\}$) exprime la suppression du concept *Empl* de tous les individus travaillant ($worksFor$) sur le projet p.

Par rapport à une base de connaissances $\mathcal{K}$, les auteurs déterminent $TR_\alpha(\mathcal{K})$ qui constitue la transformation de $\mathcal{K}$ suite à une action α. Contrairement à notre analyseur statique *TBox*, le calcul se fait en mode régressif pour incorporer les effets d'une suite d'actions afin de produire la plus faible précondition $TR_\alpha(\mathcal{K})$ et prouver la satisfiabilité de $\mathcal{K}$, c'est-à-dire montrer qu'il existe un modèle $\mathcal{I}$ de $\mathcal{K}$ où $\Delta^\mathcal{I}$ est un ensemble fini. Toutefois, notre approche ne se limite pas uniquement à décider si la formule initiale reste satisfiable après les mises à jour, mais à inférer une formule valide en supposant une formule initiale valide.

Concernant la vérification des langages basés sur le calcul des situations tels que Golog [83] et Flux [119] conçus pour représenter des mondes changeant dynamiquement, des théories d'action basées sur la logique de description sont proposées dans la littérature [16, 84, 15, 13]. Étant données une séquence finie d'actions atomiques et une description éventuellement incomplète du monde initial, ces travaux permettent de vérifier si une propriété reste valide après l'exécution de cette séquence. Certains se focalisent sur des propriétés exprimant des *TBox* acycliques [15], alors que d'autres [16, 84] s'intéressent aux *TBox* générales. Baader et al. [13] utilisent les deux composants *ABox* et *TBox* pour vérifier des programmes Golog, un langage d'actions fondé sur la logique du premier ordre et dont l'interprète maintient une représentation explicite du monde dynamique conçu. Ce langage permet d'énoncer des axiomes exprimant des préconditions données par un utilisateur, des effets des actions et l'état initial du monde.

La logique de l'action consiste en des contraintes *TBox* $\mathcal{T}$ du domaine à ne pas violer, une description *ABox* $\mathcal{A}$ incomplète du monde initial, un ensemble fini d'actions Σ et un ensemble d'assertions pertinentes *ABox* $\mathcal{D}$ exprimant toutes les assertions de $\mathcal{A}$ et des pré- et postconditions

des actions. Afin de traduire l'effet d'une action noté ε sur une interprétation $\mathcal{I}$ exprimant une instance du domaine, des conditions sont nécessaires pour décider d'abord si cette action α est applicable sur $\mathcal{I}$. Il s'agit de vérifier que $\alpha \in \Sigma$, $\mathcal{I}$ est un modèle de $\mathcal{T}$, $\mathcal{I}$ satisfait la précondition, α est consistante avec $\mathcal{I}$ et que $\mathcal{I}'$ l'interprétation résultante de l'application de α est un modèle de $\mathcal{T}$. Si toutes ces conditions sont satisfaites, alors l'effet ε est exprimé par un ensemble d'axiomes appelé projection.

Contrairement à ce qui est présenté dans ce chapitre, ces travaux interdisent l'application d'une action si l'interprétation résultante n'est pas consistante avec les assertions définies. Cependant, l'analyseur statique *TBox* étudie l'effet des instructions Small-t$\mathcal{ALC}$ sur les assertions *TBox* données et transforme dans tous les cas une formule *TBox* initiale en une autre d'une manière que cette dernière soit consistante avec les instructions et assertions *ABox* des règles.

D'autres travaux mettent l'accent sur la dualité des composants *ABox* et *TBox* d'une base de connaissances, notamment sur le problème de répondre à des requêtes d'une base de données tolérant l'incohérence [22]. Lembo et al. [81, 82] traitent ce problème en essayant d'obtenir des connaissances consistantes pour répondre aux requêtes en se basant sur des méthodes de révision de la base de données. En considérant qu'en pratique dans la plupart des cas la *TBox* est supposée consistante et que les assertions de l'*ABox* peuvent contredire les axiomes de la *TBox*, leurs travaux consistent à construire une *ABox* $\mathcal{A}'$ à partir de l'*ABox* $\mathcal{A}$ inconsistante avec la *TBox* $\mathcal{T}$ pour réparer la base de connaissances $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$ qui résulte en $\langle \mathcal{T}, \mathcal{A}' \rangle$ consistante. D'autres révisent la base de connaissances $\mathcal{K}$ en considérant les connaissances terminologiques et le minimum d'axiomes possible de $\mathcal{K}$ pour assurer sa consistance [51, 97].

3.7 Conclusion

La spécification *ABox* écrite par le développeur concerne les propriétés locales des instances manipulées par la règle. Le fait d'y ignorer des propriétés peut avoir un impact important sur la vérification du résultat souhaité. Cependant, la spécification *TBox* permet de vérifier l'effet des règles sur le graphe global en réduisant les sommets à des ensembles. Les propriétés *TBox* expriment ainsi des relations de subsomption entre les concepts englobant les sommets

Le processus de vérification de ces propriétés est basé sur une analyse statique *TBox* qui capture l'effet des instructions *ABox* sur les propriétés *TBox* en inférant des formules *TBox* assurant la cohérence des deux niveaux. Cette analyse statique *TBox* est fondée principalement sur une analyse *ABox*. Dès lors, notre proposition exige de la part du développeur de renforcer sa spécification *ABox* pour ensuite vérifier l'effet global de la transformation par les formules *TBox* après l'écriture et la vérification séparées des règles. La vérification *ABox* ou *TBox* ne dispense pas de la vérification à l'autre niveau. Les deux sont ainsi complémentaires et permettent conjointement de vérifier localement et globalement la transformation.

Chapitre 4

Environnement de développement Small-t$\mathcal{ALC}$

Sommaire

4.1 Assistance à l'identification d'erreurs 122
4.1.1 Tests *ABox* 122
4.1.1.1 Assertions de test et formules Small-t$\mathcal{ALC}$ 122
4.1.1.2 Données de test et instructions Small-t$\mathcal{ALC}$ 123
4.1.1.3 Génération de tests par analyse dynamique 126
4.1.1.4 Génération de tests par analyse statique 127
4.1.2 Diagnostics *TBox* 129
4.2 Synoptique de l'environnement Small-t$\mathcal{ALC}$ 131
4.3 Étude de cas : gestion des services d'un hôpital 132
4.3.1 Écriture et correction de la règle *receivePatient* 133
4.3.2 Nouvelle écriture de la règle *assignDoctor* 135
4.3.3 Inférence d'une formule *TBox* après la règle *assignDoctor* 136
4.3.4 Localisation des erreurs dans la règle *allocateRoom* 137
4.3.5 Vérification dynamique des formules *TBox* 139
4.4 Autres environnements de transformation de graphes 143
4.4.1 Outils existants 143
4.4.2 Comparatif avec Small-t$\mathcal{ALC}$ 147
4.4.3 Synthèse 152
4.5 Conclusion 153

Les travaux dans le domaine de vérification ont connu récemment une convergence de diverses techniques, notamment une synergie entre la preuve et les tests. Comme souligné par Dijkstra [45], les tests servent à démontrer la présence d'erreurs mais jamais leur absence. En pratique, ils jouent un rôle primordial pour la mise au point des programmes ; ils sont de plus rentables et simples à utiliser. A cet égard, à coté des outils formels décrits aux deux chapitres précédents, l'étude s'intéresse à ces approches plus conventionnelles. Dans ce cadre, plusieurs outils ont été développés afin de tendre vers un environnement de développement dédié à l'écriture de programmes de transformation de graphes corrects. Ils font l'objet de la section 4.1 et sont synthetisés en 4.2. Une étude de cas est présentée à la section 4.3 concernant la gestion des services d'un hôpital. Le chapitre se conclut par la section 4.4 qui compare notre environnement Small-t$\mathcal{ALC}$ à d'autres outils de transformation de graphes.

4.1 Assistance à l'identification d'erreurs

L'arsenal formel est complété par des outils visant à fournir des diagnostics qui aident le développeur à identifier des erreurs dans son code et sa spécification *ABox* et dans ses formules *TBox*. La génération automatique de tests *ABox* relatifs à la spécification d'une règle permet de localiser les faits *ABox* violés de la pré- ou postcondition comme présenté à la section 4.1.1. A la section 4.1.2, nous montrons comment un raisonnement *TBox* sur les *TFormules* du point d'entrée du programme identifie les propriétés qui ne sont pas respectées par le graphe en cours de transformation.

4.1.1 Tests *ABox*

Les tests, contrairement à la preuve formelle, s'avèrent un moyen a priori simple pour détecter et localiser les erreurs dans un programme. Dans cette optique, étant donné le code d'une règle avec ses pré- et postconditions Small-t$\mathcal{ALC}$, des tests sont générés automatiquement en exploitant le formalisme du langage qui permet de traduire directement un fait de la spécification en une assertion de test [85, 18] Ces tests s'appliquent à des graphes également générés à partir des formules Small-t$\mathcal{ALC}$.

4.1.1.1 Assertions de test et formules Small-t$\mathcal{ALC}$

Nous avons défini une bibliothèque d'assertions de test Small-t$\mathcal{ALC}$ qui offre au développeur la possibilité de tester l'effet de ses règles et d'y identifier les erreurs : toute assertion non vérifiée est signalée défaillante. Pour faciliter leur interprétation, les assertions de test sont définies avec une syntaxe propice aux faits des formules comme le montre le tableau 4.1 où le paramètre *graph* d'une assertion correspond au graphe en cours d'examen.

TABLE 4.1 – Assertions associées aux faits Small-t$\mathcal{ALC}$

`i : C`	`assertExistNode(graph, i, C)`
`i :! C`	`assertNotExistNode(graph, i, C)`
`i r j`	`assertExistEdge(graph, i, r, j)`
`i !r j`	`assertNotExistEdge(graph, i, r, j)`
`i = j`	`assertNodesAreEqual(graph, i, j)`
`i != j`	`assertNodesAreNotEqual(graph, i, j)`
`i : (<= n r C)`	`assertAtMostNumberOutgoingEdges(graph, i, C, r, n)`
`i : (>= n r C)`	`assertAtLeastNumberOutgoingEdges(graph, i, C, r, n)`
`i : (= n r C)`	`assertEqualNumberOutgoingEdges(graph, i, C, r, n)`

De par la proximité sémantique des faits Small-t$\mathcal{ALC}$ et des assertions de test, toute précondition ou postcondition *ABox* peut se dériver automatiquement en un test. Par exemple, la formule $a : A$ *and* $a : (\leq 2\ r\ A)$ *and* $a\ !r\ a$ donne lieu au test de la figure 4.1. L'échec d'au moins une assertion indique l'insatisfiabilité de la formule. Ainsi, l'échec du premier test indique ainsi que le nœud désigné par a n'est pas de concept A.

```
assertExistNode(graph, a, A);
assertAtLeastNumberOutgoingEdges(graph, a, A, r, 2);
assertNotExistEdge(graph, a, r, a);
```

FIGURE 4.1 – Test généré à partir de la formule $a : A$ *and* $a : (\leq 2\ r\ A)$ *and* $a\ !r\ a$

Les assertions de test Small-t$\mathcal{ALC}$ ont été conçues et implémentées en tant que surcouche des assertions JUnit [6]. Ce choix de réalisation a été motivé par le souhait de prototyper des outils de test dérivés d'analyses statique et dynamique des triplets Small-t$\mathcal{ALC}$, et non d'offrir un environnement de type xUnit [61] dédié au test unitaire des règles.

Afin d'exécuter les tests, des données d'entrée doivent évidemment être fournies. Dans notre contexte, ces données sont des graphes.

4.1.1.2 Données de test et instructions Small-t$\mathcal{ALC}$

Pour assurer plus de crédibilité à la formule testée, il est important de générer un maximum de données de test. Dans cet objectif, nous avons adapté l'algorithme de Molloy et Reed [92] afin de produire un ensemble de graphes homomorphes à un graphe dit typique [88].

Un graphe typique correspondant à une formule $ABox$ peut être généré grâce au formalisme du langage Small-t$\mathcal{ALC}$: chaque fait $ABox$ exprime l'existence d'un nœud ou d'un arc du graphe. Dès lors, une formule $ABox$ est transformée en une séquence d'instructions add définissant un graphe représentant le modèle le plus basique associé à cette formule. Par exemple, la formule $x : C$ and x r y se traduit par la séquence $add(x : C); add(x\ r\ y)$. Les faits invoquant des restrictions considèrent le nombre de restrictions. Par exemple, les deux faits $x : (\leq 2\ r\ C)$ et $x : (\geq 2\ r\ C)$ se traduisent par la même séquence d'instructions $add(x\ r\ y_1); add(y_1 : C); add(x\ r\ y_2); add(y_2 : C)$. A noter que seuls les faits positifs de la formule sont considérés lors de la génération de la séquence d'instructions définissant le graphe source. Les faits exprimant des négations, par exemple $x : !C$, $x\ !r\ y$ et $x \,!= y$ sont négligés lors de cette étape ; ils seront pris en compte à la génération des graphes homomorphes à ce graphe comme présenté par la suite. Un fait exprimant une égalité entre les nœuds, par exemple $x1 = x2$ se traduit en remplaçant toutes les occurrences de $x2$ dans la formule en question par $x1$ avant le processus de génération des instructions. Le graphe de la figure 4.2 correspond au graphe typique de la formule $a : A$ and $a : (<= 3\ r\ A)$ and $b : A$.

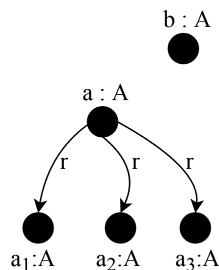


FIGURE 4.2 – Graphe typique de la formule $a : A$ and $a : (<= 3\ r\ A)$ and $b : A$

L'algorithme de Molloy et Reed est basé sur un appariement aléatoire des arcs du graphe [92]. Son objectif est de construire un graphe à partir de demi-arêtes à connecter entre deux nœuds, selon une politique d'appariement aléatoire respectant la distribution des degrés des nœuds. Il s'agit de : (1) donner à chaque nœud du graphe le bon nombre de demi-arêtes connectées à ce nœud, (2) numéroter toutes les demi-arêtes, (3) relier chacune des demi-arêtes à une autre choisie aléatoirement.

L'algorithme de Molloy et Reed a été adapté à notre contexte en ignorant le caractère aléatoire des connexions entre les nœuds du graphe et en considérant les concepts d'appartenance des nœuds.

L'algorithme de génération pour les restrictions se décline comme suit :

- sélectionner un arc r ayant pour cible un nœud anonyme, c'est-à-dire un nœud associé implicitement aux restrictions,
- sélectionner un nœud n autre que le nœud anonyme cible de l'arc r, mais appartenant au même concept,

— créer un arc temporaire étiqueté r, reliant le nœud source de l'arc sélectionné au nouveau nœud n sélectionné,

— rajouter l'arc créé s'il n'appartient pas déjà au graphe, et supprimer l'arc sélectionné, à condition qu'aucun fait de la formule initiale ne spécifie l'absence de cet arc, soit x $!r$ n où x désigne la source de cet arc.

Considérons le graphe typique de la figure 4.2 généré à partir de la formule $a : A$ *and* $a : (<= 3\ r\ A)$ *and* $b : A$. La figure 4.3 présente trois graphes homomorphes à ce graphe parmi plusieurs générés par cet algorithme.

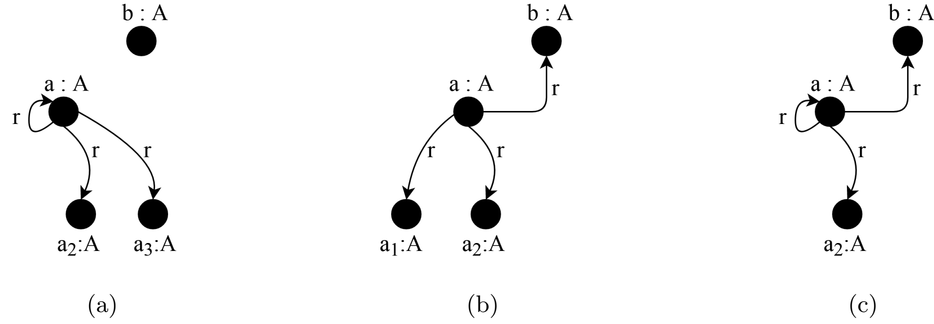


FIGURE 4.3 – Trois graphes associés à la formule $a : A$ *and* $a : (<= 3\ r\ A)$ *and* $b : A$ et homomorphes au graphe de la figure 4.2

Le calcul de la borne supérieure du nombre de graphes générés à partir d'un graphe typique considère le nombre $n1$ de nœuds nommés de ce graphe, le nombre $n2$ de nœuds anonymes, et $n = inf(n1, n2)$ le nombre total de nœuds considérés lors de la construction de l'ensemble des ensembles de nœuds conduisant à une mise à jour du graphe typique. Le choix de k nœuds dans le graphe parmi ces n discernables, où l'ordre n'intervient pas, conduit à $\begin{pmatrix} n \\ k \end{pmatrix} = \frac{k!}{n!(n-k)!}$ combinaisons possibles. En itérant sur les valeurs de k, $\sum_{k=1}^{n} \begin{pmatrix} n \\ k \end{pmatrix} = 2^n - 1$ sera une borne supérieure du nombre de graphes homomorphes pouvant être générés à partir du graphe typique. Un théorème bien connu en analyse combinatoire est ainsi identifié : un ensemble de n éléments possède 2^n parties.

