# Qualification of source code generators in the avionics domain : automated testing of model transformation chains

Elie Richa

ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY

TELECOM ParisTech

2015-ENST-0082

EDITE - ED 130

**Doctorat ParisTech**

**T H È S E**

**pour obtenir le grade de docteur délivré par**

**TELECOM ParisTech**

**Spécialité « Informatique et Réseaux »**

*présentée et soutenue publiquement par*

**Elie Richa**

le 15 décembre 2015

# Qualification of Source Code Generators in the Avionics Domain: Automated Testing of Model Transformation Chains

| | | | |
|---|---|---|---|
| Directeur de thèse : | **M. Laurent Pautet** | – | **TELECOM ParisTech** |
| Co-encadrant de thèse : | **M. Étienne Borde** | – | **TELECOM ParisTech** |
| Encadrant industriel : | **M. Jose Ruiz** | – | **AdaCore** |
| Co-encadrant industriel : | **M. Matteo Bordin** | – | **AdaCore** |

**Jury**

**M. Jordi Cabot** — Rapporteur
Professeur, Universitat Oberta de Catalunya, Barcelone, Espagne

**M. Jean-Pierre Talpin** — Rapporteur
Directeur de Recherches, Inria, Rennes, France

**M. Fabrice Kordon** — Examinateur
Professeur, Université Pierre et Marie Curie, Paris, France

**Mme Anne Etien** — Examinatrice
Maître de Conférences, Laboratoire d'Informatique Fondamentale de Lille, France

**M. Dominique Blouin** — Examinateur
Ingénieur de Recherche, Hasso-Plattner-Institut, Potsdam, Allemagne

**M. Laurent Pautet** — Directeur de thèse
Professeur, TELECOM ParisTech, Paris, France

**M. Étienne Borde** — Co-encadrant de thèse
Maître de Conférences, TELECOM ParisTech, Paris, France

**M. Jose Ruiz** — Encadrant industriel
Ingénieur de Recherche, AdaCore, Paris, France

T H È S E

**TELECOM ParisTech**
école de l'Institut Mines-Télécom - membre de ParisTech
46 rue Barrault 75013 Paris - (+33) 1 45 81 77 77 - www.telecom-paristech.fr

*"And a One, a Two, a One Two Three Four"*
*— a famous musician*

# Remerciements

Arrive enfin le moment ultime de ce morceau, les dernières notes avant l'accord final. Je me rappelle alors de tous ceux qui ont rendu cela possible. Ils sont nombreux et ma reconnaissance est grande, alors je leur dédie ce dernier couplet.

Etienne Borde et Laurent Pautet, merci pour votre soutien continu tant sur le plan professionnel que personnel. Vous m'avez fait confiance lorsque j'ai voulu m'aventurer dans des territoires inconnus, et vous êtes restés présents et patients dans les moments difficiles. Pour tout cela, je vous remercie profondément.

Matteo Bordin et Jose Ruiz, j'ai appris beaucoup en travaillant avec vous. Merci pour les discussions, les conseils et les encouragements. Avec vous j'ai appris à croire en mes idées et les défendre. Merci surtout pour les bons rapports humains que nous avons pu garder malgré les difficultés.

Je voudrais également remercier les rapporteurs Jordi Cabot et Jean-Pierre Talpin pour avoir accepté de relire ce manuscrit de taille considérable, ainsi que tous les autres membres du jury, Fabrice Kordon, Anne Etien et Dominique Blouin, pour l'attention qu'ils ont portée à mon travail et pour leurs remarques sincères et constructives.

Je tiens à remercier aussi tous ceux que j'ai côtoyés à AdaCore et à Télécom ParisTech. En particulier, Charly, Pierre-Marie, Raphaël, Chrissy, merci de m'avoir remonté le moral dans les nombreux moments de fatigue. Xavier, Cuauhtémoc, Robin, camarades de bureau et de peine (et de cookies), nos journées passées au labo resteront parmi mes meilleurs souvenirs de la thèse. Merci particulièrement à Claire et Xavier, pour m'avoir aidé maintes fois à remettre les choses en perspective.

Les zikos! Amélie, Alexandre, Anthony, Ben D., Ben Whitman, Brislee-Habibi, Cathy, Christophe, Doriane, Elisa, Ellen, Etienne, Finola, Jake, Lauren, Laurent, Noémie, Priscillia, Sinéad-Habibi, Tess, Thomas, Timothée, Valeria. Vous êtes devenus ma famille parisienne, avec les joies et les difficultés d'une vrai famille. Merci pour les Open-Mics, les concerts, les verres et toutes les nuits que j'ai passées à rédiger dans un coin du bar en votre compagnie bienveillante. Merci surtout de m'avoir fait redécouvrir mes entrailles musicales. Une pensée particulière à Noémie: nos concerts en super-héros sur scène me donnaient le courage de persévérer dans ma thèse. Une immense gratitude aussi pour Finola: merci pour ton soutien dans les moments sombres, et merci de m'avoir forcé à prendre des pauses; sans toi j'aurais certainement perdu la raison.

J'ai fait les derniers pas de ce parcours avec Morgane. Merci pour ton immense soutien dans ces dernières étapes, les plus difficiles. Tu es restée à mes côtés pour

de longues nuits de rédaction et tu m'as aidé moralement et concrètement à finir ce travail. J'espère pouvoir faire de même pour toi dans la réalisation de tes ambitions.

Enfin, je réserve le dernier remerciement à ma famille. Mes parents Najla et Pierre, du haut de mes modestes 28 ans, je reste pour la majeure partie ce que vous m'avez fait et je vous en suis profondément reconnaissant, qualités et défauts confondus. Je vous dédie cet accomplissement et le partage avec vous. Mes sœurs chéries Hala et Sana, vous êtes mes héroïnes. Vous avez su éclairer mes moments les plus sombres. Cet accomplissement vous appartient aussi, il est l'un des fruits de nos parcours individuels et collectifs. Ma famille, je vous aime et je souhaite à chacun de trouver le bonheur, le sien.

Et voilà que je rédige les derniers mots comme j'ai souvent rédigé cette thèse, dans un coin du bar, à mes côtés les amis, une bière, et une guitare.

# Abstract

In the avionics industry, Automatic Code Generators (ACG) are increasingly used to produce parts of embedded software automatically from model-based specifications. Since the generated code is part of critical software that can endanger human lives, safety standards require an extensive and thorough verification of ACGs called *qualification*. In this thesis in collaboration with AdaCore, we investigate the testing of ACGs which is a major aspect of qualification, with the objective of reducing the effort required by testing activities while maintaining a high error detection capability necessary for qualification.

In a first part of this work, we investigate the aspect of the exhaustiveness of test data which is typically ensured by unit testing techniques. Acknowledging the practical difficulties in applying unit testing in code generation chains, we propose a methodology aiming at **ensuring the high exhaustiveness of unit testing, but using only integration tests** which are easier to carry out. To this end, we assume that the ACG is specified in the ATL model transformation language, and propose first a **translation of ATL to the theory of Algebraic Graph Transformations (AGT)**. Relying on existing theoretical results in AGT, we define a **translation of postconditions expressing the exhaustiveness of unit testing into equivalent preconditions** on the input of the ACG. This translation is based on the construction of *weakest liberal preconditions* ($wlp$) that we extend and refine to our context of ATL transformations. The preconditions thus constructed allow the production of integration tests that ensure the same level of exhaustiveness as unit tests. We provide formal definitions and proofs of our extensions of the AGT theory, along with **simplification strategies to optimise the highly complex implementation** of our analysis. Finally, an experimental validation of our proposals is carried out with a focus on the assessment of the effectiveness of our simplification strategies. The contributions of this part of the thesis are available in the form of an Open Source tool called *ATLAnalyser*[1].