L'algorithme de génération de l'ensemble des ensembles de nœuds, et donc de la génération des graphes homomorphes, est exponentiel. Pour un ensemble de n éléments donné, la valeur maximale du nombre de combinaisons pouvant être formées est atteinte pour $\frac{n}{2}$. Dès lors, $\begin{pmatrix} n \\ \frac{n}{2} \end{pmatrix}$ est la plus

grande des valeurs parmi toutes les valeurs $\begin{pmatrix} n \\ k \end{pmatrix}$ calculées. Par conséquent, la complexité de la génération de toutes les combinaisons pour un ensemble de n éléments est $O(n * \begin{pmatrix} n \\ \frac{n}{2} \end{pmatrix})$. Grâce à la formule de Stirling $n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$, la complexité est exponentielle. Bien que la complexité soit exponentielle, le nombre de nœuds d'une formule Small-t$\mathcal{ALC}$ est suffisamment réduit pour exécuter en temps raisonnable cet algorithme [88].

La réussite d'un ensemble de tests appliqués à un graphe conforme à une formule *ABox* ne prouve pas que ces tests réussissent sur les autres graphes satisfaisant cette formule. La génération d'un grand nombre de graphes permet simplement une meilleure couverture des tests, notamment dans le cas où les tests sont appliqués à un graphe généré identique au graphe contre-exemple du prouveur que notre algorithme est en mesure d'identifier comme illustré ultérieurement.

4.1.1.3 Génération de tests par analyse dynamique

Les tests faisant suite à une analyse dynamique permettent de vérifier la satisfaction de la postcondition par rapport à la précondition et aux instructions du triplet. Comme illustré à la figure 4.4, le principe est comme suit :

1. un graphe source est fourni par le développeur, sinon, plusieurs graphes sont générés automatiquement à partir de la précondition de la règle,
2. le code de transformation est appliquée au(x) graphe(s) source(s),
3. un ensemble de tests est généré à partir de la postcondition et exécuté sur le(s) graphe(s) cible(s).

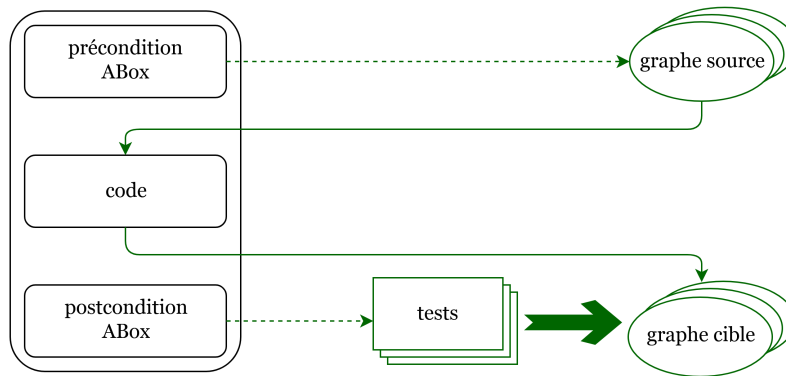


FIGURE 4.4 – Processus de génération des tests par analyse dynamique *ABox*

Vu que chaque assertion de test correspond à un fait, les assertions qui échouent permettent d'identifier les faits inconsistants de la postcondition. Considérons l'exemple incorrect de la règle *update* du premier chapitre repris à la figure 4.5. Son graphe typique est celui de la figure 4.2 et le test généré à partir de la postcondition est indiqué à la figure 4.6. La première assertion correspond au fait $a : !A$, la deuxième à $a : (\geq 2\ r\ A)$ et la troisième à $b : A$.

```
pre: a : A and a : (≥ 3 r A) and b : A;
select n with a r n and n : A;
delete(a r n);
delete(a : A);
post: a :!A and a : (≥ 2 r A) and b : A;
```

FIGURE 4.5 – Triplet de Hoare incorrect

```
assertNotExistNode(graph, a, A);
assertAtLeastNumberOutgoingEdges(graph, a, A, r, 2);
assertExistNode(graph, b, A);
```

FIGURE 4.6 – Test associé à la postcondition *ABox* de la règle *update*

L'application de la règle au graphe typique et l'exécution du test n'entraîne aucun échec. Par contre, la réalisation de ce même processus avec le graphe de la figure 4.3a, qui est similaire au graphe contre-exemple du prouveur, montre que le fait $a : (\geq 2\ r\ A)$ est inconsistant. Ceci conduit le développeur à revoir son code ou à corriger ce fait en $a : (\geq 1\ r\ A)$.

4.1.1.4 Génération de tests par analyse statique

Étant donnés un code et ses pré- et postconditions, l'analyseur statique suppose que le code de transformation est correct et vérifie les pré- et postconditions vis-à-vis des instructions. Il peut procéder en mode progressif afin de produire une postcondition à partir de la précondition et du code du triplet, ou en mode régressif pour la production d'une précondition.

En considérant le mode progressif, la comparaison entre la postcondition de l'analyseur et la postcondition du triplet se fait par le biais de tests. Initialement, deux ensembles de tests et deux ensembles de graphes sont générés à partir des deux postconditions à comparer. La figure 4.7 montre le processus de comparaison ; il s'agit d'appliquer les tests qui correspondent à une des postconditions aux graphes associés à l'autre postcondition :

— l'application ❶ des tests de la postcondition du triplet aux graphes associés à la postcondition de l'analyseur permet d'identifier des erreurs dans la postcondition du triplet,

— l'application ❷ des tests de la postcondition de l'analyseur aux graphes associés à la postcondition du triplet permet au développeur, en plus, de renforcer éventuellement sa postcondition.

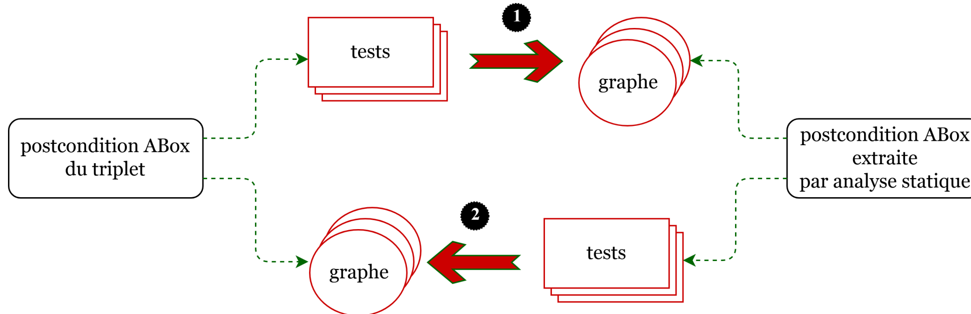


FIGURE 4.7 – Processus de génération des tests par analyse statique *ABox*

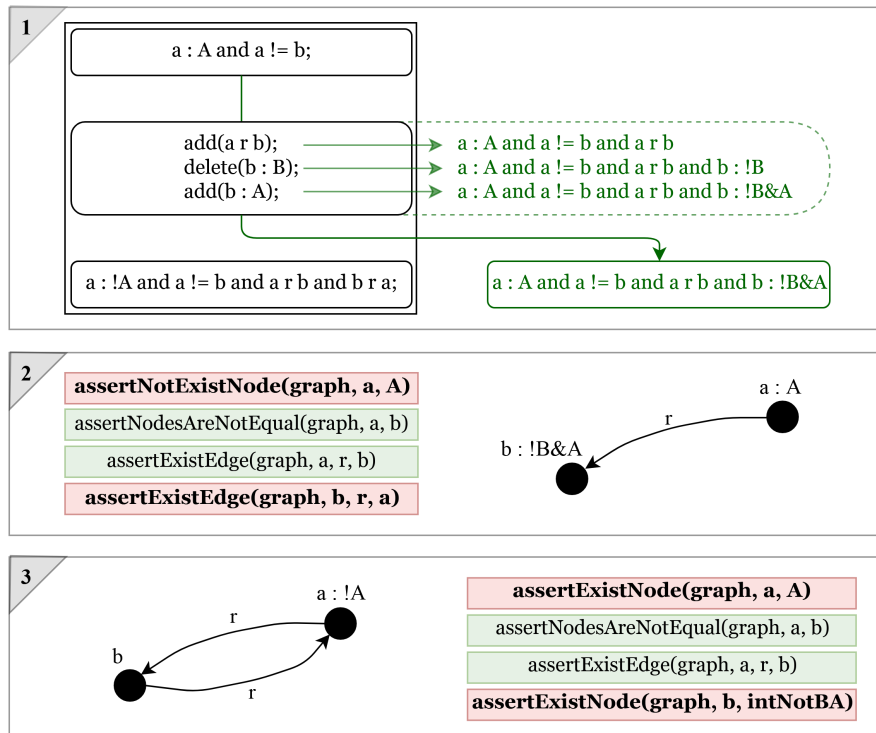


FIGURE 4.8 – Exemple d'identification d'erreurs par analyse statique [1]

1. Le concept *intNotBA* mentionné à la dernière assertion de la figure 4.8 est défini comme concept composé traduisant l'intersection !B&A entre le complément de concept B et le concept atomique A.

Afin d'illustrer ce processus, considérons la règle de la figure 4.8 manipulant un nœud a de concept A et un nœud b différent de a. La transformation consiste à ajouter un arc r entre a et b, à supprimer le nœud b de concept B et à l'ajouter au concept A. La postcondition proposée par le développeur pour son triplet est telle que a est de concept $!A$, que a est différent de b et qu'il existe deux arcs r connectant a à b et b à a. Vis-à-vis de cette postcondition incorrecte, le générateur de tests par analyse statique est capable de mettre en évidence des tests qui échouent selon le processus matérialisé par les trois cadres de la figure 4.8 :

(1) extraction d'une postcondition à partir de la précondition et du code du triplet, soit $a : A$ *and* $a != b$ *and* $a\ r\ b$ *and* $b : !B\&A$;

(2) génération des tests associés à la postcondition du triplet incluant notamment les tests $a : !A$ et $b\ r\ a$, et application de ces tests aux graphes de la postcondition de l'analyseur, par exemple, au fragment de graphe typique de la figure 4.8 qui stipule que a est de type A et b de type $!B\&A$;

(3) génération des tests associés à la postcondition de l'analyseur, dont $a : A$ et $b : !B\&A$, et application de ces tests aux graphes de la postcondition du triplet.

La figure 4.8 synthétise cette boucle de tests et indique en gras les tests qui échouent. L'étape (2) signale que les tests $a : !A$ et $b\ r\ a$ invalident la postcondition attendue ; l'étape (3) alerte le développeur sur sa postcondition au regard des faits $a : A$ et $b : !B\&A$.

Contrairement aux preuves formelles, les tests sont plus simples à mettre en œuvre et permettent, en plus, d'obtenir des informations de localisation des erreurs dans une règle. D'où leur efficacité à assister des développeurs novices à améliorer l'écriture de leurs règles de transformation.

4.1.2 Diagnostics *TBox*

A partir d'une formule *TBox* supposée valide, le moteur d'inférence vérifie statiquement une formule *TBox* après une séquence d'appels de règles Small-t$\mathcal{ALC}$. Cette assistance peut aussi s'envisager dynamiquement pour la formule *TBox* donnée en prémisse.

Étant donné que les ontologies peuvent naturellement être assimilées à des graphes orientés et étiquetés, un raisonneur offrant des services d'inférence sur les ontologies peut être exploité. Les faits *TBox* Small-t$\mathcal{ALC}$ exprimant des relations d'équivalence et d'inclusion de concepts peuvent ainsi être aisément vérifiés sur les graphes mis à jour par les règles appelées. Plus spécifiquement, l'environnement Small-t$\mathcal{ALC}$ exploite le raisonneur HermiT [56] afin d'identifier des relations de subsomption entre les classes d'une ontologie.

Contrairement au moteur d'inférence *TBox*, le raisonneur exige la définition d'un graphe source de la part du développeur. Les *TFormules* sont vérifiées vis-à-vis des graphes mis à jour au point où les *TFormules* sont mentionnées ; elles sont donc indépendantes. Comme le montre la figure 4.9, chaque fait d'une *TFormule* se traduit en des services de raisonnement qui s'appliquent à l'ontologie

FIGURE 4.9 – Processus de vérification d'une formule *TBox*

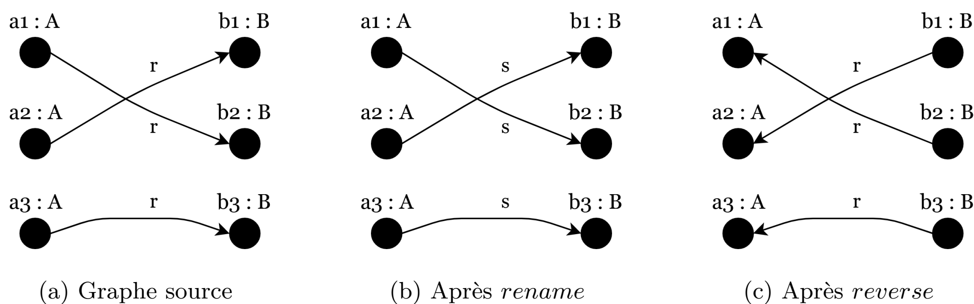
associée au graphe source. Le raisonneur indique finalement les faits *TBox* qui sont satisfiables vis-à-vis du graphe et ceux qui ne le sont pas.

Considérons une transformation d'un graphe qui consiste en deux ensembles de nœuds, le premier de concept A et le deuxième de concept B avec des arcs r sortants de tous les nœuds de A vers tous les nœuds de B. La transformation consiste à inverser les arcs r : dans le graphe cible, les arcs r seront des arcs sortants des nœuds de B vers les nœuds de A. En Small-t$\mathcal{ALC}$, cette transformation est décrite à la figure 3.18 du chapitre 3 et se compose de deux règles : la première *rename* qui renomme les arcs r en s afin de les marquer, et la deuxième *reverse* qui supprime les arcs des couples de nœuds (a, b) typés s et les remplace par des couples (b, a) typés r.

Les formules *TBox* sont specifiées par les clauses *assert* à la figure 4.10 et les graphes source et cible de chaque règle sont synthetisés à la figure 4.11.

```
main {
  assert: (some r- A) = B;
  rename! ;
  assert: (some r- A) = empty and (some s- A) = B;
  reverse! ;
  assert: (some r- A) = empty and (some r A) = B and (some s- A) = B;
}
```

FIGURE 4.10 – Point d'entrée annoté de formules *TBox*



(a) Graphe source (b) Après *rename* (c) Après *reverse*

FIGURE 4.11 – Évolution du graphe après chaque règle

Les deux premières *TFormules* au regard du graphe source 4.11a sont valides. Par contre, la dernière égalité d'ensembles $(some\ s\text{-}\ A) = B$ caractérisant le graphe cible est incorrecte. Dès lors, le développeur peut considérer conjointement cette égalité et le code de sa règle *reverse* et transformer l'égalité en $(some\ s\text{-}\ A) = empty$.

4.2 Synoptique de l'environnement Small-t$\mathcal{ALC}$

Visant à assister le développeur à différents niveaux pour atteindre des programmes de transformation de graphes corrects, l'environnement Small-t$\mathcal{ALC}$ propose plusieurs composants formels comme l'extracteur de préconditions *ABox* et le moteur d'inférence *TBox*, et moins formels comme le générateur de tests *ABox* et le raisonneur *TBox*. Il s'accompagne d'un compilateur qui permet l'exécution des transformations. La figure 4.12 illustre les différents composants de l'environnement expérimental Small-t$\mathcal{ALC}$. Plus spécifiquement, le tableau 4.2 présente les composants développés durant cette thèse ainsi que les techniques utilisées lors de leur implémentation. Ces composants s'appuient sur des API Java et des outils existants sous Eclipse dont il conviendrait de revoir les fonctionnalités, notamment le *framework* JUnit qui met en œuvre des tests unitaires Java et qui n'est pas ainsi adapté à la prise en compte de tests interdépendants comme peuvent l'être ceux associés à des pré- ou postconditions exprimées en forme normale disjonctive.

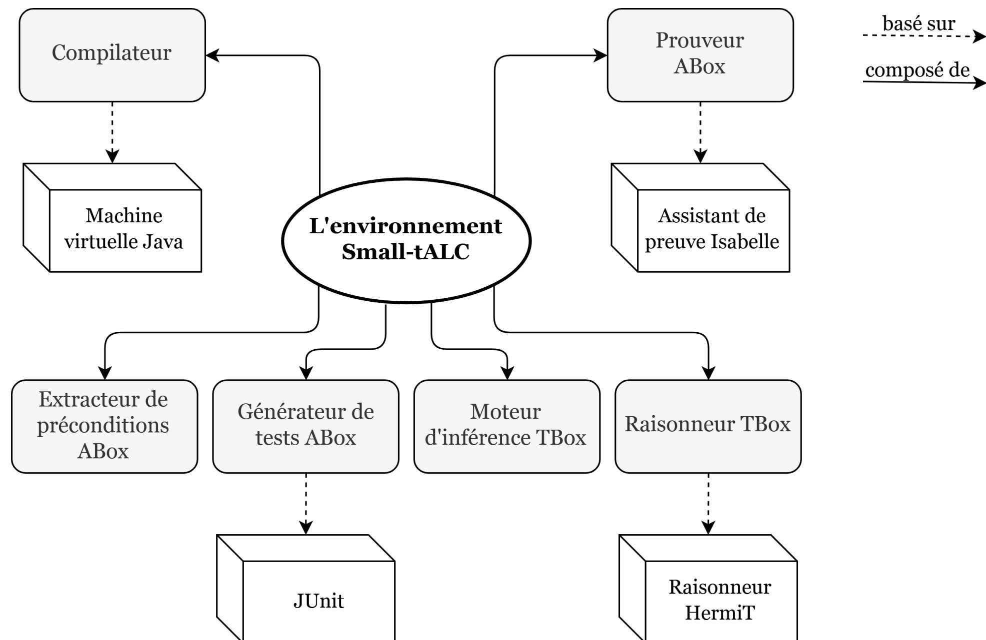


FIGURE 4.12 – L'environnement expérimental Small-t$\mathcal{ALC}$

Table 4.2 – Composants Small-t$\mathcal{ALC}$ développés durant cette thèse

Composant	Langage	Outil	Chapitre
Extracteur de préconditions *ABox*	Java		2
Moteur d'inférence *TBox*	Java		3
Générateur de tests *ABox*	Java	Framework JUnit	4
Raisonneur *TBox*	Java	API HermiT	4

La table 4.3 résume l'effort de développement des différents composants de l'environnement quantifié en nombre de lignes de code [2] dont 8000 ont été développés durant cette thèse. Aux 1800 lignes de code du générateur de tests *ABox* s'ajoutent celles du processus régressif d'extraction de préconditions *ABox*.

Table 4.3 – Nombre de lignes de code des composants Small-t$\mathcal{ALC}$

Composant	Nombre de lignes de code
Compilateur	4770
Extracteur de préconditions *ABox*	2180
Moteur d'inférence *TBox*	690
Générateur de tests *ABox*	1800
Raisonneur *TBox*	440
Prouveur *ABox*	7000
Total	16880

Le projet Eclipse de l'environnement Small-t$\mathcal{ALC}$ ainsi que les instructions d'installation sont distribués depuis le site du projet CLIMT [3] à la rubrique *Small-t$\mathcal{ALC}$ environment*.

4.3 Étude de cas : gestion des services d'un hôpital

Afin d'illustrer l'utilisation des différents composants de l'environnement Small-t$\mathcal{ALC}$, cette section développe un scénario possible de mise au point d'un programme de transformation. Étoffons les activités d'un hôpital abordées en partie au chapitre 2, en écrivant un programme composé de trois règles :

2. Christian Percebois a contribué pour la moitié du développement du compilateur et Pierre Mespoulet a développé le générateur des données de test. Le prouveur, issu du projet CLIMT et développé en Isabelle/HOL, a été écrit par Martin Strecker de l'équipe ACADIE. L'extraction Scala des fonctions Isabelle a nécessité un code glu proposé par Nadezhda Baklanova, aujourd'hui intégré au compilateur.

3. https ://www.irit.fr/Climt/Software/smalltalc.html

— *receivePatient* pour accueillir un patient et l'inscrire dans le département qui correspond à sa maladie,
— *assignDoctor* pour affecter à un patient un médecin traitant rattaché au même département d'inscription du patient,
— *allocateRoom* pour réserver une chambre du département au patient.

4.3.1 Écriture et correction de la règle *receivePatient*

Cette première règle *receivePatient* permet de sélectionner une personne *p* non inscrite à l'hôpital et de l'affecter au département *dep* qui correspond à sa maladie. Considérons une première version comme indiqué à la figure 4.13. Ne comportant pas de restrictions, la règle semble simple à écrire selon le développeur. Par contre, le composant de preuve *ABox* ne réussit pas à la vérifier et produit le contre-exemple de la figure 4.14.

```
rule receivePatient {
 pre: p : InQueue and p suffers disease
      and dep : Department and dep inChargeOf disease;
 delete(p : InQueue);
 add(p : Patient);
 add(dep registers p);
 post: p : Patient and p suffer disease
       and dep : Department and dep inChargeOf disease
       and dep registers p;
}
```

FIGURE 4.13 – Première version de la règle *receivePatient*

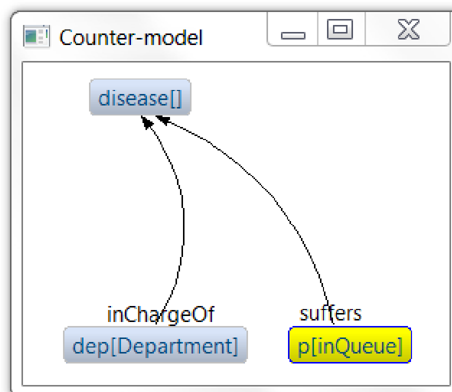
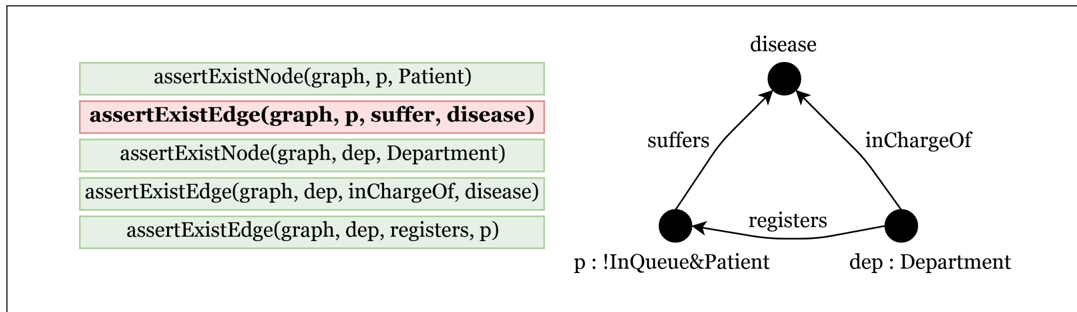


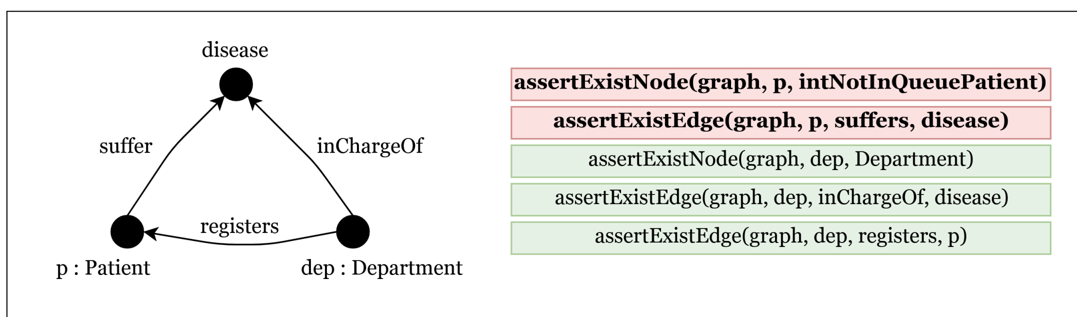
FIGURE 4.14 – Contre-exemple de la règle *receivePatient* de la figure 4.13

Du point de vue du développeur, ce graphe produit par le prouveur respectant la précondition doit en principe atteindre la postcondition. Après son échec à identifier la source du problème, il exploite le générateur de tests par analyse statique *ABox* pour localiser l'erreur.

En mode progressif, cet outil génère deux tests : un qui correspond à la postcondition du triplet et appliqué au graphe de la postcondition de l'analyseur, et l'autre qui correspond à la postcondition de l'analyseur et appliqué au graphe de la postcondition du développeur. Les figures 4.15a et 4.15b illustrent ces deux situations, en considérant les graphes typiques correspondants.



(a) Test associé à la postcondition du triplet et graphe typique de la postcondition de l'analyseur



(b) Test associé à la postcondition de l'analyseur et graphe typique de la postcondition du triplet [4]

FIGURE 4.15 – Résultats du générateur de tests par analyse statique ABox

D'après la figure 4.15a, le développeur constate que c'est le fait *p suffer disease* de la postcondition qui pose problème. En effet, ce n'est qu'une erreur syntaxique involontaire non évidente à identifier : *suffer* à la place *suffers*. De plus, à la figure 4.15b, l'échec du premier test montre que le développeur peut renforcer sa postcondition en indiquant que p appartient au concept $!Queue$. Le fait $p : Patient$ peut ainsi être transformé en $p : !Queue\&Patient$. Par conséquent, le développeur

4. Le concept *intNotInQueuePatient* mentionné à la première assertion de la figure 4.15b est défini comme concept composé traduisant l'intersection $!InQueue\&Patient$ entre le complément du concept $InQueue$ et le concept atomique $Patient$.

```
rule receivePatient {
 pre: p : InQueue and p suffers disease
      and dep : Department and dep inChargeOf disease;
 delete(p : InQueue);
 add(p : Patient);
 add(dep registers p);
 post: p : !Queue&Patient and p suffers disease
       and dep : Department and dep inChargeOf disease
       and dep registers p;
}
```

FIGURE 4.16 – Version finale et correcte de la règle *receivePatient*

modifie sa postcondition comme indiqué à la figure 4.16 et soumet son nouveau triplet au prouveur qui atteste que cette première règle du programme est formellement correcte.

4.3.2 Nouvelle écriture de la règle *assignDoctor*

La deuxième règle *assignDoctor* chargée d'affecter un médecin à un patient s'avère plus complexe : un chef de département ne peut traiter au plus que trois patients. Afin d'éviter la violation de cette contrainte, cette règle a été écrite au chapitre 2 en supposant que le développeur a simplement décidé d'affecter aux patients des médecins n'ayant pas le statut de chef de département. Rappelons à la figure 4.17 la première version du code et de la postcondition, et considérons désormais qu'un chef peut effectivement traiter des patients.

```
rule assignDoctor {
  add(p refDr dr);
  add(dr treats p);
  post: p : Patient and dr : Doctor
        and dep : Department and dr worksIn dep
        and dep registers p and head : Doctor
        and dep directedBy head
        and head : (≤ 3 treats Patient)
        and dr treats p and p : (≤ 1 refDr Doctor);
}
```

FIGURE 4.17 – Version initiale de la règle *assignDoctor*

Considérant l'extracteur de préconditions *ABox* visant à l'obtention d'un triplet correct par construction, douze propositions déjà détaillées à section 2.3 du chapitre 2, sont exposées au développeur. Parmi ces formules, il s'agit d'identifier celles qui n'imposent aucune contrainte d'*aliasing*

sur le médecin *dr* et qui indiquent que le patient *p* n'a pas encore de référent. Ceci se traduit par l'absence des deux faits *dr = head* et *dr != head* et la présence du fait *p !refDr dr* dans la précondition. Une seule des préconditions proposées par l'extracteur respecte ces exigences :

p : Patient and dr : Doctor and dep : Department and dr worksIn dep
and dep registers p and head : Doctor and dep directedBy head
and head : ($\leq$ 2 treats Patient) and ***p !refDr dr*** *and p : ($\leq$ 0 refDr Doctor)*

```
rule assignDoctor {
 pre: p : Patient and dr : Doctor
      and dep : Department and dr worksIn dep
      and dep registers p and head : Doctor
      and dep directedBy head
      and head : (≤ 2 treats Patient)
      and p !refDr dr and p : (= 0 refDr Doctor);
 add(dr treats p);
 add(p refDr dr);
 post: p : Patient and dr : Doctor
       and dep : Department and dr worksIn dep
       and dep registers p and head : Doctor
       and dep directedBy head
       and head : (≤ 3 treats Patient)
       and dr treats p and p : (≤ 1 refDr Doctor);
}
```

FIGURE 4.18 – Version finale de la règle *assignDoctor*

La figure 4.18 montre la nouvelle version de la règle dotée de cette précondition.

4.3.3 Inférence d'une formule *TBox* après la règle *assignDoctor*

Les deux premières règles *receivePatient* et *assignDoctor* définies et vérifiées jusqu'à présent sont appelées en séquence depuis le point d'entrée *main* du programme. Pour traiter tous les patients et les attribuer à des médecins, les appels doivent être itératifs. Le développeur complète alors son code par des formules *TBox* avant et après chaque appel itératif. Il renforce ainsi la vérification *ABox* par ce niveau ensembliste *TBox* comme présenté à la figure 4.19.

La première *TFormule !(InQueue = empty)* signifie que l'ensemble *InQueue* ne doit pas être vide avant l'exécution du programme. La deuxième *TFormule* stipule que l'ensemble *InQueue* est vide après l'appel itératif de *receivePatient*, que tout patient est inscrit dans un département, que tous les médecins travaillent dans un moins un département et qu'aucun patient n'a un médecin traitant. La troisième *TFormule (only refDr- Patient)* $\subseteq$ *Doctor* est telle que les patients ont des

```
main {
  assert: !(InQueue = empty);
  receivePatient!;
  assert: InQueue = empty
           and (some registers- Patient) ⊆ Department
           and (some worksIn Department) = Doctor
           and (only refDr- Patient) = empty;
  assignDoctor!;
  assert: (only refDr- Patient) ⊆ Doctor;
   }
```

FIGURE 4.19 – Point d'entrée du programme *Hospital*

médecins traitants après l'appel itératif d'*assignDoctor*. On notera que cette formule n'établit pas une correspondance entre un patient et son médecin traitant, mais considère l'ensemble des patients et l'ensemble des médecins traitants, et la relation de subsomption qui les relie. Associer un et un seul médecin à un patient est assuré par le fait *dr treats p* en postcondition de la règle *assignDoctor*.

Considérant la deuxième *TFormule* en tant que prémisse de l'appel *assignDoctor*!, le moteur *TBox* infère statiquement :

$$\begin{gathered} \textit{InQueue} = \textit{empty and} \\ \textit{(some registers- Patient)} \subseteq \textit{Department and} \\ \textit{(some worksIn Department)} = \textit{Doctor and} \\ \textit{(only refDr- Patient)} \subseteq \textit{Doctor} \end{gathered}$$

La troisième *TFormule* de la figure 4.19 est ainsi validée par le moteur d'inférence. Qui plus est, le développeur peut reprendre à son compte les autres inclusions et égalités d'ensembles établies par l'appel *receivePatient*!.