The second part of the research investigates the oracles of the integration tests of an ACG, *i.e.* the means of validating the code generated by a test execution. Given the large number of tests required by qualification, we seek **automatic test oracles** capable of verifying the validity of the code generated by a test execution. To this end, we propose a language for the **specification of simple textual constraints**

[1] *ATLAnalyser*, `https://github.com/elericha/atlanalyser`

defining the expected patterns of source code. We can then **validate the result of a test automatically** by checking the conformance of the generated code with the expected patterns. This approach is experimentally assessed on a real Simulink® to Ada/C code generator called QGen[2] developed at AdaCore.

**Keywords:** qualification, automatic code generation, analysis of model transformations, testing, ATL, OCL, algebraic graph transformation, nested graph conditions, weakest precondition, model-to-text test oracles

[2]QGen, `http://www.adacore.com/qgen`

# Résumé

Dans l'industrie de l'avionique, les générateurs automatiques de code (GAC) sont de plus en plus utilisés pour produire automatiquement des parties du logiciel embarqué à partir d'une spécification sous forme de modèle. Puisque le code généré fait partie d'un logiciel critique dont dépendent des vies humaines, les standards de sûreté exigent une vérification extensive et approfondie du GAC: la qualification. Dans cette thèse en collaboration avec AdaCore, nous abordons la problématique du test des GACs qui est un point clé de leur qualification, et nous cherchons des moyens de réduire l'effort exorbitant des activités de test tout en assurant le niveau élevé de détection d'erreurs nécessaire à la qualification.

Dans un premier volet de cette thèse, nous abordons le sujet de l'exhaustivité des données de test qui est habituellement assurée par le test unitaire du GAC. Vu la difficulté de mise en œuvre du test unitaire pour les GACs, nous proposons une méthodologie visant à **garantir le niveau d'exhaustivité du test unitaire en n'utilisant que des données de test d'intégration** qui sont plus faciles à produire et à maintenir. Pour cela nous supposons que le GAC est spécifié dans le langage de transformation de modèles ATL, et nous proposons une **traduction de ATL vers la théorie des Transformations Algébriques de Graphes (AGT)**. En se basant sur des résultats théoriques existants en AGT, nous définissons une **traduction de postconditions exprimant l'exhaustivité du test unitaire en des préconditions équivalentes** sur l'entrée du GAC. Cette traduction se base sur la construction du *weakest liberal precondition* ($wlp$) que nous augmentons et adaptons à nos besoins dans le contexte ATL. Les préconditions ainsi construites permettent à terme la production de tests d'intégration qui assurent le même niveau d'exhaustivité que le test unitaire. Nous fournissons les définitions et les preuves formelles de nos extensions de la théorie AGT, ainsi que **des stratégies de simplification visant à optimiser l'algorithme fortement complexe** de notre analyse. Enfin, une validation expérimentale de nos propositions est effectuée avec une attention particulière à l'évaluation de l'efficacité de nos stratégies de simplification. L'ensemble de nos contributions est disponible sous forme d'un outil Open Source appelé *ATLAnalyser*[3].

Le second volet du travail concerne les oracles des tests d'intégration, c'est à dire le moyen de valider le code généré par le GAC lors d'un test. Etant donné le grand nombre de tests nécessaires à la qualification, nous cherchons des **oracles automatiques de test** capables de vérifier la validité du code généré lors d'un test. Nous proposons ainsi un **langage de spécification de contraintes textuelles simples** ex-

[3] *ATLAnalyser*, `https://github.com/elericha/atlanalyser`

primant les patrons de code source attendus. Nous pouvons alors **valider automatiquement le résultat d'un test** en vérifiant la conformité du code généré avec les patrons attendus. Cette approche est déployée expérimentalement à AdaCore dans le cadre de la qualification de QGen[4], un générateur de code Ada/C à partir de modèles Simulink®.

**Mots-clés:** qualification, génération automatique de code, analyses de transformations de modèles, test, ATL, OCL, transformations algébriques de graphes, nested graph conditions, weakest precondition, oracles de tests model-to-text

[4]QGen, `http://www.adacore.com/qgen`

# Table of Contents

# List of Figures

# List of Tables

# List of Code Listings

# Résumé de la Thèse en Français

## Qualification des Générateurs de Code Source dans le Domaine de l'Avionique: le Test Automatisé des Chaines de Transformation de Modèles

### Contents

# 1 Contexte

## 1.1 Certification de Systèmes Critiques et Qualification d'Outils

Les logiciels embarqués dans les avions, les trains ou les voitures sont dit *critiques* en raison des conséquences catastrophiques que peut engendrer leur défaillance. Etant donné l'enjeu important, les logiciels critiques sont soumis à des standards de sureté stricts tel que le standard de certification DO−178C dans le domaine de l'avionique. Ce standard règlemente tous les aspects du développement du logiciel tel que la planification, l'organisation, le développement et la vérification. Par conséquent, le développement de logiciels critiques certifiés est extrêmement couteux, notamment en raison des vérifications poussées requises sur l'ensemble des artéfacts de développement.

Etant donné le coût élevé de la vérification, les industriels cherchent à éliminer certaines activités de vérification portant sur le code source en utilisant des Générateurs Automatiques de Code (GACs) et en montrant que ceux-ci produisent du code source présentant par construction les propriétés de correction désirées. Cela est possible lorsque le GAC est lui-même développé conformément au standard de qualification d'outils DO−330.

La qualification d'un GAC est aussi rigoureuse et couteuse que la certification d'un logiciel critique embarqué. Cependant le coût élevé de la qualification est compensé par l'utilisation de l'outil à plusieurs reprises durant le cycle de vie du système critique, notamment durant la phase de maintenance. Ainsi à chaque utilisation du GAC, les vérifications évitées grâce à la qualification représentent un retour sur l'investissement initial dans la qualification de l'outil. Pour cette raison la qualification d'outils est un sujet de recherche très actif dans les milieux académique et industriel des systèmes critiques aujourd'hui.

Les fournisseurs d'outils tel que le partenaire industriel de cette thèse, AdaCore, doivent adopter des méthodologies efficaces de développement et de vérification des GACs de sorte à pouvoir proposer aux constructeurs de systèmes critiques des outils qualifiés rentables. Cette thèse se concentre en particulier sur les aspects de test des GACs qui s'avère être très couteux en raison des critères élevés d'exhaustivité et de couverture requis par le standard de qualification.

### 1.2 Chaines de Transformations de Modèles et Génération de Code

Dans le contexte industriel de cette thèse, nous nous intéressons en particulier au générateur de code QGen[5] développé par AdaCore. QGen prend en entrée des modèles Simulink®[6] et génère du code C ou Ada, selon le choix de l'utilisateur. Comme beaucoup d'outils manipulant des modèles, QGen est implémenté sous forme d'une *chaine de transformations de modèles*. Ainsi le travail de cette thèse porte sur les problématiques de test des chaines de transformation de modèles. La section suivante rappelle les principaux résultats existant autour de ce sujet, notamment la méthodologie générale typiquement adoptée dans l'état de l'art pour le test des transformations de modèles.

## 2 Etat de l'Art

De nombreux travaux existants traitent de la vérification de transformations de modèles, notamment par le test. L'analyse de ces travaux nous a mené à extraire un même processus sous-jacent commun à la majorité des approches de test existantes.

La première partie du processus est la sélection d'un *critère d'adéquation*, illustré dans la Figure 1. Ce critère peut prendre en entrée une variété de sources d'information telles que le métamodèle d'entrée de la transformation, sa spécification, ou même son implémentation dans certains cas. Les critères d'adéquation proposés par les différentes approches de test existantes diffèrent dans le choix des sources d'information et dans la manière dont les informations sont combinées [Fleurey *et al.*, 2004; Brottier *et al.*, 2006; Fleurey *et al.*, 2007; Mottu *et al.*, 2012; Sen *et al.*, 2009; Guerra, 2012; Guerra and Soeken, 2015; González and Cabot, 2012; González and Cabot, 2014]. Bien que des terminologies différentes sont employées, le rôle d'un critère d'adéquation est toujours de générer un ensemble de contraintes que nous désignons par *exigences de test* (*test requirements*). Les exigences de test visent à partitionner le domaine d'entrée de la transformation: chaque exigence de test définit un sous-ensemble du domaine d'entrée. Afin d'assurer l'exhaustivité au sens du critère d'adéquation choisi, il faut donc s'assurer que chaque exigence de test (càd chaque contrainte) soit satisfaite au moins une fois durant la campagne de test.

Dans la deuxième partie du processus, l'ensemble des exigences de test obtenu peut être utilisé de deux manière. Si l'on dispose déjà d'un ensemble de modèles de test existants, nous pouvons évaluer le nombre d'exigences de test satisfaites par

[5]QGen, `http://www.adacore.com/qgen`
[6]Simulink, `http://www.mathworks.com/products/simulink`

cet ensemble et déterminer ainsi un *score d'adéquation*. Si l'on ne dispose pas déjà d'un ensemble de modèles de test, nous pouvons donner l'ensemble d'exigences de test en entrée à un processus de *génération automatique de modèles* afin de produire un ensemble de modèles qui satisfait toutes les exigences de test par construction. Plusieurs approches proposent déjà des techniques de génération de modèles [Brottier *et al.*, 2006], dont les plus récentes et les plus prometteuses repose sur des solveurs SAT ou SMT de contraintes [Sen *et al.*, 2008; Sen *et al.*, 2009; Sen *et al.*, 2012; Mottu *et al.*, 2012; Guerra, 2012; Guerra and Soeken, 2015; Aranega *et al.*, 2014].

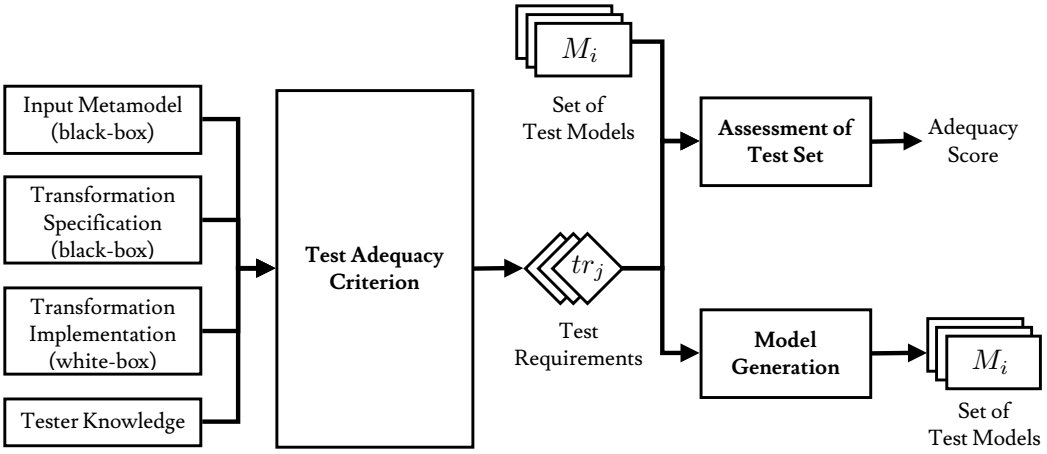



Figure 1: Processus générique du critère d'adéquation et de la génération de tests

Enfin, la dernière partie du processus de test concerne la validation des résultats de test par un *oracle de test*. Comme illustré par la Figure 2, le rôle d'un oracle de test est de décider si le résultat d'un test est un succès (*PASS*) ou un échec (*FAIL*). Divers types d'oracles sont proposés dans l'état de l'art [Mottu *et al.*, 2008] et diffèrent par le type d'information utilisé pour valider la sortie d'un test. Les oracles simples consistent par exemple à comparer la sortie de chaque test avec un résultat attendu validé manuellement au préalable. Cependant les oracles les plus intéressants sont ceux qui reposent sur une spécification sous forme de contraintes exprimant les caractéristiques que doit exhiber la sortie du test en relation avec les caractéristiques de ses entrées (*i.e.* un contrat) [Cariou *et al.*, 2004; Guerra *et al.*, 2010; Cariou *et al.*, 2009; Guerra *et al.*, 2013; Guerra and Soeken, 2015].

## 3 Problématique

Après avoir présenté les aspects principaux du test des GACs et un aperçu des techniques de test existantes, nous identifions à présent deux problèmes précis qui sont traités dans le cadre de ce travail.

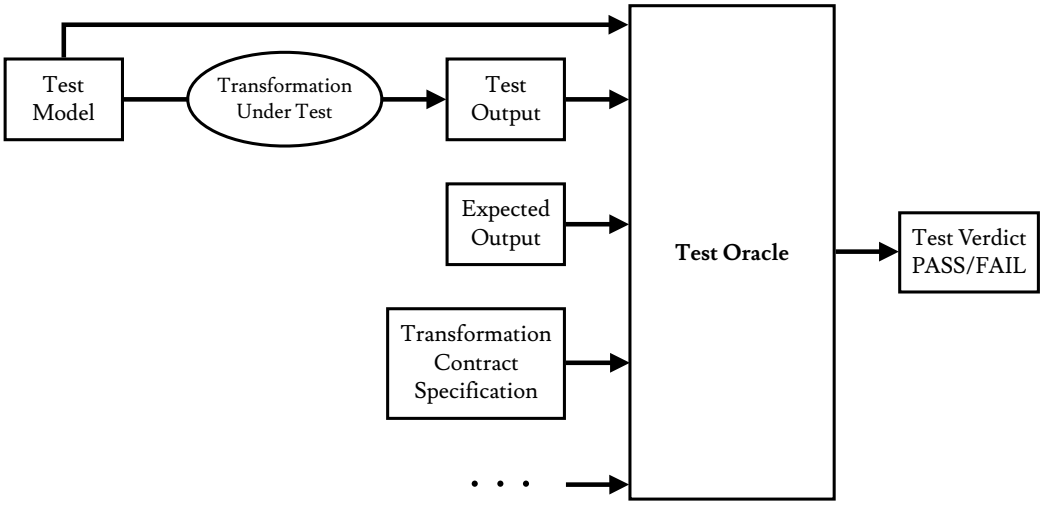



Figure 2: Processus générique des oracles de test

## 3.1 Test Unitaire et Test d'Intégration des Chaines de Transformations de Modèles

### 3.1.1 Test Unitaire et Test d'Intégration

La qualification requiert deux genres de vérification: le *test unitaire* et le *test d'intégration*. Le test unitaire considère chaque composant de l'outil indépendamment du reste et vise à vérifier la bonne implémentation de ses fonctionnalités. En revanche le test d'intégration considère l'outil en entier et cherche à vérifier le bon fonctionnement de l'outil de bout en bout. Pour un GAC ayant l'architecture illustrée dans la figure 3, un composant de l'outil est une étape de transformation $T_i$. Le test unitaire consiste alors à produire des modèles de test $M_{i,j}$ dans le langage intermédiaire $MM_i$, à exécuter $T_i$ sur ces modèles de test, et à valider les modèles de sortie $M_{i+1,j}$ par un oracle adéquat. Par ailleurs, le test d'intégration s'intéresse à la chaine complète et consiste à produire des modèles de test $M_{0,j}$ dans le langage d'entrée $MM_0$ de la chaine, à exécuter la chaine complète, et à valider le résultat final $M_{N,j}$ par un oracle adéquat.