4.3.4 Localisation des erreurs dans la règle *allocateRoom*

La troisième règle affecte une chambre au patient dans le département où il est inscrit. Afin de sélectionner un patient p ne disposant pas encore de chambre r, il ne suffit pas d'écrire p !*allocated* r. Cette écriture ne vérifie pas que p n'est affecté à aucune chambre, mais sélectionne une chambre r non attribuée au patient p. Il est ainsi inévitable d'écrire dans la précondition le fait p : (= 0 *allocated Room*). De plus, cette même précondition doit s'assurer pour le filtrage du graphe que le patient p est déjà affecté à un médecin traitant, soit *p refDr d*. Une première version de cette règle est donnée à la figure 4.20 et soumise au composant de preuve qui produit le contre-exemple de la figure 4.21.

```
rule allocateRoom {
 pre: p : Patient and r : Room&AvailableRoom
      and r  isIn  dep and dep  registers  p
      and p refDr d
      and p : (= 0  allocated  Room);
 add(p  allocated  r);
 delete(r : AvailableRoom);
 post: p : Patient and r : Room&!AvailableRoom
       and p  allocated  r
       and p : (= 0  allocated  Room);
}
```

FIGURE 4.20 – Première version de la règle *allocateRoom*

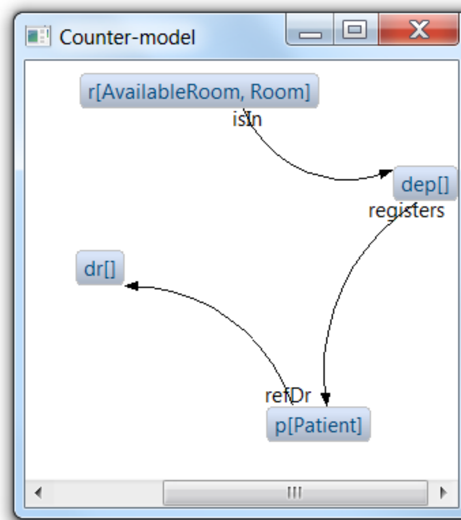


FIGURE 4.21 – Contre-exemple de la règle *allocateRoom* de la figure 4.20

Pour localiser l'erreur, le développeur choisit ce graphe comme graphe source du générateur de tests. L'analyseur dynamique exécute ainsi le code de la règle sur ce graphe. Le graphe cible et le test généré de la postcondition sont donnés à la figure 4.22.

L'échec de la quatrième assertion *assertEqualNumberOutgoingEdge* indique que l'erreur provient de la restriction $p : (= 0$ *allocated Room*). Après l'instruction *add(p allocated r)*, le nombre d'arcs vers le concept *Room* doit être incrémenté d'une unité afin de refléter l'association entre le patient p et la chambre r. En conséquence, le développeur peut corriger cette restriction en écrivant $p : (= 1$ *allocated Room*) ; une alternative consiste à la supprimer dès lors que le fait p *allocated* r de la

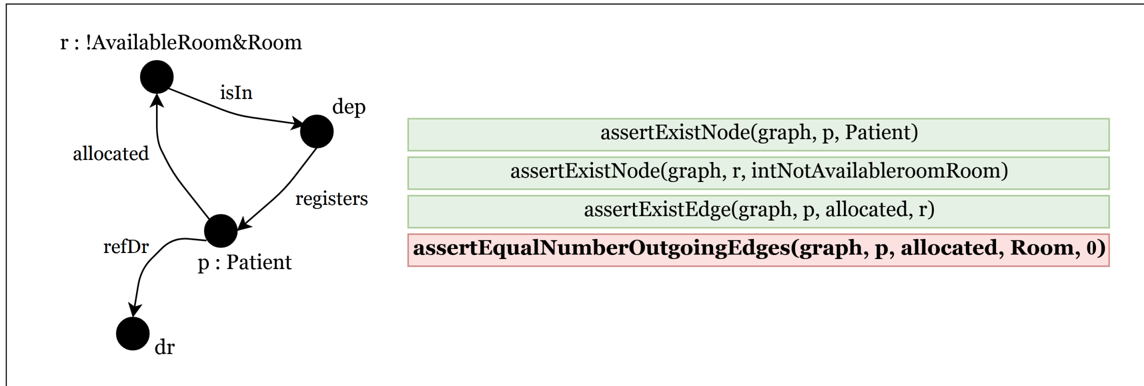


FIGURE 4.22 – Graphe cible et test associé à la postcondition de la règle *allocateRoom*

```
rule allocateRoom {
 pre: p : Patient and r : Room&AvailableRoom
      and r  isIn  dep and dep  registers  p
      and p refDr d
      and p : (= 0  allocated  Room);
 add(p  allocated  r);
 delete(r : AvailableRoom);
 post: p : Patient and r : Room&!AvailableRoom
       and p  allocated  r;
}
```

FIGURE 4.23 – Version finale de la règle *allocateRoom*

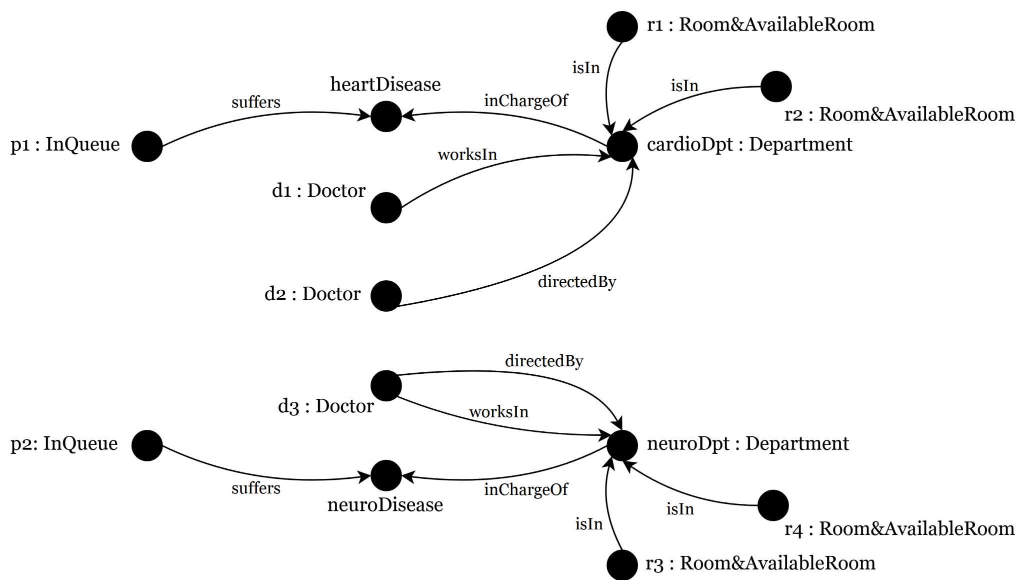
postcondition type l'arc entre p et r par le rôle *allocated*. La version finale et correcte de la règle est donnée à la figure 4.23.

4.3.5 Vérification dynamique des formules *TBox*

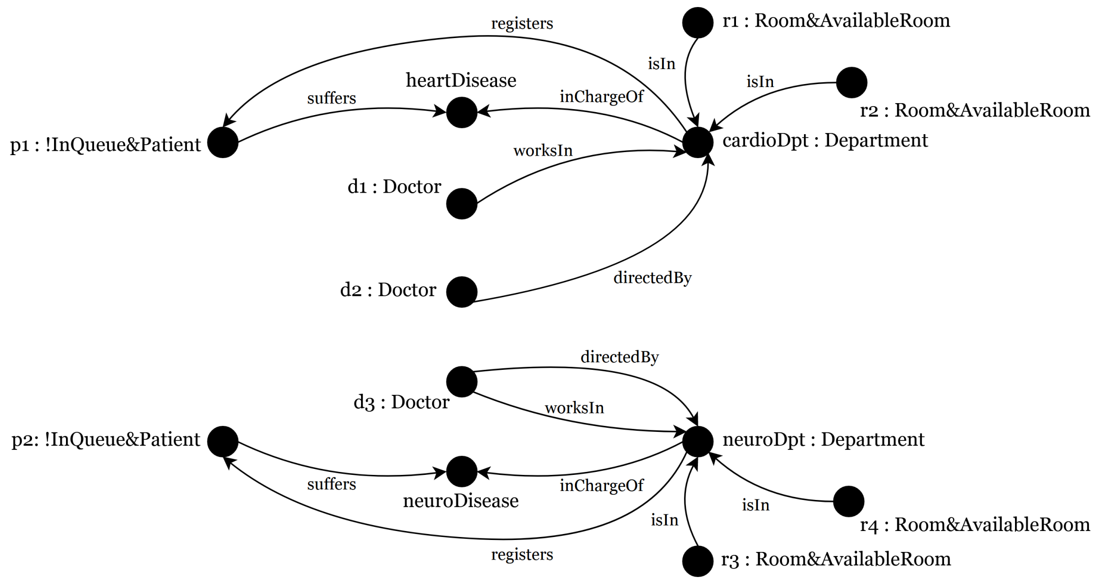
La figure 4.24 correspond à une nouvelle version du point d'entrée du programme en y ajoutant l'appel itératif de la règle *allocateRoom* et en complétant la troisième *TFormule* par les faits inférés après l'appel *assignDoctor* ! comme explicité à la section 4.3.3. En amont de l'appel itératif *allocateRoom* !, la *TFormule* ajoutée confirme l'absence de placement des patients, soit *(only allocated- Patient)* = *empty* ; en aval, le développeur exprime maintenant que certaines chambres sont désormais occupées, soit *(only allocated- Patient)* ⊆ *Room* et qu'il existe toujours des chambres libres à l'hôpital, ce qui se traduit par !*(AvailableRoom* = *empty)*. Le développeur souhaitant vérifier l'ensemble des *TFormules* sur son graphe source, active le raisonneur *TBox* en définissant comme graphe source le graphe de la figure 4.25a.

```
main {
  assert: !(InQueue = empty);
  receivePatient!;
  assert: InQueue = empty
          and (some registers- Patient) ⊆ Department
          and (some worksIn Department) = Doctor
          and (only refDr- Patient) = empty;
  assignDoctor!;
  assert: (only refDr- Patient) ⊆ Doctor
          and (some isIn Department) ⊆ AvailableRoom
          and (only allocated- Patient) = empty;
  allocateRoom!;
  assert: (only allocated- Patient) ⊆ Room
          and !(AvailableRoom = empty);
}
```

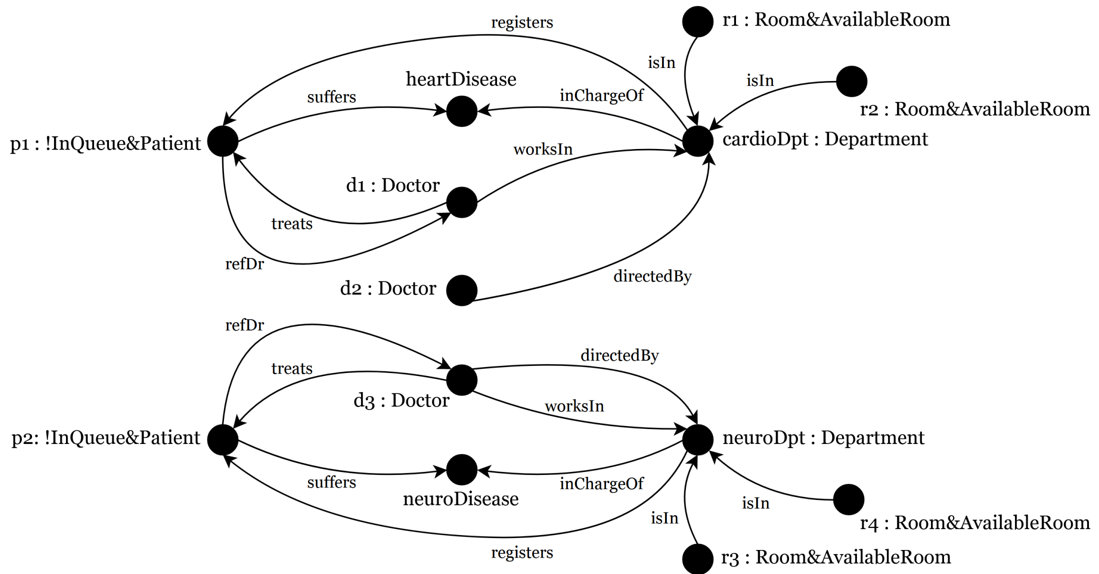
FIGURE 4.24 – Point d'entrée du programme *Hospital*



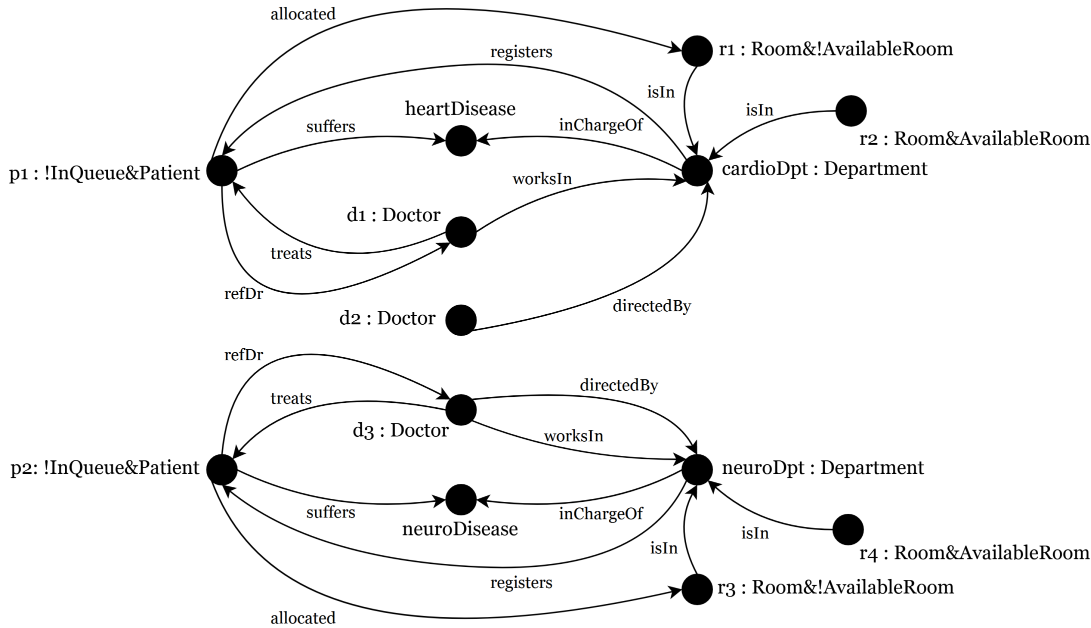
(a) Graphe source



(b) Mise à jour du graphe après *receivePatient*!



(c) Mise à jour du graphe après *assignDoctor*!



(d) Graphe cible

FIGURE 4.25 – Évolution du graphe après chaque règle

Le raisonneur *TBox* indique que toutes les *TFormules* sont valides sauf la deuxième où le fait *TBox (some worksIn Department) = Doctor* ne l'est pas. En effet, le graphe source montre que le nœud $d2$ de concept *Doctor* n'a pas d'arc sortant *worksIn* vers un nœud de concept *Departement*. Le développeur s'aperçoit ainsi qu'il a marqué le médecin $d2$ en question comme chef du département sans indiquer qu'il y est rattaché. L'erreur provient ainsi du graphe source défini ; ce type d'erreur n'est pas détectable par le moteur d'inférence *TBox* qui ne s'attache pas à valider le format d'un graphe source spécifique. Le développeur ajoute l'arc concerné à son graphe source et relance le raisonneur qui valide toutes les *TFormules*.

Grâce aux composants de l'environnement Small-t$\mathcal{ALC}$, le développeur a pu atteindre un programme de transformation correct gérant le service d'accueil d'un hôpital. Ces outils peuvent être exploités selon le choix d'un développeur dans d'autres scénarios que celui décrit. Cependant, bien qu'un développeur puisse utiliser tout composant de l'environnement d'assistance au cours de la construction de son programme, ces composants sont indépendants et n'interagissent pas entre eux. Par exemple, une interaction entre le prouveur et le générateur de cas de test est envisageable et simple à mettre en œuvre en considérant automatiquement le contre-exemple produit comme donnée de test, permettant ainsi de fournir directement de riches rétroactions aux développeurs.

4.4 Autres environnements de transformation de graphes

Cette section a pour objectif de positionner Small-t$\mathcal{ALC}$ vis-à-vis de certains outils de transformation existants [1, 2, 3, 4, 5, 7, 19, 54, 69, 73, 78, 100, 101, 108, 115, 120]. Elle compare les techniques de vérification délayées, après avoir présenté succinctement à la sous-section 4.4.1 les outils que nous avons retenus pour cette comparaison. Le positionnement de nos travaux est décrit aux sous-sections 4.4.2 et 4.4.3.

4.4.1 Outils existants

Parmi les outils existants, seuls ceux qui mettent en avant la vérification et la validation sont analysés : AGG, Augur, Epsilon, Fujaba, GP 2, GROOVE, Henshin, PROGRES et UML-RSDS.

AGG

AGG (*The Attributed Graph Grammar System*) [1, 112, 117, 118] est un environnement de transformation de graphes basé sur l'approche catégorique *single pushout* (SPO). Il consiste en une interface graphique comprenant plusieurs éditeurs visuels, un interprète et un ensemble d'outils de validation. Un graphe AGG est un graphe typé constitué de nœuds pouvant être attribués par des objets Java.

Les outils de validation AGG sont basés sur une analyse de paires critiques, c'est-à-dire des paires de règles de transformation en conflit. Ces outils proposent un contrôle de cohérence des graphes, une vérification de l'applicabilité de séquences de règles et une détection de conflits et de dépendances entre règles. Trois types de conflit sont identifiés par l'analyse statique AGG ; les deux premiers sont liés à la structure du graphe et le dernier concerne ses attributs :

1. Une règle entraîne la suppression d'un élément du graphe ayant une occurrence en membre gauche d'une autre règle.
2. Une règle produit un graphe associé à une condition d'application négative d'une autre règle.
3. Une règle modifie les attributs qui correspondent au membre gauche d'une autre règle.

La détection des conflits d'un ensemble de règles n'affecte pas l'applicabilité des règles dès lors qu'AGG les applique dans un ordre prédéfini plusieurs fois jusqu'à ce qu'aucune ne puisse être appliquée. Ainsi, si un conflit entraîne la non application d'une règle, AGG l'ignore et procède à la suivante.

Augur

Dès lors que les systèmes de transformation de graphes étendent les structures de graphes statiques avec la possibilité de décrire les changements dynamiques à l'aide de règles de transformation, ils s'adaptent à la description du comportement dynamique des systèmes concurrents et distribués.

Dans ce contexte, l'outil Augur [68] a été conçu pour la vérification de systèmes dynamiques traduits en tant que transformations de graphes en les sur-approximant par des réseaux de Petri. La nouvelle version Augur 2 [69, 71], permet la manipulation des graphes attribués, c'est-à-dire des graphes avec des attributs (entier et chaîne de caractères) associés aux nœuds et aux arcs.

Ces travaux s'intéressent essentiellement à des graphes exprimant des états infinis d'un système. Ils exploitent la vérification par *model-checking*.

Epsilon

Epsilon [3] est un ensemble de langages et d'outils de génération de code, de transformation de modèles à modèles, de validation de modèles, de comparaison, de migration et de restructuration opérationnels avec EMF, UML, Simulink, XML et autres types de modèles. Parmi cet ensemble, EVL (*Eclipse Validation Language*) [70] est un langage de validation basé sur le langage à objets Epsilon EOL (*Epsilon Object Language*). Les contraintes d'EVL sont similaires à celles d'OCL. Cependant, EVL considère les dépendances entre contraintes. Ainsi si la contrainte A échoue, la contrainte B ne sera pas évaluée. En cas d'échec de la validation, des messages d'erreur sont affichés, ainsi que des suggestions de correction que les développeurs peuvent invoquer pour détecter les incohérences.

Fujaba

L'environnement Fujaba (*From UML to Java and Back Again*) [4] est très répandu dans le domaine des transformations de modèles. Il est utilisé pour la génération automatique de code Java à partir des diagrammes UML et inversement. Geiger et al. [53] exploitent les tests unitaires afin de vérifier la cohérence entre les scénarios des cas d'utilisation et leur implémentation en Fujaba. Il s'agit de générer des tests JUnit à partir des scénarios décrits, et de vérifier si les programmes correspondants s'exécutent correctement.

GP 2

GP 2 [101] est une nouvelle version du langage GP (*Graph Programming*) [100], un langage de transformation de graphes expérimental non-déterministe basé sur l'approche *double pushout* (DPO) avec réétiquetage (*relabelling*). Un programme GP consiste en un ensemble de schémas de règles conditionnelles, c'est-à-dire des règles accompagnées de conditions sur le graphe étiqueté à transformer. Les conditions sont typées par des entiers ou des chaînes de caractères. La figure 4.26 spécifie la règle conditionnelle *bridge* qui déclare trois nœuds x, y, et z, et deux arcs a et b reliant respectivement x et y, et y et z, et qui ajoute l'arc $a + b$ entre x et z [100].

Afin de vérifier les transformations GP, Poskitt et al. [104] adoptent la logique de Hoare en exprimant les pré- et postconditions par des *E-conditions*, c'est-à-dire des conditions imbriquées étendues à des expressions et des contraintes sur les étiquettes.

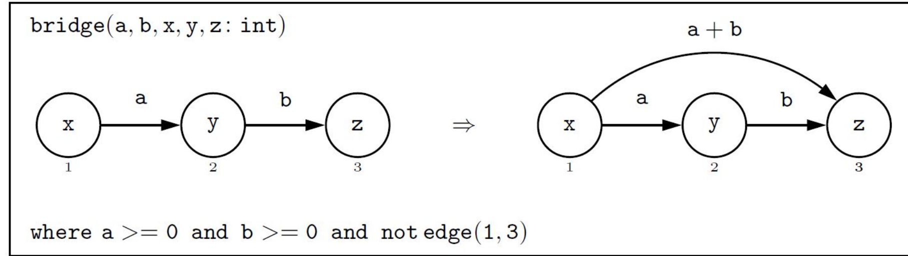


FIGURE 4.26 – Schéma de règle conditionnelle en GP [100]

Le programme *colouring* de la figure 4.27 attribue aux nœuds source et cible d'un arc des couleurs distinctes. Les couleurs sont modélisées par des entiers attachés aux nœuds. La précondition de la transformation indique que tout nœud du graphe est étiqueté par un entier. La postcondition affirme que l'étiquette e d'un nœud est formée d'une séquence de deux entiers notée $e1_e2$ et que deux nœuds adjacents se caractérisent par des séquences distinctes. Les *E-conditions* correspondantes aux pré- et postconditions de ce programme sont les suivantes :

Précondition : $\neg\exists\left(\text{ⓐ} \mid \texttt{not type(a) = int}\right)$

Postcondition : $\forall\left(\text{ⓐ}_1,\ \exists\left(\text{ⓐ}_1 \mid \texttt{a = b_c and type(b,c) = int}\right)\right)$

$\wedge\ \neg\exists\left(\text{(x_i)} \xrightarrow{k} \text{(y_i)} \mid \texttt{type(i,k,x,y) = int}\right)$

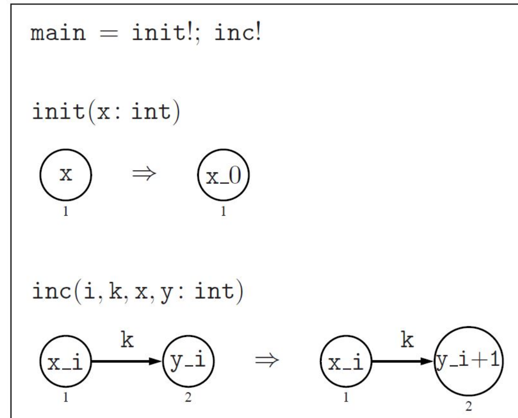


FIGURE 4.27 – Programme *colouring* en GP [104]

Récemment, Poskitt et al. [105] étendent ces *E-conditions* en des *M-conditions* afin d'exprimer des propriétés monadiques du second ordre pour vérifier des spécifications globales sur le graphe,

tel que l'acyclicité, la présence des chemins, etc. Il s'agit d'introduire des quantificateurs sur les ensembles des nœuds et d'arcs du graphe et de décorer les morphismes par des contraintes sur ces ensembles.

GROOVE

GROOVE (*GRaph-based Object-Oriented VErification*) [108] est un ensemble d'outils pour la transformation de graphes et la génération de graphes d'états basés sur l'approche SPO. La fonctionnalité principale de GROOVE est d'appliquer récursivement toutes les règles prédéfinies sur un graphe source et sur tous les graphes générés. Le résultat sera ainsi un espace d'états formé de tous les graphes produits. L'ordre d'application des règles est arbitraire. Un contrôle prioritaire des règles peut être choisi soit directement en attribuant des priorités aux règles de telle sorte qu'une règle ne peut s'appliquer que si aucune règle plus prioritaire n'est applicable, soit en exploitant le langage de contrôle de GROOVE. Par exemple, le programme $a; try\{b;\}; else\{c;\}$ exécute initialement la règle a, puis la règle b. Si b n'est pas applicable alors c s'applique. Si a n'est déjà pas applicable, aucune règle ne sera appliquée. Les autres structures de contrôle sont plus conventionnelles ; elles concernent le bouclage tant que possible, le choix aléatoire entre règles, et l'appel simple non récursif d'une règle. GROOVE explore et stocke tout l'espace d'états des graphes atteignables, ce qui permet la vérification formelle des transformations par *model-checking* [55].

Henshin

Henshin [5, 12] est un langage de modélisation et un ensemble d'outils basé sur l'approche DPO des transformations de graphes. Il est fondé sur le modèle de données d'EMF (*Eclipse Modeling Framework*). L'environnement Henshin se compose principalement d'un moteur de transformations, de plusieurs éditeurs et d'un générateur d'espaces d'états pour un raisonnement par *model- checking*.

PROGRES

PROGRES (*PROgrammed Graph REwriting Systems*) [106, 114, 115] est l'un des premiers systèmes de transformation de graphes conçus. Il permet la modélisation de la structure et du comportement des systèmes d'une manière hybride : visuelle et textuelle. Cet environnement, dédié aux graphes orientés et étiquetés comme en Small-t$\mathcal{ALC}$, inclut un éditeur syntaxique, un interprète et un metteur au point. Conformément à l'approche SPO, une transformation de graphe en PROGRES est décrite graphiquement par une règle munie d'un membre gauche et d'un membre droit décorés éventuellement par des conditions d'application textuelles.

Le système PROGRES offre aux développeurs un analyseur sémantique des instructions de transformation. Par exemple, la figure 4.28 montre un message d'erreur émis par l'analyseur qui détecte que le nom *overlapping* du chemin n'est pas déclaré.

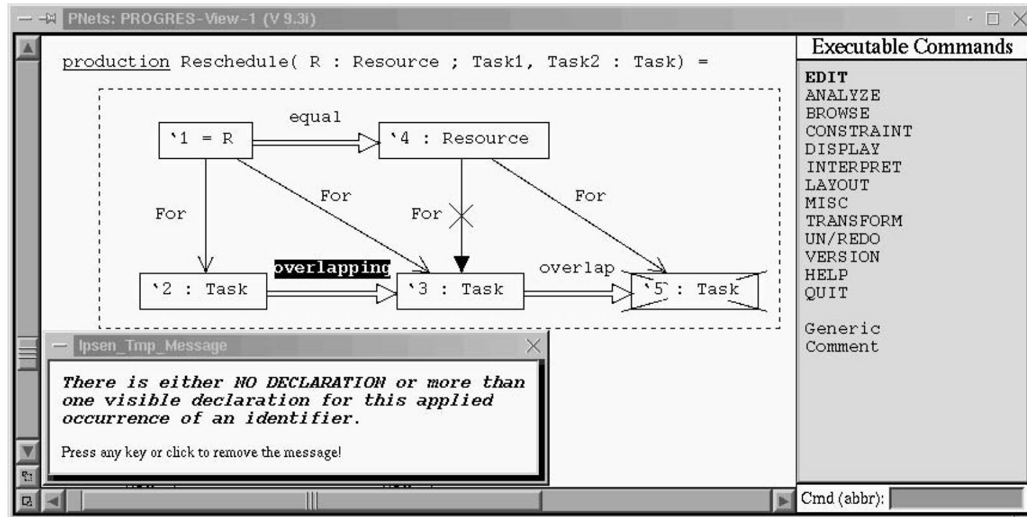


FIGURE 4.28 – Erreur sémantique en PROGRES

Afin d'aider les développeurs à détecter et éliminer les erreurs, il procède de manière incrémentale et fournit quelques suggestions de mise à jour du code [94].

UML-RSDS

UML-RSDS (*Reactive System Development Support*) [73] est un langage et un outil conçus pour un développement basé sur les modèles. Le langage exploite UML et OCL pour la modélisation des systèmes à haut niveau d'abstraction. Ces derniers peuvent ainsi être spécifiés par des diagrammes de classes UML, de cas d'utilisation, d'états-transitions et d'activités avec des préconditions, post-conditions et invariants exprimés en OCL [74]. En UML-RSDS, la dérivation d'une conception correcte par construction d'un système à partir de la spécification OCL peut être automatisée [75].

Lano et al. [77] ont prouvé des propriétés sur les transformations UML-RSDS par la méthode B [76], telles que la correction syntaxique et la préservation sémantique des transformations à partir desquelles un code Java peut être généré automatiquement.

4.4.2 Comparatif avec Small-t$\mathcal{ALC}$

Les outils dédiés aux graphes sont généralement universels. Ils s'adressent à tous les domaines qui peuvent être représentés par des graphes. A l'inverse, les outils dédiés aux modèles sont centrés sur la modélisation des programmes. Le monde des graphes est plus ancien, plus mature, plus formel que le monde des modèles [65].

TABLE 4.4 – Comparaison entre Small-t$\mathcal{ALC}$ et les autres outils présentés

Outil	Domaine	Validation et vérification
Small-t$\mathcal{ALC}$	Graphe	Logique de Hoare Analyses statiques et dynamiques Tests
AGG	Graphe	Analyse de paires critiques
Augur	Graphe	Énumération de l'espace d'états Sur-approximation par réseaux de Petri
Epsilon	Multi	Langage de validation
Fujaba	Multi	Tests
GP 2	Graphe	Logique de Hoare
GROOVE	Graphe	Énumération de l'espace d'états
Henshin	Modèle	Énumération de l'espace d'états
Progres	Graphe	Analyse statique
UML-RSDS	Modèle	*Theorem prover*

Le tableau 4.4 résume le domaine d'application de chaque outil décrit à la sous-section précédente, et mentionne les techniques de vérification et de validation exploitées par chacun. Il complète le comparatif des outils existants initialement proposés par Jakumeit et al. [65], en y incluant notre environnement Small-t$\mathcal{ALC}$. D'autres synthèses sur ces techniques existent [11, 8], sans toutefois établir un parallèle exhaustif entre les outils.

GP 2

Le langage GP 2 [104] est le plus proche de Small-t$\mathcal{ALC}$. Il permet d'exprimer des pré- et postconditions portant sur les entités manipulées afin de vérifier une règle, ce qui correspond aux pré- et postconditions *ABox* d'une règle Small-t$\mathcal{ALC}$. Un composant de preuve a été développé pour automatiser l'examen des règles [72, 98]. GP 2 manipule de plus des expressions associées aux nœuds et arcs, ce que ne permet pas Small-t$\mathcal{ALC}$.

Comme pour nos formules *TBox*, GP 2 propose la manipulation de variables dénotant des ensembles, en logique monadique du second ordre, ce qui permet l'écriture de propriétés globales

sur le graphe transformé. Considérons par exemple, la *M-contrainte* suivante caractérisant un graphe biparti :

$$\exists_V \mathtt{X,Y}\ [\ \forall(\ \bullet_v\ ,\ \exists(\ \bullet_v\ \mid (\mathtt{v}\in\mathtt{X}\ \mathtt{or}\ \mathtt{v}\in\mathtt{Y})\ \mathtt{and\ not}\ (\mathtt{v}\in\mathtt{X}\ \mathtt{and}\ \mathtt{v}\in\mathtt{Y})))$$
$$\wedge\ \forall(\ \bullet_v\ \ \bullet_w\ ,\exists(\ \bullet_v\!\!\rightarrow\!\!\bullet_w\ )\Rightarrow\exists(\ \bullet_v\ \ \bullet_w\ \mid \gamma_{\mathtt{col}}))\ ]$$

Un graphe G satisfera cette contrainte s'il existe deux sous-ensembles X et Y de nœuds tels que :

1. Chaque nœud de G appartient exactement à l'un des deux ensembles.
2. S'il existe un arc reliant deux nœuds, alors ces derniers n'appartiennent pas à un même ensemble.

En Small-t$\mathcal{ALC}$, un graphe biparti s'exprime par la formule :

$$X \cup Y = all\ and\ X \cap Y = empty$$
$$and\ (some\ r\ X) \cap X = empty\ and\ (some\ r\ Y)\ \cap\ Y = empty$$

Les *M-contraintes* GP 2 introduisent des quantificateurs sur les entités du graphe pour exprimer des propriétés globales. En Small-t$\mathcal{ALC}$, ces propriétés sont prises en compte implicitement par les relations de subsomption *TBox* suite à la mise à jour des faits *ABox*.

En ce qui concerne la décidabilité [103], les pré- et postconditions GP 2 sont des formules du premier ordre, donc non-décidables [52]. Les *M-contraintes*, formules de la logique du second ordre, sont aussi non-décidables, comme les formules *TBox* en Small-t$\mathcal{ALC}$.