Bien que les deux sortes de tests sont requis par les standards de qualification, nous observons en réalité que le test d'intégration est largement préféré par rapport au test unitaire. Pour des projets tels que le générateur de code QGen développé à AdaCore ou le compilateur GCC qui présente une architecture en chaine similaire, nous observons que la vaste majorité des tests sont des tests d'intégration alors que les tests unitaires sont presque inexistants. Cette observation est aussi confirmée par des travaux existants [Stuermer *et al.*, 2007] qui soulignent la difficulté de mise en œuvre du test unitaire pour les GACs.

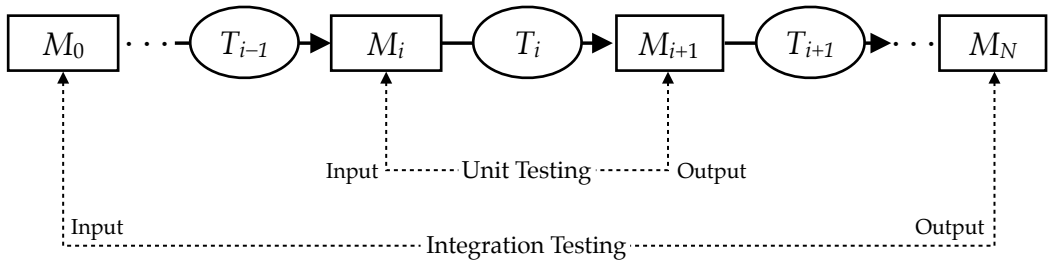



Figure 3: Test unitaire et test d'intégration d'une chaine de transformation de modèles

Ces observations s'expliquent par le fait que le test unitaire nécessite des données de test développées dans les langages intermédiaires de la chaine. Ces langages internes sont typiquement très complexes et ne disposent pas d'outils d'édition et de visualisation dédiés (étant internes à l'outil), ce qui rend la production manuelle des données de test difficile. Bien qu'une production automatique soit possible grâce à plusieurs travaux existants [Sen *et al.*, 2008; Sen *et al.*, 2009; Sen *et al.*, 2012; Mottu *et al.*, 2012; Guerra, 2012; Guerra and Soeken, 2015; Aranega *et al.*, 2014], elle résulte souvent en un très grand nombre de tests dont la maintenance reste problématique. En effet les langages internes de l'outil peuvent évoluer au cours du cycle de développement et l'identification des données de test affectées et leur mise à jour n'est pas triviale.

En revanche, les tests d'intégration sont avantageux car ils utilisent des données de test exprimées dans le langage d'entrée de la chaine. Ce langage, qui est celui employé par les utilisateurs de l'outil, est ainsi plus simple et propose un niveau d'abstraction élevé. Il s'agit souvent d'un langage stable qui évolue peu et qui dispose d'un bon éditeur. L'ensemble de ces aspects facilitent donc la production et la maintenance des tests d'intégration. Il serait donc souhaitable de n'utiliser que des tests d'intégration pour la vérification des GACs.

Cependant le test unitaire joue un rôle très important dans la qualification de l'outil et ne peut être simplement évité. En effet, il permet de cibler des fonctionnalités précises de chaque composant et d'atteindre ainsi l'exhaustivité élevée requise par les standards de qualification. Se pose alors la question suivante: comment peut on assurer le même niveau d'exhaustivité que le test unitaire en n'utilisant que des tests d'intégration?

### 3.1.2 Assurer l'Exhaustivité du Test Unitaire par le Test d'Intégration

En s'inspirant des travaux de [Bauer *et al.*, 2011], nous proposons d'assurer l'exhaustivité du test unitaire en n'utilisant que des tests d'intégrations. Nous il-

lustrons cela pour une chaine de 3 transformations $T_0$, $T_1$ et $T_2$. Nous considérons tout d'abord chaque transformations séparément, et nous appliquons les méthodes traditionnelles de sélection de critère d'adéquation et production d'un ensemble d'exigences de test unitaire. Nous désignons par $tr_{i,j}$ l'exigence de test $j$ de la transformation $T_i$. Nous supposons que cette étape produit 3 exigences de test pour $T_0$, 2 exigences de test pour $T_1$ et 2 exigences de test pour $T_2$ qui sont regroupées dans le Tableau 1. Ensuite, en considérant toujours chaque transformation séparément, nous appliquons les méthodes traditionnelles de sélection d'oracles de tests unitaires, et nous supposons que nous sommes capables de spécifier un oracle automatique $to_i$ pour chaque transformation $T_i$. Chaque oracle $to_i$ est une contrainte sur l'entrée et la sortie de $T_i$ permettant de valider la bonne exécution de la transformation.

| | **Test Requirements and Oracles** | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Integration Test Models** | $T_0$ | | | | $T_1$ | | | $T_2$ | | |
| | $tr_{0,0}$ | $tr_{0,1}$ | $tr_{0,2}$ | $to_0$ | $tr_{1,0}$ | $tr_{1,1}$ | $to_1$ | $tr_{2,0}$ | $tr_{2,1}$ | $to_2$ |
| $M_{0,0}$ | T | F | F | T | F | T | T | F | F | T |
| $M_{0,1}$ | F | T | F | T | T | F | T | T | F | T |
| $M_{0,2}$ | F | F | T | T | F | T | T | F | T | T |

Table 1: Tests d'intégration assurant l'exhaustivité du test unitaire

A présent, au lieu de procéder avec des techniques classiques de génération de modèles de test unitaire, nous considérons plutôt un ensemble de modèles de test d'intégration que nous supposons exister déjà. Il s'agit des modèles $M_{0,0}$, $M_{0,1}$ et $M_{0,2}$ du Tableau 1. Nous exécutons alors la chaine complète sur chaque modèle de test d'intégration $M_{0,k}$ et nous procédons comme suit:

1. Avant l'exécution de chaque transformation $T_i$ nous évaluons toutes ses exigences de test unitaire $tr_{i,j}$ sur les modèles intermédiaires $M_{i,k}$ et nous sauvegardons les résultats d'évaluation dans le tables 1 où "T" indique la satisfaction de l'exigence de test et "F" indique le contraire.

2. Après l'exécution de chaque transformation $T_i$ nous évaluons son oracle de test $to_i$ afin de valider l'exécution de chaque étape.

Par ce procédé nous pouvons constater dans le tableau 1 que toutes les exigences de test unitaires ont été satisfaites en n'utilisant que des tests d'intégration.

Cependant que se passe-t-il lorsque nous détectons une exigence de test unitaire qui n'est jamais satisfaite durant la campagne de test? Ce scénario est illustré dans le Tableau 2. Il faudrait dans ce cas produire un nouveau test d'intégration qui mènerait à la satisfaction de l'exigence de test unitaire. L'approche de [Bauer *et al.*,

2011] à la base de notre méthode de test ne propose pas de moyen de production de ce nouveau test d'intégration. Nous arrivons ainsi au problème central traité dans cette thèse: *Comment produire a nouveau test d'intégration pour couvrir une exigence de test unitaire non-satisfaite?*.

| | **Test Requirements and Oracles** | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Integration Test Models** | $T_0$ | | | | $T_1$ | | | $T_2$ | | |
| | $tr_{0,0}$ | $tr_{0,1}$ | $tr_{0,2}$ | $to_0$ | $tr_{1,0}$ | $tr_{1,1}$ | $to_1$ | $tr_{2,0}$ | $tr_{2,1}$ | $to_2$ |
| $M_{0,0}$ | T | F | F | T | F | T | T | F | F | T |
| $M_{0,4}$ | F | T | F | T | F | F | T | T | F | T |
| $M_{0,2}$ | F | F | T | T | F | T | T | F | T | T |

Table 2: Identification d'exigences de test non-satisfaites

### 3.1.3 Problème: Production de Modèles de Test d'Intégration pour Couvrir des Exigences de Test Unitaire

Le problème identifié est illustré par la Figure 4: Etant donné une exigence de test $tr_{i,j}$ non-satisfaite, comment produire un modèle de test d'intégration $M_0$ qui mènerait à la satisfaction de l'exigence de test? Nous explorons dans ce qui suit des pistes de solutions existantes de l'état de l'art et nous expliquons pourquoi elles ne sont pas satisfaisantes.

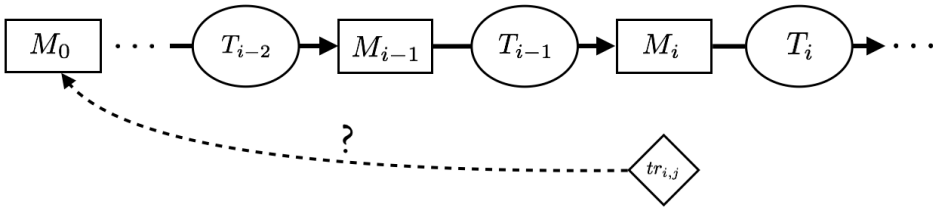



Figure 4: Satisfaction des exigences de test unitaire

Une première piste illustrée par la flèche 1 de la Figure 5 consiste à considérer le problème comme un *Problème de Satisfaction de Contrainte* (CSP). Nous pouvons alors appliquer un solveur de contraintes (SAT ou SMT) pour obtenir un modèle $M_i$ satisfaisant. Cependant il s'agit d'un modèle dans un langage intermédiaire de la chaine or nous cherchons un modèle dans le langage d'entrée de la chaine. D'autres approches de l'état de l'art permettent d'inclure dans le CSP la définition des transformations précédentes $T_{i-1} \circ \cdots \circ T_1 \circ T_0$ à travers la notion du *transformation model* [Büttner *et al.*, 2012b] ce qui permet en théorie (flèche 3) d'obtenir le modèle $M_0$ recherché. Cependant le CSP devient dans ce trop complexe et les solveurs de contraintes rencontrent alors des problèmes de passage à l'échelle.

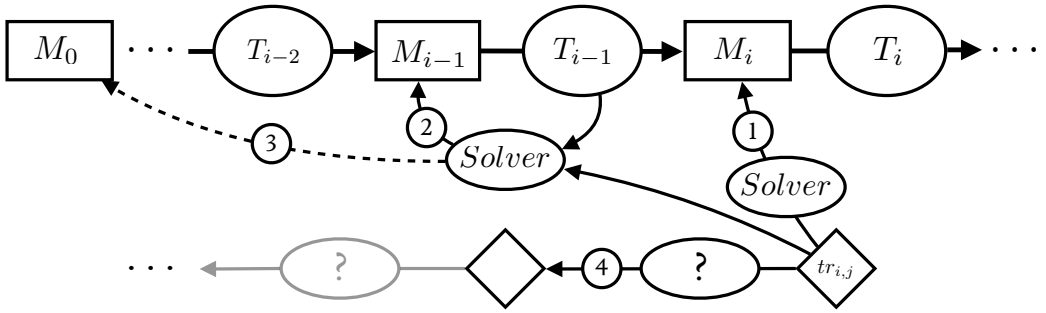



Figure 5: Solutions existantes de génération de modèles

Nous recherchons plutôt une solution itérative illustrée par la flèche 4. Cette solution nécessite de développer une analyse qui prenne en entrée une *contrainte* et qui produise en sortie une *contrainte* également, de manière à permettre un traitement itératif, étape par étape, du problème. Par une telle analyse nous pourrions traduire l'exigence de test non-satisfaite étape par étape en amont le long de la chaine, jusqu'à une contrainte équivalente au niveau du langage d'entrée de la chaine. Là, les approaches classiques de génération de modèles peuvent fournir le modèle $M_0$ recherché. Ainsi le premier problème traité par cette thèse est formulé de la manière suivante:

*Traduction d'une exigence de test non-satisfaite, en arrière le long d'une chaine de transformations de modèle, en une contrainte équivalente sur le langage d'entrée de la chaine.*

Après cette première partie où nous avons cherché à couvrir les exigences de test unitaire à l'aide de tests d'intégration, nous allons nous intéresser en seconde partie aux oracles de ces tests d'intégration.

## 3.2 Oracles de Test des Transformations Model-to-Code

Dans la partie précédente nous avons proposé une approche de test basée sur l'exécution de tests d'intégration exclusivement. A présent nous nous intéressons aux oracles de ces tests, c'est à dire le moyen de décider si le résultat d'un test est un succès ou un échec. Un test d'intégration considère l'outil testé comme une boite noire, c'est à dire que l'entrée du test est un modèle de test et la sortie est le Code Source (CS) généré automatiquement. L'oracle d'un tel test devra inspecter l'entrée et la sortie du test, et valider si le code source généré implémente correctement la sémantique du modèle d'entrée.

Figure about integration test oracle

Figure 6: Oracle d'un test d'intégration

Lorsque l'on cherche à valider le CS il faut s'intéresser à deux aspects: la *syntaxe* du CS, c'est à dire sa structure, et la *sémantique* du CS, c'est à dire son comportement à l'exécution. Ayant constaté que la validation sémantique du CS est déjà traitée de manière satisfaisante par l'état de l'art [Sturmer and Conrad, 2003; Stuermer *et al.*, 2007], nous choisissons de nous concentrer sur la validation syntaxique du CS. Nous avons là aussi deux moyens de considérer la structure du CS:

1. Considérer le CS sous la forme de son arbre de *syntaxe abstraite*.

2. Considérer le CS sous la forme de sa *syntaxe concrète* textuelle.

Nous choisissons de considérer le CS sous la forme de sa syntaxe concrète pour la raison suivante. Dans le contexte de qualification les oracles de test doivent nécessairement se baser sur une spécification précises des *exigences* de l'outil. Ces exigences font partie des documents de qualification présentés aux autorités de certifications et validés par celles ci. Il est donc judicieux d'adopter une notation simple et facile à comprendre par un large éventail de profils de personnes. Dans ce contexte la forme textuelle du CS est plus connue et plus facile à comprendre que la forme en arbre de syntaxe abstraite qui est habituellement connue par les spécialistes de compilation. C'est donc pour cette raison que nous choisissons de construire nos oracles de tests de manière à valider le CS sous sa forme textuelle.