AGG

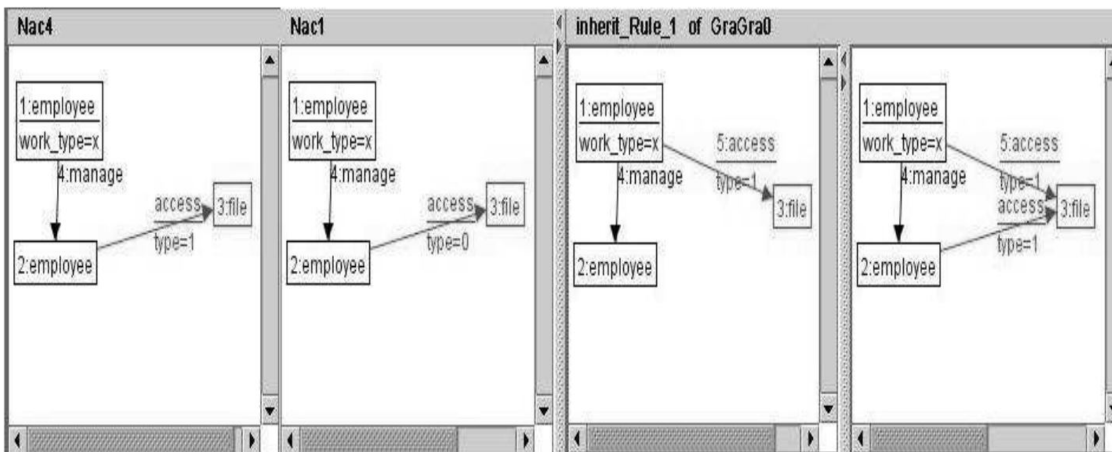
Les conflits détectés par l'analyseur statique d'AGG [118] concernent des modifications effectuées par la règle pouvant conduire à un graphe empêchant l'application d'autres règles. En Small-t$\mathcal{ALC}$, ces contraintes peuvent se reporter dans la postcondition de la règle. Autrement dit, pour vérifier qu'une règle ne conduit pas à un graphe satisfaisant une condition négative NAC de la règle qui suit, la postcondition peut être renforcée pour indiquer la non-existence d'un tel modèle.

Soient les deux règles de transformation AGG de la figure 4.29 qui donnent l'autorisation à des employés d'accéder à des fichiers :

1. La première règle permet au nœud *employee* d'accéder au nœud *file* par un arc ayant un attribut *type* égal à 1. Cette règle est soumise, de plus, à une condition booléenne interne $x = 3$ non explicitée dans le membre gauche de la règle. Dès lors, seul le nœud *employee* ayant $work_type = 3$ est concerné par cette règle. La condition d'application négative de cette règle est codée par la première case à gauche, et les membres gauche et droit de la règle sont reportées respectivement aux cases du milieu et de droite de la figure.
2. La deuxième règle permet à un nœud *employee* d'accéder au nœud *file* si son gestionnaire a le droit d'accès. Les deux cases de gauche correspondent à des conditions d'application

négatives qui interdisent l'application de cette règle dans le cas où un arc *access* de valeur 0 ou 1 existe déjà entre *employee* et *file*. Comme précédemment, les deux cases de droite exhibent la transformation proprement dite.

(a) Première règle



(b) Deuxième règle

FIGURE 4.29 – Deux règles AGG en conflit [96]

Ces deux règles illustrent le deuxième type de conflit détecté par l'analyse critique des paires en AGG : la deuxième règle donne à tout employé le droit d'accès à un fichier quelle que soit la valeur de *work_type*, si son gestionnaire en a le droit. En revanche, la première règle interdit l'accès au fichier si *work_type* est différent de 3.

En Small-t$\mathcal{ALC}$, ce problème d'analyse statique de paires critiques n'a pas été abordé. En revanche, étant donné que le langage est capable d'exprimer la négation des concepts et des rôles, ainsi que des restrictions, ce conflit peut être résolu manuellement par un développeur. Le sous-graphe représenté par un NAC d'une règle $r1$ s'exprime par sa précondition Small-t$\mathcal{ALC}$ qui indique la non-existence d'un tel graphe ; dans le même temps, sa postcondition peut spécifier la non-existence du sous-graphe qui représente le NAC de la règle $r2$ qui la suit. Une vérification automatique par analyse statique de la cohérence entre la postcondition de $r1$ et la précondition de $r2$ peut être aisément mise en œuvre en Small-t$\mathcal{ALC}$.

Par ailleurs, Small-t$\mathcal{ALC}$ ne manipule pas actuellement des attributs associés aux nœuds comme en AGG, mais ces attributs peuvent être exprimés par des nœuds liés par des relations *attribute* aux nœuds concernés. Par exemple, en considérant qu'un employé ayant *work_type = 3* est codé par un nœud e de concept *Employee* relié à un nœud *work_type*_3 par l'arc *attribute*, la postcondition de la deuxième règle s'exprime par :

e : Employee and e !attribute work_type_3 and e : (=0 access File)

La dernière restriction vérifie qu'un employé e n'ayant pas *work_type = 3* n'a accès à aucun fichier. Un développeur peut ainsi adapter sa règle Small-t$\mathcal{ALC}$ pour que la postcondition comportant cette formule soit valide.

Ces conflits qui concernent principalement l'applicabilité des règles peuvent aussi être identifiés par notre moteur d'inférence statique *TBox* qui examine les propriétés ensemblistes à satisfaire avant et après les règles pour assurer leur bon enchaînement.

PROGRES

PROGRES [106] dispose d'un analyseur statique conventionnel permettant de vérifier la syntaxe et la sémantique des transformations en analysant les opérations et les instructions d'une transformation complexe. Le développeur peut aussi itérativement annuler et restaurer (*undo&redo*) des étapes de transformation afin d'identifier les opérations erronées. L'environnement Small-t$\mathcal{ALC}$ ne propose pas d'assistance syntaxique à l'écriture des transformations. Par contre, les erreurs sémantiques, ainsi que la violation des spécifications peuvent être identifiées par les analyseurs statique et dynamique comme illustré à l'exemple de la section 4.3.1 de ce chapitre, où l'écriture erronée d'une étiquette d'un arc a causé l'échec de la vérification d'une règle. L'analyse dynamique de la règle a ainsi aidé le développeur à l'identifier.

Augur, GROOVE et Henshin

Augur [71], GROOVE [55] et Henshin [12] vérifient une transformation selon l'approche *model-checking*. Cette approche est souvent confrontée à une explosion combinatoire de l'espace d'états, ce qui rend son exploration sensible. Small-t$\mathcal{ALC}$, qui vérifie les transformations avec la logique de

Hoare en exploitant son composant de preuve, offre une panoplie d'outils complémentaires visant à localiser la cause d'une anomalie, comme le générateur de cas de test *ABox* et le raisonneur *TBox*. De plus, comme souligné au chapitre 2, une démarche auto-active d'écriture et de mise au point des règles est envisageable.

Augur abstrait des graphes en des réseaux de Petri par le biais d'une relation de simulation entre les graphes accessibles et les marques accessibles du réseau obtenues par tirage des transitions. Small-t$\mathcal{ALC}$ ne sur-approxime pas les graphes à transformer, mais exploite l'interprétation abstraite en considérant les ensembles d'appartenance des nœuds du graphe pour permettre la vérification *TBox*.

Epsilon, Fujaba et UML-RSDS

Les autres outils valident et vérifient des transformations dans d'autres contextes que celui de Small-t$\mathcal{ALC}$. Le langage EVL de l'environnement Epsilon [70] est dédié à la validation des modèles et méta-modèles en Epsilon. Il étend le langage OCL et permet, comme en Small-t$\mathcal{ALC}$, la définition d'invariants, de pré- et postconditions et d'autres contraintes. Il partage avec Small-t$\mathcal{ALC}$ la notion d'assistance au développeur pour localiser et corriger les erreurs.

Fujaba [53] exploite les transformations de modèles pour la conception multi-paradigme des programmes. Il partage avec Small-t$\mathcal{ALC}$ la notion de validation par la génération de tests unitaires en JUnit. Ces tests sont exploités pour valider la mise en œuvre de scénarios décrits par les diagrammes de cas d'utilisation. Si leurs tests permettant de montrer la présence d'incohérences, ils ne localisent pas les erreurs dans les transformations comme en Small-t$\mathcal{ALC}$.

UML-RSDS [77] exploite OCL pour décrire les contraintes considérées comme des invariants sur les modèles. Bien qu'OCL exprime de manière déclarative des contraintes significatives, il ne permet pas de retours significatifs aux développeurs dans le cas d'échec de vérification de certaines contraintes. Il se limite à signaler l'invariant ou l'instance erronés. Ceci implique qu'un développeur doit être un expert OCL pour pouvoir identifier les sources d'erreurs dans ses transformations, contrairement à Small-t$\mathcal{ALC}$ qui s'appuie sur le même formalisme logique pour son code et ses spécifications.

4.4.3 Synthèse

De nombreux outils de transformation de graphes et d'autres dédiés aux transformations de modèles basées sur la réécriture de graphes sont proposés, tout en mettant à disposition des techniques pour vérifier leurs programmes. Les outils présentés dans cette section sont énumérés à la figure 4.30 en les positionnant par rapport à Small-t$\mathcal{ALC}$, du plus proche au plus éloigné selon le critère de vérification.

GP 2 et Epsilon spécifient leurs transformations par des pré- et postconditions comme en Small-t$\mathcal{ALC}$. GP 2, qui est le plus proche de Small-t$\mathcal{ALC}$, exploite aussi la logique monadique du second

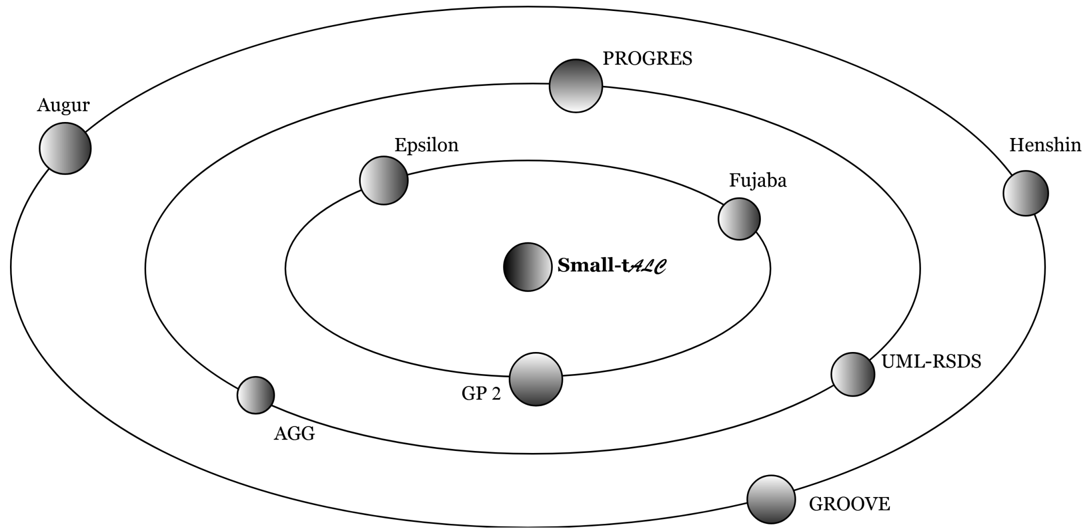


FIGURE 4.30 – Positionnement des outils existants par rapport à Small-t$\mathcal{ALC}$

ordre pour vérifier en plus des propriétés globales. Fujaba vérifie les transformations en générant des tests unitaires comme le générateur de cas de test Small-t$\mathcal{ALC}$.

PROGRES, AGG et UML-RSDS sont plus éloignés de Small-t$\mathcal{ALC}$. PROGRES partage avec Small-t$\mathcal{ALC}$ la possibilité de construire itérativement des transformations en exploitant des analyses statiques. AGG effectue une analyse statique des paires critiques, ce qui n'est pas abordé par Small-t$\mathcal{ALC}$, mais peut être établi manuellement. UML-RSDS, dédié aux transformations de modèles, ne vérifie que des invariants et ne fournit pas de retours significatifs pour aider les développeurs à corriger leurs transformations.

Augur, GROOVE et Henshin déploient la technique *model-checking*, éloignée de Small-t$\mathcal{ALC}$.

4.5 Conclusion

Small-t$\mathcal{ALC}$ se distingue par son environnement d'assistance en offrant au développeur des composants formels et d'autres semi-formels, tous bâtis sur le même formalisme $\mathcal{ALCQI}$. Ce chapitre s'est focalisé sur l'assistance semi-formelle, complémentaire à celle proposée dans les deux chapitres précédents. Étant donnée une règle Small-t$\mathcal{ALC}$, un générateur de cas de test *ABox* permet d'identifier les erreurs dans le code. Ce générateur peut s'envisager par analyse statique ou dynamique, selon le besoin du développeur. Quant au niveau *TBox*, un raisonneur est mis en œuvre pour vérifier dynamiquement la validité de chaque formule *TBox* donnée dans le point d'entrée du programme vis-à-vis d'un graphe source.

Conclusion et perspectives

Dans le cadre du projet CLIMT (*Categorical and Logical Methods in Model Transformation*), un langage basé sur la logique $\mathcal{ALCQI}$, appelé Small-t$\mathcal{ALC}$, a été conçu permettant d'écrire des règles de transformation à la Hoare, c'est-à-dire sous la forme d'une précondition P, d'un code S et d'une postcondition Q, soit $\{P\}S\{Q\}$. Ce langage s'accompagne d'un composant de preuve chargé d'automatiser le processus de vérification d'un triplet, en calculant la plus faible précondition $wp(S, Q)$ à l'égard du code S et de la postcondition Q, et en s'assurant que la précondition du triplet est plus forte que la formule calculée, soit $P \Rightarrow wp(S, Q)$. Dans le cas où cette implication n'est pas vérifiée, un contre-exemple est produit.

Vu que peu de praticiens sont familiers avec les méthodes formelles et que la rétroaction du prouveur se limite à un contre-exemple peu exploitable, cette thèse vise à compléter l'environnement existant dans l'objectif d'assister les développeurs à construire des transformations Small-t$\mathcal{ALC}$ correctes, en considérant à la fois les propriétés du sous-graphe à filtrer par la règle et les propriétés du graphe transformé suite à un enchaînement de règles. Dans ce contexte, les instructions du langage sont revisitées selon l'angle *ABox* et *TBox* des logiques de description : la transformation des éléments du graphe s'assimile à la mise à jour *ABox* des individus et de leurs relations appartenant aux concepts et rôles de la *TBox*.

Bilan des contributions

Afin de compléter l'environnement CLIMT, plusieurs outils *ABox* et *TBox* ont été développés : les outils *ABox* permettent la construction de règles de transformation correctes, et les outils *TBox* vérifient l'effet global des règles sur le graphe ainsi que leur bon enchaînement. Ces outils forment ensemble un environnement d'assistance au développement des programmes Small-t$\mathcal{ALC}$. Trois contributions de la thèse se dégagent :

1. **Extraction des préconditions *ABox* d'une règle**
   La première contribution consiste à aider le développeur à construire une règle *ABox* en suggérant des préconditions qui rendent correct le triplet formé initialement d'un code et d'une postcondition. Le calcul des préconditions est établi par une analyse statique en exploitant les

plus faibles préconditions et en les simplifiant par un calcul d'alias qui identifie les variables de la règle pouvant désigner un même nœud du graphe. Le développeur peut ensuite choisir une des préconditions extraites *ABox* selon son intention, ou affiner son code et sa postcondition dans plusieurs itérations pour atteindre le triplet reflétant la transformation souhaitée.

2. **Vérification *TBox* des règles**
   La deuxième contribution définit un niveau de vérification des transformations plus abstrait. Il s'agit d'exploiter la *TBox* des logiques de description pour exprimer des assertions sur des ensembles de nœuds du graphe. Les propriétés spécifiques forment ainsi des formules *TBox* à vérifier avant et après l'application des règles. Un moteur d'inférence *TBox* a été développé pour raisonner sur les règles *ABox* ; il exploite l'analyse statique par interprétation abstraite pour mesurer l'effet des règles sur une formule *TBox* donnée en prémisse d'une transformation. Une dépendance entre les deux niveaux de vérification *ABox* et *TBox* est ainsi constatée. Ces deux niveaux sont complémentaires ; ils permettent conjointement de vérifier localement et globalement des transformations.

3. **Environnement d'assistance à la construction des programmes Small-t$\mathcal{ALC}$**
   La troisième contribution vise à exploiter des approches de vérification plus conventionnelles. Outre les deux outils formels présentés, il s'agit de proposer des outils moins formels permettant d'identifier des erreurs aux deux niveaux *ABox* et *TBox* d'un programme Small-t$\mathcal{ALC}$. Dans ce cadre, un générateur de cas de test *ABox* a été développé visant à localiser les faits erronés dans un triplet suite à une analyse statique ou dynamique du code selon le choix du développeur. Il s'agit de générer automatiquement des tests à partir des pré- et postconditions, ainsi qu'un ensemble de données de test produit à partir de ces spécifications. Un raisonneur *TBox* est également disponible pour vérifier dynamiquement les formules *TBox* données en exécutant le point d'entrée du programme sur un graphe source défini. Tous ces outils construisent ensemble, avec le composant de preuve développé initialement et un compilateur, l'environnement d'assistance Small-t$\mathcal{ALC}$.