Une conséquence de ce choix est que le générateur de code testé devient une transformation *Model-to-Text*: il prend en entrée un modèle et produit en sortie des artefacts textuels dont nous cherchons à valider le contenu. L'état de l'art contient peu de travaux concernant la spécification d'oracles de test de transformations Model-to-Text [Wimmer and Burgueño, 2013], ce qui nous a poussé à traiter ce problème dans le contexte particulier de la qualification de générateurs de codes à AdaCore. Ainsi le second problème traité dans cette thèse est formulé de la manière suivante:

*Proposer une approche de spécification d'oracles de tests de générateurs de code qui se focalise sur la syntaxe textuelle du code généré.*

Nous avons ainsi défini les deux problèmes traités dans cette thèse. Le premier problème concerne la traduction d'exigences de test unitaire non-satisfaites en contraintes équivalentes sur le langage d'entrée d'une chaine de transformations, le but étant d'assurer une exhaustivité équivalente au test unitaire en n'utilisant que des tests d'intégration. Le second problème s'intéresse aux oracles de ces tests d'intégration et consiste à proposer une approche de spécification d'oracles qui valident la syntaxe textuelle du code généré lors d'un test. Nous présentons dans la suite les solutions que nous proposons pour résoudre ces problèmes.

# 4 Contributions

## 4.1 Traduction Arrière des Exigences de Test Unitaire

### 4.1.1 Principe Général

Dans la première partie de la thèse nous avons démontré qu'il est possible de couvrir les exigences de test unitaire à l'aide de tests d'intégration, et nous nous sommes concentré sur le problème de la production de nouveaux tests d'intégration ciblant la couverture d'exigences de test non-satisfaites. Nous avons ensuite conclu qu'il était nécessaire de traduire une exigence de test unitaire non-satisfaite d'une étape intermédiaire de la chaine en une contrainte équivalente sur le langage d'entrée de la chaine, ce qui permettrait la production d'un nouveau test d'intégration. Pour réaliser cette traduction arrière, nous proposons l'approche illustrée par la Figure 7. Etant donné une exigence de test unitaire non-satisfaite $tr_{i,j}$ de l'étape $T_i$, nous proposons de considérer $tr_{i,j}$ comme une *postcondition* de l'étape précédente $T_{i-1}$. Nous définissons alors une construction que nous appelons *Post*2*Pre* qui transforme la postcondition en une précondition équivalente garantissant la satisfaction de la postcondition. Nous appelons cette précondition $etr_{i,j,i-1}$ *l'exigence de test équivalente* de $tr_{i,j}$ à l'étape $T_{i-1}$.

La précondition étant elle aussi une contrainte, nous pouvons à nouveau la considérer comme une postcondition de l'étape précédente, et répéter ainsi le même raisonnement pour toutes les étapes précédentes jusqu'à obtenir une exigence de test $etr_{i,j,0}$ sur le langage d'entrée de la chaine. Un nouveau modèle de test $M_0$ satisfaisant $etr_{i,j,0}$ peut alors être produit automatiquement à l'aide de techniques existantes de génération de tests. Etant donné que chaque application de *Post*2*Pre* produit une précondition garantissant la satisfaction de la postcondition, nous avons la certitude que le nouveau modèle de test $M_0$ aboutit à la satisfaction de l'exigence de test de départ $tr_{i,j}$ lors de l'exécution du test, ce qui résout ainsi le problème initial.

### 4.1.2 Réalisation à l'aide de la Théorie des Transformations Algébriques de Graphes

Dans notre recherche des moyens de définir la construction *Post*2*Pre* nous nous sommes intéressé aux notions de postcondition et de précondition introduites dans [Hoare, 1969] comme un outil pour la preuve de correction de programmes. Ces notions définies initialement pour des programmes impératifs classiques similaires à C et Java ont par la suite été réutilisés dans [Habel *et al.*, 2006a] et [Poskitt, 2013]

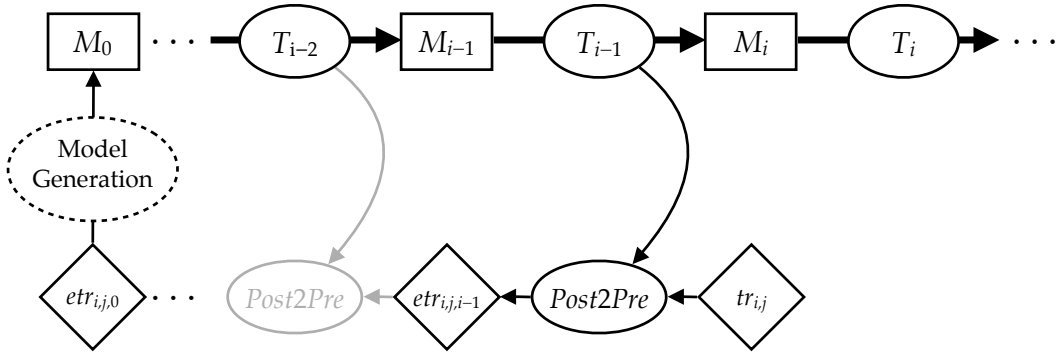



Figure 7: Traduction arrière étape par étape des exigences de test

pour la preuve de correction de programmes transformant des graphes dans le contexte de la théorie des *Transformations Algébriques de Graphes* (AGT). En particulier, des travaux existants ont proposé la construction de la *plus faible précondition* ou *weakest precondition (wp)* comme un moyen de traduire une postcondition $d$ requise en sortie d'une transformation de graphes $P$, en une précondition $wp(P, d)$ à l'entrée de $P$ qui garantit la satisfaction de la postcondition. Comme les transformations de modèles s'apparentent beaucoup à des transformations de graphes, et puisque la construction du $wp$ fournit les propriétés que nous cherchons, nous avons trouvé judicieux de baser notre solution sur cette théorie.

Cependant comme la théorie AGT n'est pas adaptée au développement de générateurs de code en contexte industriel, nous avons opté pour le langage ATL [Jouault and Kurtev, 2006] pour la spécification des transformations de modèles de la chaine, et OCL [OMG, 2014] pour la spécification des exigences de test à satisfaire. Dans certains contextes ATL est utilisé pour implémenter des outils de génération de code pour des systèmes embarqués critiques [Cadoret *et al.*, 2012] ce qui justifie davantage notre choix.

Comme illustré par la Figure 8, notre solution consiste alors à transposer le problème dans la théorie AGT. Une première étape $ATL2AGT$ traduit la transformation de modèle ATL en un transformation de graphes AGT. Ensuite, l'étape $OCL2NGC$ traduit l'exigence de test OCL à satisfaire en une contrainte équivalente sous forme de *Nested Graph Constraints* (NGC) dans la théorie AGT. Enfin, l'étape *Post2Pre* considère la contrainte NGC comme une postcondition de la transformation de graphe, et la traduit en une précondition équivalente. *Post2Pre* sera basée sur les principes de la construction $wp$ enrichie par des constructions supplémentaires qui seront détaillée par la suite.

Dans la suite nous donnons un aperçu des différentes parties de notre solution en identifiant les contributions apportées dans chacune.

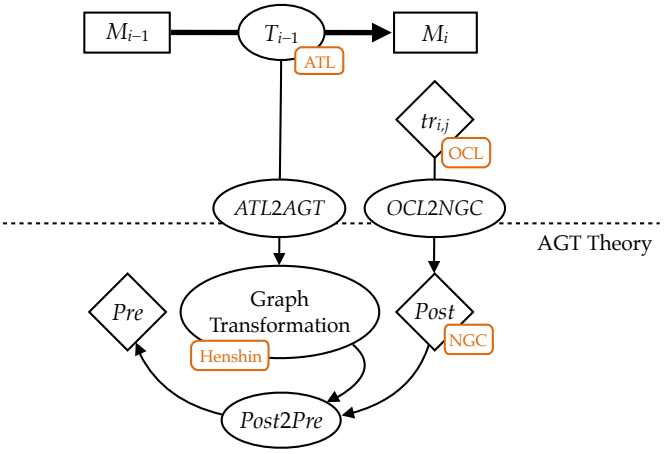



Figure 8: Traduction en AGT et traduction arrière des exigences de test avec *Post2Pre*

### 4.1.3 Traductions ATL et OCL vers AGT – *ATL2AGT* et *OCL2NGC*

Nous proposons tout d'abord une traduction du sous ensemble déclaratif d'ATL vers le formalisme théorique AGT. La difficulté principale de ce travail réside dans le support des *mécanismes de résolution* (*default and non-default resolve mechanisms*) d'ATL qui n'ont pas d'équivalent dans la sémantique AGT. Les mécanismes de résolutions permettent à chaque règle de transformation ATL de référencer des objets créés par d'autres règles lors de la création de références entre les objets constituant le modèle de sortie. L'un des cas d'utilisation les plus intéressants est celui où une règle $R_1$ référence des objets créés par une règle $R_2$, et *vice versa* $R_2$ référence des objets créés par $R_2$. En revanche la sémantique AGT ne fournit pas ce genre de mécanismes et le seul moyen pour une règle AGT $\rho_1$ de référencer des objets créés par une règle $\rho_2$ est en ordonnançant $\rho_2$ avant $\rho_1$ lors de l'exécution de la transformation. Cela complexifie le support du scénario du référencement mutuel de deux règles.

**Schéma général de la traduction** Nous résolvons ce problème en traduisant une transformation ATL en une transformation AGT organisée en deux phases successives:

1. *La phase d'instanciation* se compose de *règles d'instanciations* AGT qui créent les objets du modèle de sortie sans les connecter. Ces règles créent également des *objets de trace* qui lient les éléments d'entrée de chaque règle aux éléments de sortie. Ainsi chaque règle ATL se traduit en une règle d'instanciation AGT.
2. *La phase de résolution* se compose de *règles de résolution* AGT qui créent des liens entre les objets du modèle de sortie et qui initialisent leurs attributs.

Ces règles se basent sur les objets de trace créés par la première phase afin d'émuler les mécanismes de résolution de ATL. Ainsi, chaque affectation d'attribut ou de référence (une *binding*) dans une règle ATL se traduit en une (ou plusieurs) règle(s) de résolution.

En construisant la transformation résultante de cette manière, notre traduction transpose une transformation ATL en une transformation AGT équivalente à l'exécution qui prend en charge les mécanismes de résolution conformément à la sémantique ATL. Cette première contribution de notre travail constitue une formalisation de la sémantique ATL à l'aide de la théorie AGT. Cette formalisation nous permettra dans un second temps de mettre en œuvre l'analyse *Post*2*Pre* portant sur les contraintes de transformation, mais elle pourrait être exploitée au delà du contexte de cette thèse pour l'analyse formelle de transformations ATL à l'aide du formalisme AGT.

```
1  rule R1 {
2  from s  : IN!A
3   (s.refB->exists(b | b.name = 'Hodor'))
4  to t1 : OUT!D
5     (name <- s.name + '1',
6      refD <- t2,
7      refE <- s.refB),
8   t2 : OUT!D
9     (name <- s.name + '2') }
10
11
12
13 rule R2 {
14 from s : IN!B
15 to t : OUT!E
16    (refD <- thisModule.resolveTemp(s.refA, 't2') ) }
```

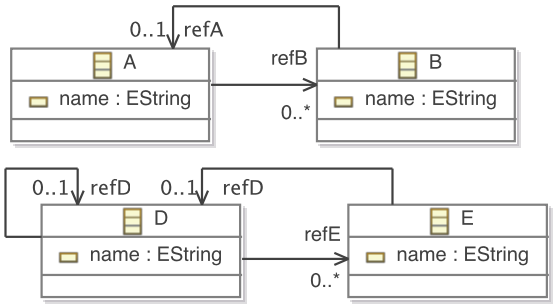



Figure 9: Exemple de transformation ATL

Par exemple, la transformation ATL de la Figure 9 est traduite par notre approche en la transformation AGT suivante:

$$T_{AGT} = R1_{Inst}\downarrow;\ R2_{Inst}\downarrow;\ R1^{t1,name}_{Res}\downarrow;\ R1^{t1,refD}_{Res}\downarrow;\ R1^{t1,refE}_{Res}\downarrow;\ R1^{t2,name}_{Res}\downarrow;\ R2^{t,refD}_{Res}\downarrow$$

où les règles $R1_{Inst}$ et $R1^{t1,refE}_{Res}$ sont représentées graphiquement dans la Figure 10.

**Traitement des expressions OCL et préservation de l'ordre** Une seconde contribution de notre travail concerne la traduction des expressions OCL utilisées dans les règles de transformation ATL en des *Nested Graph Conditions* (NGC) équivalentes jouant le rôle de conditions d'application des règles d'instantiation de de résolution AGT. Pour cette partie du travail, nous nous sommes basés sur des

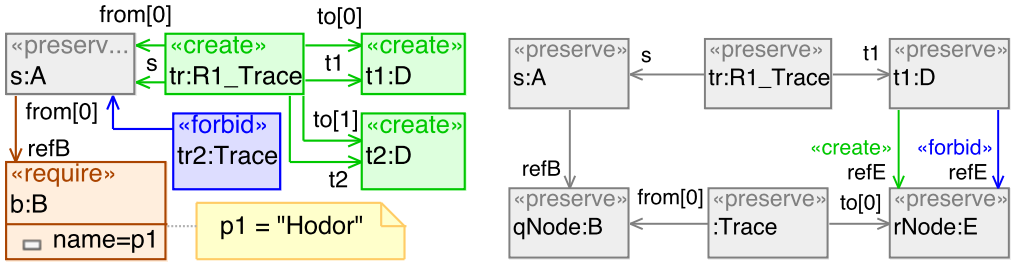



Figure 10: Règle d'instantiation $R1_{Inst}$ et règle de résolution $R1_{Res}^{t1,refE}$

travaux existants de traduction de OCL vers NGC [Radke *et al.*, 2015b]. Cependant, constatant que les traductions existantes ne permettent pas la prise en charge des ensembles ordonnés (`OrderedSet`s) OCL, nous avons abordé ce problème en particulier. Notre contribution fut alors de compléter les traductions existantes par des éléments supplémentaires assurant le respect de l'ordre des objets d'un ensemble ordonné lors de leur traitement.

Ces deux premières contributions relatives à la traduction d'ATL et de OCL vers la théorie AGT ont fait l'objet d'une publication [Richa *et al.*, 2015] qui a reçu le *Prix du Meilleur Article* à la conférence internationale ICMT'15.

### 4.1.4 Traduction des Postconditions en Préconditions – *Post2Pre*

Ayant définit la traduction des transformations ATL et des contraintes OCL vers la théorie AGT, nous passons à la définition de la traduction d'une postcondition en une précondition équivalente. Pour cela nous nous sommes basés sur les définitions relatives à la construction *weakest precondition* (*wp*) [Radke *et al.*, 2015a; Ehrig *et al.*, 2012a; Habel *et al.*, 2006a], en particulier à la construction *weakest liberal precondition* (*wlp*) de la plus faible précondition libérale, afin de définir l'analyse *Post2Pre* dont nous avons besoin.

La première difficulté de ce travail est que pour les transformations de graphes que nous traitons (celles correspondant à des transformations ATL) la construction *wlp* est théoriquement infinie. C'est pourquoi nous proposons une version bornée de cette construction qui conserve des propriétés similaires au *wlp*. Pour ce faire, nous introduisons un nouveau concept théorique de *l'itération bornée* pour lequel nous définissons une construction *wlp* bornée et nous prouvons formellement ses propriétés. En analysant alors une version bornée de la transformation, nous obtenons une construction *wlp* bornée qui fournit les propriétés désirées. Cependant le résultat d'une telle analyse est uniquement applicable à la version bornée et non plus à la transformation d'origine. C'est pourquoi nous définissons une nouvelle construction appelée *scopedWlp* qui ajoute au *wlp* une condition supplémentaire et nous prouvons formellement que *scopedWlp* est valable pour la

transformation non-bornée tout en étant une construction finie. Il faut toutefois noter qu'à la différence de $wlp$, $scopedWlp$ fournit une précondition libérale qui n'est pas la plus faible, mais qui assure les propriétés que nous recherchons.