L'étude proposée présente cependant des limitations :

- L'extracteur de préconditions *ABox* calcule une formule en forme normale disjonctive. Les conjonctions qui la caractérisent présentent en outre des redondances de faits et de formules logiques. Cette duplication ne favorise pas toujours la lisibilité des conjonctions proposées par l'analyseur.
- Afin d'inférer des propriétés *TBox* à partir des règles *ABox*, une prémisse *TBox* est exigée, ce qui peut s'avérer complexe à spécifier pour un développeur car il n'existe aucune aide dans notre environnement pour établir une telle prémisse.
- L'inférence de certains faits *TBox* est quelquefois indécidable. En effet, le moteur d'inférence n'est pas toujours en mesure d'inférer des propriétés à partir des faits donnés en prémisse, notamment, après l'appel itératif d'une règle dès lors qu'il la parcourt une seule fois.

Perspectives

Les travaux présentés dans cette thèse ouvrent de nombreuses perspectives. Citons en trois respectivement à court, moyen et long terme :

1. **Aide à la découverte des invariants de boucle**
   Actuellement, notre extracteur de préconditions se limite à vérifier l'invariant de boucle donné par le développeur en se basant sur les conditions de vérification. Tant que cet invariant n'est pas correct, aucune précondition ne pourra être extraite. Dans ce cadre, il serait pertinent à court terme d'aider un développeur à découvrir les invariants de boucle [44, 122], en calculant une formule *inv* qui rend correcte la condition de vérification d'une boucle par rapport à une postcondition $Q : vc(\{inv\}\ while\ c\ do\ s, Q) = (inv \wedge\ \neg c \Rightarrow Q) \wedge\ (inv \wedge\ c \Rightarrow wp(s, inv)) \wedge vc(s, inv)$.
2. **Capacités de raisonnement *TBox***
   Jusqu'à présent, les faits *TBox* en Small-t$\mathcal{ALC}$ ne considèrent que des concepts complexes et des rôles simples et inverses. Nous envisageons d'étudier l'extension du raisonnement pour des rôles complexes en considérant d'autres constructeurs de rôles [14, 23], notamment la conjonction des rôles $\mathcal{R}$, le rôle transitif $\mathcal{S}$, le rôle hiérarchique $\mathcal{H}$ et la restriction sur le nombre de rôles $\mathcal{N}$. Ceci permettrait d'accroître l'expressivité des assertions *TBox* et d'exprimer ainsi plus de propriétés du second ordre.
3. **Langage d'action en logique de description**
   Plusieurs travaux ont abordé la notion d'action au sein de langages logiques pour décrire des interactions dynamiques avec des environnements changeants [10, 83]. Small-t$\mathcal{ALC}$ permet déjà de décrire des actions *ABox* dans un contexte impératif. Nous envisageons un cadre plus général des applications que celui de la réécriture de graphes en dotant le langage de mécanismes de raisonnement. Cette perspective renvoie à la description et représentation d'actions à partir de différents effets et conditions d'exécutabilité.

Bibliographie

[1] "The attributed graph grammar system : A development environment for attributed graph transformation systems." [Online]. Available : http://tfs.cs.tu-berlin.de/agg/

[2] "Atl." [Online]. Available : http://www.eclipse.org/atl/

[3] "Epsilon." [Online]. Available : https://www.eclipse.org/epsilon/

[4] "The fujaba tool suite," 2012. [Online]. Available : https://web.cs.upb.de/archive/fujaba/

[5] "Henshin." [Online]. Available : https://www.eclipse.org/henshin/

[6] "Junit." [Online]. Available : https://junit.org/

[7] "The eclipse viatra framework." [Online]. Available : https://www.eclipse.org/viatra/

[8] L. Ab. Rahim and J. Whittle, "A survey of approaches for verifying model transformations," *Software & Systems Modeling*, vol. 14, no. 2, pp. 1003–1028, May 2015. [Online]. Available : https://doi.org/10.1007/s10270-013-0358-0

[9] S. Ahmetaj, D. Calvanese, M. Ortiz, and M. Šimkus, "Managing change in graph-structured data using description logics," *ACM Trans. Comput. Logic*, vol. 18, no. 4, pp. 27 :1–27 :35, Nov. 2017. [Online]. Available : http://doi.acm.org/10.1145/3143803

[10] J. J. Alferes, A. Brogi, J. A. Leite, and L. M. Pereira, "Evolving logic programs," in *Logics in Artificial Intelligence*, S. Flesca, S. Greco, G. Ianni, and N. Leone, Eds. Berlin, Heidelberg : Springer Berlin Heidelberg, 2002, pp. 50–62.

[11] M. Amrani, L. Lucio, G. Selim, B. Combemale, J. Dingel, H. Vangheluwe, Y. Le Traon, and J. R. Cordy, "A tridimensional approach for studying the formal verification of model transformations," in *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, ser. ICST '12. Washington, DC, USA : IEEE Computer Society, 2012, pp. 921–928. [Online]. Available : http://dx.doi.org/10.1109/ICST.2012.197

[12] T. Arendt, E. Biermann, S. Jurack, C. Krause, and G. Taentzer, “Henshin : Advanced concepts and tools for in-place emf model transformations,” in *Model Driven Engineering Languages and Systems*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2010, pp. 121–135.

[13] F. Baader and B. Zarrieß, “Verification of golog programs over description logic actions,” in *Frontiers of Combining Systems*, P. Fontaine, C. Ringeissen, and R. A. Schmidt, Eds. Berlin, Heidelberg : Springer Berlin Heidelberg, 2013, pp. 181–196.

[14] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, Eds., *The Description Logic Handbook : Theory, Implementation, and Applications*. New York, NY, USA : Cambridge University Press, 2003.

[15] F. Baader, C. Lutz, M. Miličic, U. Sattler, and F. Wolter, “Integrating description logics and action formalisms : First results,” in *Proceedings of the 20th National Conference on Artificial Intelligence - Volume 2*, ser. AAAI’05. AAAI Press, 2005, pp. 572–577. [Online]. Available : http://dl.acm.org/citation.cfm?id=1619410.1619424

[16] F. Baader, M. Lippmann, and H. Liu, “Using causal relationships to deal with the ramification problem in action formalisms based on description logics,” in *Logic for Programming, Artificial Intelligence, and Reasoning*, C. G. Fermüller and A. Voronkov, Eds. Berlin, Heidelberg : Springer Berlin Heidelberg, 2010, pp. 82–96.

[17] N. Baklanova, J. H. Brenas, R. Echahed, C. Percebois, M. Strecker, and H. N. Tran, “Provably correct graph transformations with small-talc,” in *Proceedings of the 11th International Conference on ICT in Education, Research and Industrial Applications : Integration, Harmonization and Knowledge Transfer, Lviv, Ukraine, May 14-16, 2015.*, 2015, pp. 78–93. [Online]. Available : https://www.irit.fr/~Martin.Strecker/Publications/icteri2015.html

[18] N. Baklanova, J. H. Brenas, R. Echahed, A. Makhlouf, C. Percebois, M. Strecker, and H. N. Tran, “Coding, executing and verifying graph transformations with small-t$\mathcal{ALCQ}$e,” in *7th Int. Workshop on Graph Computation Models (GCM)*, 2016, URL : http://gcm2016.inf.uni-due.de/ [accessed : 2018-05-06].

[19] D. Balasubramanian, A. Narayanan, C. P. van Buskirk, and G. Karsai, “The graph rewriting and transformation language : Great,” *ECEASST*, vol. 1, 2006.

[20] M. Beaumont, “Efficient computation of weakest preconditions,” *RWTH Aachen University*, 2015.

[21] B. Becker, L. Lambers, J. Dyck, S. Birth, and H. Giese, “Iterative development of consistency-preserving rule-based refactorings,” in *Theory and Practice of Model Transformations : 4th International Conference, ICMT*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2011, pp. 123–137.

[22] M. Bienvenu and C. Bourgaux, "Inconsistency-Tolerant Querying of Description Logic Knowledge Bases," in *Tutorial Lectures of the 12th International Reasoning Web Summer School : Logical Foundation of Knowledge Graph Construction and Query Answering* , ser. Lecture Notes in Computer Science, 2016, vol. 9885, pp. 156–202. [Online]. Available : https://hal.inria.fr/hal-01633000

[23] A. Borgida, "On the relative expressiveness of description logics and predicate logics," *Artificial Intelligence*, vol. 82, no. 1, pp. 353 – 367, 1996. [Online]. Available : http://www.sciencedirect.com/science/article/pii/0004370296000045

[24] J. H. Brenas, "Hoare-like verification of graph transformation," Theses, Université Grenoble Alpes, Oct. 2016. [Online]. Available : https://tel.archives-ouvertes.fr/tel-01680448

[25] J. H. Brenas, R. Echahed, and M. Strecker, "A hoare-like calculus using the SROIQσ logic on transformations of graphs," in *Theoretical Computer Science - 8th IFIP International Conference, TCS*, 2014, pp. 164–178.

[26] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang, "Footprint analysis : A shape analysis that discovers preconditions," in *Static Analysis*, H. R. Nielson and G. Filé, Eds. Berlin, Heidelberg : Springer Berlin Heidelberg, 2007, pp. 402–418.

[27] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang, "Compositional shape analysis by means of bi-abduction," *J. ACM*, vol. 58, no. 6, pp. 26 :1–26 :66, Dec. 2011. [Online]. Available : http://doi.acm.org/10.1145/2049697.2049700

[28] D. Calvanese, M. Ortiz, and M. Simkus, "Evolving graph databases under description logic constraints," in *Proc. of the 26th Int. Workshop on Description Logics (DL 2013)*, ser. CEUR Workshop Proceedings, http ://ceur-ws.org/, vol. 1014, 2013, pp. 120–131.

[29] M. Chaabani, R. Echahed, and M. Strecker, "Logical foundations for reasoning about transformations of knowledge bases," in *Description Logics*, 2013.

[30] M. Chein and M.-L. Mugnier, "Conceptual graphs are also graphs," in *Graph-Based Representation and Reasoning*, N. Hernandez, R. Jäschke, and M. Croitoru, Eds. Cham : Springer International Publishing, 2014, pp. 1–18.

[31] N. Chomsky, *Syntactic structures.* The Hague : Mouton, 1957.

[32] R. Clarisó, J. Cabot, E. Guerra, and J. de Lara, "Backwards reasoning for model transformations," *J. Syst. Softw.*, vol. 116, no. C, pp. 113–132, Jun. 2016. [Online]. Available : https://doi.org/10.1016/j.jss.2015.08.017

[33] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Loewe, "Algebraic approaches to graph transformation, part i : Basic concepts and double pushout approach," Tech. Rep., 1996.

[34] D. Costal, C. Gómez, A. Queralt, and E. Teniente, "Drawing preconditions of operation contracts from conceptual schemas," in *Advanced Information Systems Engineering*, Z. Bellahsène and M. Léonard, Eds. Berlin, Heidelberg : Springer Berlin Heidelberg, 2008, pp. 266–280.

[35] B. Courcelle, "Handbook of graph grammars and computing by graph transformation," G. Rozenberg, Ed. River Edge, NJ, USA : World Scientific Publishing Co., Inc., 1997, ch. The Expression of Graph Properties and Graph Transformations in Monadic Second-order Logic, pp. 313–400. [Online]. Available : http://dl.acm.org/citation.cfm?id=278918.278932

[36] B. Courcelle, "The monadic second-order logic of graphs. i. recognizable sets of finite graphs," *Information and Computation*, vol. 85, no. 1, pp. 12 – 75, 1990. [Online]. Available : http://www.sciencedirect.com/science/article/pii/089054019090043H

[37] B. Courcelle, "Monadic second-order definable graph transductions : a survey," *Theoretical Computer Science*, vol. 126, no. 1, pp. 53–75, 1994.

[38] B. Courcelle and J. Engelfriet, *Graph Structure and Monadic Second-Order Logic : A Language-Theoretic Approach*, 1st ed. New York, NY, USA : Cambridge University Press, 2012.

[39] P. Cousot, "Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique des programmes," Habilitation à diriger des recherches, Institut National Polytechnique de Grenoble - INPG ; Université Joseph-Fourier - Grenoble I, Mar. 1978. [Online]. Available : https://tel.archives-ouvertes.fr/tel-00288657

[40] P. Cousot, "Interprétation abstraite," *Technique et science informatique*, vol. 19, no. 1-2, p. 3, 2000.

[41] P. Cousot, "Constructive design of a hierarchy of semantics of a transition system by abstract interpretation," *Theoretical Computer Science*, vol. 277, no. 1, pp. 47 – 103, 2002, static Analysis. [Online]. Available : http://www.sciencedirect.com/science/article/pii/S0304397500003133

[42] P. Cousot and R. Cousot, "Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM, 1977, pp. 238–252.

[43] P. Cousot and R. Cousot, "Systematic design of program analysis frameworks," in *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '79. New York, NY, USA : ACM, 1979, pp. 269–282. [Online]. Available : http://doi.acm.org/10.1145/567752.567778

[44] C. Csallner, N. Tillmann, and Y. Smaragdakis, "Dysy : Dynamic symbolic execution for invariant inference," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08. New York, NY, USA : ACM, 2008, pp. 281–290. [Online]. Available : http://doi.acm.org/10.1145/1368088.1368127

[45] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, Eds., *Structured Programming.* London, UK, UK : Academic Press Ltd., 1972.

[46] F. Deckwerth and G. Varró, "Attribute handling for generating preconditions from graph constraints," in *Graph Transformation*, H. Giese and B. König, Eds. Cham : Springer International Publishing, 2014, pp. 81–96.

[47] E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," *Commun. ACM*, vol. 18, no. 8, pp. 453–457, Aug. 1975. [Online]. Available : http://doi.acm.org/10.1145/360933.360975

[48] D. Distefano, P. W. O'Hearn, and H. Yang, "A local shape analysis based on separation logic," in *Tools and Algorithms for the Construction and Analysis of Systems*, H. Hermanns and J. Palsberg, Eds. Berlin, Heidelberg : Springer Berlin Heidelberg, 2006, pp. 287–302.

[49] H. Ehrig and A. Habel, *Graph Grammars with Application Conditions.* Berlin, Heidelberg : Springer Berlin Heidelberg, 1986, pp. 87–100. [Online]. Available : https://doi.org/10.1007/978-3-642-95486-3_7

[50] H. Ehrig, M. Pfender, and H. J. Schneider, "Graph-grammars : An algebraic approach," in *14th Annual Symposium on Switching and Automata Theory (swat 1973)*, Oct 1973, pp. 167–180.

[51] T. Eiter, T. Lukasiewicz, and L. Predoiu, "Generalized consistent query answering under existential rules," in *Proceedings of the 15th International Conference on the Principles of Knowledge Representation and Reasoning, KR 2016*, J. P. Delgrande and F. Wolter, Eds. AAAI Press, April 2016, pp. 359–368. [Online]. Available : http://www.aaai.org/ocs/index.php/KR/KR16/paper/view/12888

[52] J. H. Gallier, *Logic for Computer Science : Foundations of Automatic Theorem Proving.* New York, NY, USA : Harper & Row Publishers, Inc., 1985.

[53] L. Geiger and A. Zündorf, "Transforming graph based scenarios into graph transformation based junit tests," in *Applications of Graph Transformations with Industrial Relevance*, J. L. Pfaltz, M. Nagl, and B. Böhlen, Eds. Berlin, Heidelberg : Springer Berlin Heidelberg, 2004, pp. 61–74.

[54] R. Geiß, G. V. Batz, D. Grund, S. Hack, and A. Szalkowski, "Grgen : A fast spo-based graph rewriting tool," in *Graph Transformations*, A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro, and G. Rozenberg, Eds. Berlin, Heidelberg : Springer Berlin Heidelberg, 2006, pp. 383–397.

[55] A. H. Ghamarian, M. de Mol, A. Rensink, E. Zambon, and M. Zimakova, "Modelling and analysis using groove," *International Journal on Software Tools for Technology Transfer*, vol. 14, no. 1, pp. 15–40, Feb 2012. [Online]. Available : https://doi.org/10.1007/s10009-011-0186-x

[56] B. Glimm, I. Horrocks, B. Motik, G. Stoilos, and Z. Wang, "Hermit : An owl 2 reasoner," *Journal of Automated Reasoning*, vol. 53, no. 3, pp. 245–269, Oct 2014. [Online]. Available : https://doi.org/10.1007/s10817-014-9305-1

[57] M. Grohe, "Logic, graphs, and algorithms." *Logic and automata*, vol. 2, pp. 357–422, 2008.

[58] A. Habel and K.-H. Pennemann, "Correctness of high-level transformation systems relative to nested conditions," *Mathematical Structures in Computer Science*, vol. 19, no. 2, p. 245–296, 2009.

[59] A. Habel, R. Heckel, and G. Taentzer, "Graph grammars with negative application conditions," *Fundam. Inf.*, vol. 26, no. 3,4, pp. 287–313, Dec. 1996. [Online]. Available : http://dl.acm.org/citation.cfm?id=2379538.2379542

[60] A. Habel, J. Müller, and D. Plump, "Double-pushout graph transformation revisited," *Mathematical. Structures in Comp. Sci.*, vol. 11, no. 5, pp. 637–688, Oct. 2001. [Online]. Available : http://dx.doi.org/10.1017/S0960129501003425

[61] P. Hamill, *Unit test frameworks : tools for high-quality software development.* " O'Reilly Media, Inc.", 2004.

[62] R. Heckel and A. Wagner, "Ensuring consistency of conditional graph grammars - a constructive approach -," *Electronic Notes in Theoretical Computer Science*, vol. 2, pp. 118 – 126, 1995. [Online]. Available : http://www.sciencedirect.com/science/article/pii/S1571066105801884

[63] J. Henry, "Analyse statique par interprétation abstraite et procédures de décision," Theses, Université de Grenoble, Oct. 2014. [Online]. Available : https://tel.archives-ouvertes.fr/tel-01485202

[64] C. A. R. Hoare, “An axiomatic basis for computer programming,” *Commun. ACM*, vol. 12, no. 10, pp. 576–580, Oct. 1969. [Online]. Available : http://doi.acm.org/10.1145/363235.363259

[65] E. Jakumeit, S. Buchwald, and M. Kroll, “Grgen.net,” *International Journal on Software Tools for Technology Transfer*, vol. 12, no. 3, pp. 263–271, Jul 2010. [Online]. Available : https://doi.org/10.1007/s10009-010-0148-8

[66] P. Jalote and N. Agrawal, “Using defect analysis feedback for improving quality and productivity in iterative software development,” in *2005 International Conference on Information and Communication Technology*, Dec 2005.

[67] T. Kleymann, “Hoare logic and auxiliary variables,” *Formal Aspects of Computing*, vol. 11, no. 5, pp. 541–566, Dec 1999. [Online]. Available : https://doi.org/10.1007/s001650050057

[68] B. König and V. Kozioura, “Augur – a tool for the analysis of graph transformation systems,” *EATCS BULLETIN*, vol. 87, pp. 125–137, 2005.

[69] B. König and V. Kozioura, “Augur 2 — a new version of a tool for the analysis of graph transformation systems,” *Electronic Notes in Theoretical Computer Science*, vol. 211, pp. 201 – 210, 2008, proceedings of the Fifth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2006). [Online]. Available : http://www.sciencedirect.com/science/article/pii/S1571066108002570

[70] D. S. Kolovos, R. F. Paige, and F. A. C. Polack, *On the Evolution of OCL for Capturing Structural Constraints in Modelling Languages*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2009, pp. 204–218. [Online]. Available : https://doi.org/10.1007/978-3-642-11447-2_13

[71] B. König and V. Kozioura, “Towards the verification of attributed graph transformation systems,” in *Graph Transformations*, H. Ehrig, R. Heckel, G. Rozenberg, and G. Taentzer, Eds. Berlin, Heidelberg : Springer Berlin Heidelberg, 2008, pp. 305–320.

[72] L. Lambers and F. Orejas, “Tableau-based reasoning for graph properties,” in *Graph Transformation*, H. Giese and B. König, Eds. Cham : Springer International Publishing, 2014, pp. 17–32.

[73] K. Lano, “Constraint-driven development,” *Information and Software Technology*, vol. 50, no. 5, pp. 406 – 423, 2008. [Online]. Available : http://www.sciencedirect.com/science/article/pii/S0950584907000420

[74] K. Lano, S. Kolahdouz-Rahimi, and T. Clark, “Comparing verification techniques for model transformations,” in *Proceedings of the Workshop on Model-Driven Engineering, Verification*

and Validation, ser. MoDeVVa ’12. New York, NY, USA : ACM, 2012, pp. 23–28. [Online]. Available : http://doi.acm.org/10.1145/2427376.2427381

[75] K. Lano, S. Kolahdouz-Rahimi, I. Poernomo, J. Terrell, and S. Zschaler, “Correct-by-construction synthesis of model transformations using transformation patterns,” *Softw. Syst. Model.*, vol. 13, no. 2, pp. 873–907, May 2014. [Online]. Available : http://dx.doi.org/10.1007/s10270-012-0291-7

[76] K. Lano, *The B Language and Method : A Guide to Practical Formal Development*, 1st ed. Berlin, Heidelberg : Springer-Verlag, 1996.

[77] K. Lano and S. Kolahdouz-Rahimi, “Specification and verification of model transformations using uml-rsds,” in *Proceedings of the 8th International Conference on Integrated Formal Methods*, ser. IFM’10. Berlin, Heidelberg : Springer-Verlag, 2010, pp. 199–214. [Online]. Available : http://dl.acm.org/citation.cfm?id=1929463.1929478

[78] J. d. Lara and H. Vangheluwe, “Atom3 : A tool for multi-formalism and meta-modelling,” in *Fundamental Approaches to Software Engineering*, R.-D. Kutsche and H. Weber, Eds. Berlin, Heidelberg : Springer Berlin Heidelberg, 2002, pp. 174–188.

[79] K. R. M. Leino, “Efficient weakest preconditions,” *Information Processing Letters*, vol. 93, no. 6, pp. 281 – 288, 2005. [Online]. Available : http://www.sciencedirect.com/science/article/pii/S0020019004003357

[80] K. R. M. Leino and N. Polikarpova, “Verified calculations,” in *Verified Software : Theories, Tools, Experiments : 5th International Conference, VSTTE.* Berlin, Heidelberg : Springer Berlin Heidelberg, 2014, pp. 170–190.

[81] D. Lembo and M. Ruzzi, “Consistent query answering over description logic ontologies,” in *Web Reasoning and Rule Systems*, M. Marchiori, J. Z. Pan, and C. d. S. Marie, Eds. Berlin, Heidelberg : Springer Berlin Heidelberg, 2007, pp. 194–208.

[82] D. Lembo, M. Lenzerini, R. Rosati, M. Ruzzi, and D. F. Savo, “Inconsistency-tolerant semantics for description logics,” in *Web Reasoning and Rule Systems*, P. Hitzler and T. Lukasiewicz, Eds. Berlin, Heidelberg : Springer Berlin Heidelberg, 2010, pp. 103–117.

[83] H. J. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. B. Scherl, “Golog : A logic programming language for dynamic domains,” *The Journal of Logic Programming*, vol. 31, no. 1, pp. 59 – 83, 1997, reasoning about Action and Change. [Online]. Available : http://www.sciencedirect.com/science/article/pii/S0743106696001215

[84] H. Liu, C. Lutz, M. Miličić, and F. Wolter, “Reasoning about actions using description logics with general tboxes,” in *European Workshop on Logics in Artificial Intelligence*. Springer, 2006, pp. 266–279.

[85] A. Makhlouf, H. N. Tran, C. Percebois, and M. Strecker, “Combining dynamic and static analysis to help develop correct graph transformations,” in *Tests and Proofs*, B. K. Aichernig and C. A. Furia, Eds. Cham : Springer International Publishing, 2016, pp. 183–190.

[86] A. Makhlouf, C. Percebois, and H. N. Tran, “A Precondition Calculus for Correct-by-Construction Graph Transformations,” in *International Conference on Software Engineering Advances*. http ://www.iaria.org/ : IARIA, 2017, pp. 172–177.

[87] A. Makhlouf, C. Percebois, and H. N. Tran, “An auto-active approach to develop correct logic-based graph transformations,” *International Journal On Advances in Software*, vol. 11, pp. 147–158, Jun. 2018.

[88] P. Mespoulet, “Contribution au développement de l’environnement Small-t$\mathcal{ALC}$ pour la réécriture de graphes,” Institut universitaire de technologie de Toulouse (IUT A), Tech. Rep., 2016.

[89] B. Meyer, “Applying "design by contract",” *Computer*, vol. 25, no. 10, pp. 40–51, Oct. 1992. [Online]. Available : http://dx.doi.org/10.1109/2.161279

[90] B. Meyer, *Eiffel : The Language*. Upper Saddle River, NJ, USA : Prentice-Hall, Inc., 1992.

[91] B. Meyer, “Steps towards a theory and calculus of aliasing,” *Int. J. Software and Informatics*, vol. 5, no. 1-2, pp. 77–115, 2011.

[92] M. Molloy and B. Reed, “A critical point for random graphs with a given degree sequence,” vol. 6, pp. 161–180, 03 1995.

[93] Y. Moy, “Sufficient preconditions for modular assertion checking,” in *Verification, Model Checking, and Abstract Interpretation*, F. Logozzo, D. A. Peled, and L. D. Zuck, Eds. Berlin, Heidelberg : Springer Berlin Heidelberg, 2008, pp. 188–202.

[94] M. Münch, “Programmed graph rewriting system progres,” in *Applications of Graph Transformations with Industrial Relevance*, M. Nagl, A. Schürr, and M. Münch, Eds. Berlin, Heidelberg : Springer Berlin Heidelberg, 2000, pp. 441–448.

[95] M. Nagl, “Set theoretic approaches to graph grammars,” in *Graph-Grammars and Their Application to Computer Science*, H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld, Eds. Berlin, Heidelberg : Springer Berlin Heidelberg, 1987, pp. 41–54.

[96] F. Parisi-Presicce and Y. Zhao, "Policy analysis and verification by graph transformation tools," *Electronic Notes in Theoretical Computer Science*, vol. 127, no. 1, pp. 101 – 112, 2005, proceedings of the International Workshop on Graph-Based Tools (GraBaTs 2004). [Online]. Available : http://www.sciencedirect.com/science/article/pii/S1571066105001088

[97] R. Peñaloza, "Inconsistency-tolerant instance checking in tractable description logics," in *Rules and Reasoning*, S. Costantini, E. Franconi, W. Van Woensel, R. Kontchakov, F. Sadri, and D. Roman, Eds. Cham : Springer International Publishing, 2017, pp. 215–229.

[98] K.-H. Pennemann, "Development of correct graph transformation systems," in *Graph Transformations*, H. Ehrig, R. Heckel, G. Rozenberg, and G. Taentzer, Eds. Berlin, Heidelberg : Springer Berlin Heidelberg, 2008, pp. 508–510.

[99] D. Plump, "Handbook of graph grammars and computing by graph transformation," H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, Eds. River Edge, NJ, USA : World Scientific Publishing Co., Inc., 1999, ch. Term Graph Rewriting, pp. 3–61. [Online]. Available : http://dl.acm.org/citation.cfm?id=328523.328528

[100] D. Plump, "The graph programming language gp," in *Algebraic Informatics*, S. Bozapalidis and G. Rahonis, Eds. Berlin, Heidelberg : Springer Berlin Heidelberg, 2009, pp. 99–122.

[101] D. Plump, "The design of gp 2," in *WRS*, 2011.

[102] A. Podelski, A. Rybalchenko, and T. Wies, "Heap assumptions on demand," in *Computer Aided Verification*, A. Gupta and S. Malik, Eds. Berlin, Heidelberg : Springer Berlin Heidelberg, 2008, pp. 314–327.

[103] C. M. Poskitt, "Towards the verification of graph programs," in *QUALIFYING DISSERTATION*, 2010.

[104] C. M. Poskitt and D. Plump, "Hoare-style verification of graph programs," *Fundam. Inf.*, vol. 118, no. 1-2, pp. 135–175, Jan. 2012. [Online]. Available : http://dl.acm.org/citation.cfm?id=2385016.2385022

[105] C. M. Poskitt and D. Plump, "Verifying monadic second-order properties of graph programs," in *Graph Transformation*, H. Giese and B. König, Eds. Cham : Springer International Publishing, 2014, pp. 33–48.

[106] U. Ranger and E. Weinell, "The graph rewriting language and environment progres," in *Applications of Graph Transformations with Industrial Relevance*, A. Schürr, M. Nagl, and A. Zündorf, Eds. Berlin, Heidelberg : Springer Berlin Heidelberg, 2008, pp. 575–576.

[107] A. Rensink, “Representing first-order logic using graphs,” in *Graph Transformations*, H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg, Eds. Berlin, Heidelberg : Springer Berlin Heidelberg, 2004, pp. 319–335.

[108] A. Rensink, “The groove simulator : A tool for state space generation,” in *Applications of Graph Transformations with Industrial Relevance*, J. L. Pfaltz, M. Nagl, and B. Böhlen, Eds. Berlin, Heidelberg : Springer Berlin Heidelberg, 2004, pp. 479–485.

[109] J. C. Reynolds, “Separation logic : A logic for shared mutable data structures,” in *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, ser. LICS ’02. Washington, DC, USA : IEEE Computer Society, 2002, pp. 55–74. [Online]. Available : http://dl.acm.org/citation.cfm?id=645683.664578

[110] X. Rival, “Analyse statique par interprétation abstraite,” *TSI-Technique et Science Informatiques*, vol. 30, no. 4, p. 371, 2011.

[111] G. Rozenberg, “An introduction to the nlc way of rewriting graphs,” in *Graph-Grammars and Their Application to Computer Science*, H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld, Eds. Berlin, Heidelberg : Springer Berlin Heidelberg, 1987, pp. 55–66.

[112] O. Runge, C. Ermel, and G. Taentzer, “Agg 2.0 – new features for specifying and analyzing algebraic graph transformations,” in *Applications of Graph Transformations with Industrial Relevance*, A. Schürr, D. Varró, and G. Varró, Eds. Berlin, Heidelberg : Springer Berlin Heidelberg, 2012, pp. 81–88.

[113] M. Schmidt-Schauß and G. Smolka, “Attributive concept descriptions with complements,” *Artificial Intelligence*, vol. 48, no. 1, pp. 1 – 26, 1991. [Online]. Available : http://www.sciencedirect.com/science/article/pii/000437029190078X

[114] A. Schürr, “Progres for beginners,” *RWTH Aachen, D-52056 Aachen, Germany*, 1997.

[115] A. Schürr, A. J. Winter, and A. Zündorf, “The progres approach : Language and environment,” in *Handbook Of Graph Grammars And Computing By Graph Transformation : Volume 2 : Applications, Languages and Tools.* World Scientific, 1999, pp. 487–550.

[116] M. Strecker, “Modeling and verifying graph transformations in proof assistants,” *Electron. Notes Theor. Comput. Sci.*, vol. 203, no. 1, pp. 135–148, Mar. 2008. [Online]. Available : http://dx.doi.org/10.1016/j.entcs.2008.03.039

[117] G. Taentzer, “Agg : A tool environment for algebraic graph transformation,” in *Applications of Graph Transformations with Industrial Relevance*, M. Nagl, A. Schürr, and M. Münch, Eds. Berlin, Heidelberg : Springer Berlin Heidelberg, 2000, pp. 481–488.

[118] G. Taentzer, “Agg : A graph transformation environment for modeling and validation of software,” in *Applications of Graph Transformations with Industrial Relevance*, J. L. Pfaltz, M. Nagl, and B. Böhlen, Eds. Berlin, Heidelberg : Springer Berlin Heidelberg, 2004, pp. 446–453.

[119] M. Thielscher, “Flux : A logic programming method for reasoning agents,” *Theory Pract. Log. Program.*, vol. 5, no. 4-5, pp. 533–565, Jul. 2005. [Online]. Available : http://dx.doi.org/10.1017/S1471068405002358

[120] J. H. Weber, “Grape – a graph rewriting and persistence engine,” in *Graph Transformation*, J. de Lara and D. Plump, Eds. Cham : Springer International Publishing, 2017, pp. 209–220.

[121] R. Wilhelm and D. Maurer, *Les compilateurs : théorie-construction-génération.* Masson, 1994.

[122] J. Zhai, H. Wang, and J. Zhao, “Post-condition-directed invariant inference for loops over data structures,” in *2014 IEEE Eighth International Conference on Software Security and Reliability-Companion*, June 2014, pp. 204–212.