Notre contribution fournit les définitions formelles de tous les concepts introduits ainsi que les preuves formelles de leurs propriétés. Il est important de noter que bien que ce travail soit illustré sur des transformations ATL, nos définitions et preuves ne sont pas spécifiques à ATL et sont donc applicables à des transformations de graphes arbitraires au delà du contexte de cette thèse.

### 4.1.5 Stratégies de Simplification de la Construction de Préconditions

Lorsqu'on vient à implémenter les analyses que nous proposons, il faut s'attarder sur le passage à l'échelle de ces analyses. En effet, la construction $wlp$ se base sur l'énumération de tous les recoupements (*graph overlaps*) possibles entre les graphes constituant les règles de transformation et les graphes constituant la postcondition. Cette énumération combinatoire a une complexité algorithmique très importante que nous estimons à au moins $O(2^N)$ où $N$ est le nombre d'éléments des graphes composant la postcondition et les règles. Concrètement cela se manifeste par une croissance exponentielle de la taille de la précondition au fur et à mesure que l'analyse progresse dans le traitement des règles de transformation. Ainsi, même pour des transformations et des postconditions de petite taille, l'analyse se termine le plus souvent par l'épuisement de la mémoire vive après plusieurs heures de calcul.

Pour atténuer la complexité élevée de $wlp$ nous avons développé plusieurs stratégies de simplification de la construction qui consistent à éliminer le plus tôt possible les recoupements qui ne contribuent pas à la satisfaction de la postcondition afin de limiter la croissance de la précondition calculée. Pour identifier les recoupements à éliminer nous exploitons les propriétés des transformations analysées. Par exemple lorsque l'analyse porte sur des transformations ATL exogènes[7] nous savons qu'il est impossible que le modèle d'entrée de la transformation contienne des éléments du métamodèle de sortie. En appliquant ce principe aux différentes règles à analyser, nous pouvons éliminer les parties de la précondition qui contiennent des éléments du métamodèle de sortie et empêcher l'accroissement inutile de cette précondition. De manière similaire nous pouvons également identifier les recoupements mettant en œuvre des nœuds de trace dans des situations que nous savons être impossibles dans la sémantique ATL.

[7] Une transformation est dite *exogène* lorsque le métamodèle d'entrée est différent du métamodèle de sortie.

En outre, nous proposons une version alternative du $wlp$ basée sur une construction que nous appelons $Post2Left$ qui est équivalente aux constructions de l'état de l'art mais qui permet d'appliquer les simplifications proposées dès que possible. Cela permet d'éviter de faire une grande portion de calculs inutiles. Il est important de noter que la construction $Post2Left$ ainsi que la plupart des stratégies que nous proposons ne sont pas spécifiques à ATL. Ainsi nos contributions pourraient être utiles à l'analyse de transformations AGT arbitraires au delà du contexte de cette thèse.

Grâce aux stratégies de simplification que nous proposons, une grande partie des exécutions qui auparavant se terminaient par l'épuisement de la mémoire arrive désormais à l'aboutissement de l'analyse en moins d'une minute et fournit des résultats. Cependant, comme le détaillera la section 5, ces gains de performance sont observables pour des données de taille modeste, et pour des cas plus réalistes l'analyse retrouve son comportement exponentiel et ne fournit pas de résultats utilisables. Néanmoins, nos contributions apportent des gains de performance significatifs et élargissent les limites de calculabilité de l'analyse.

#### 4.1.6 Outillage et Implémentation

Nous avons implémenté l'intégralité de nos contributions sous forme d'un outil Open Source appelé *ATLAnalyser*[8]. Pour les traductions $ATL2AGT$ et $OCL2NGC$, nous avons choisi le langage *Henshin* comme cible de la traduction car ce langage applique la sémantique AGT au même environnement Eclipse Modeling Framework (EMF) de ATL. Cela a facilité par la suite la validation de notre approche qui sera présentée en Section 5. L'analyse $Post2Pre$ est également implémentée sur la base de l'outillage Henshin et peut prendre en entrée des transformations Henshin résultant de la traduction $ATL2AGT$ ainsi que des transformations Henshin arbitraires.

Nous concluons à présent la présentation des contributions relatives à la première partie de la thèse qui porte sur la production de nouveaux d'intégration, et nous passons à présent à la seconde partie qui porte sur les oracles de ces tests d'intégration.

### 4.2 Spécification et Oracles de Tests de Transformations Model-to-Text

#### 4.2.1 Principe Genéral: Patrons de Spécification

Le second problème traité dans cette thèse est de concevoir une approche pour la spécification d'oracles de tests d'intégration d'un Générateur Automatique de

[8] *ATLAnalyser*, https://github.com/elericha/atlanalyser

Code (GAC) visant à valider la syntaxe textuelle du code généré. Pour cela nous proposons de spécifier le code attendu en sortie du GAC au moyen de la notion de *patrons de spécification* (specification template).

Un patron de spécification décrit une portion du code généré. Il exprime une contrainte simple de la forme suivante:

$$\textit{condition d'application} \quad \Rightarrow \quad \exists\,(\textit{patron de texte})$$

Concrètement, un patron de spécification prend la forme suivante:

Listing 1: Structure générale d'un patron de spécification

```
1 [template templateName (input0 : type0, input1 : type1 ...) ? (oclGuard)]
2 verbatim text, interleaved with [oclQueries/] between brackets,
3 %<regularExpressions>% between percent delimiters
4 and loop statements expressing repeating patterns:
5 [for ( iterator | collection )]
6   This text repeats for all elements of the collection.
7   We can use [oclQueriesInvolvingIterator] here.
8 [/for]
9 [/template]
```

Un patron de spécification se compose ainsi des éléments suivants:

(1) Des éléments d'entrée: un ensemble d'éléments du métamodèle d'entrée de la transformation qui influe la portion de code à spécifier. Ces éléments sont déclarés entre parenthèses à la première ligne de la définition du patron.

(2) Une garde: une contrainte sur les éléments d'entrée qui définit une condition d'applicabilité du patron. Cette condition est spécifiée après le symbole `?`.

(3) Un patron de texte qui doit exister lorsque la garde est satisfaite: le patron de texte est défini à l'aide de quatre types de contenu:

   a. du texte verbatim (en rouge ci-dessus)

   b. des requêtes OCL portant sur le modèle d'entrée, délimitées par les symboles `[` et `/]` (en noir ci-dessus)

   c. des expressions regulières délimitées par les symboles `%<` et `>%` (en bleu ci-dessus)

   d. des structures de répétition (commandes `[for ...][/for]`)

Ainsi, les éléments (1) et (2) expriment la partie *condition d'application* de la contrainte, et les éléments de (4) constituent le *patron de texte* de la contrainte. La spé-

cification de la sortie attendue de l'ACG s'exprime alors sous forme d'un ensemble de patrons définissant les portions de codes qui doivent exister dans le code généré. Nous expliquons dans la section suivante comment comment cette spécification est utilisée comme oracle de test.

### 4.2.2 Oracles de Tests

En pratique, nous utilisons le langage Acceleo [Acceleo, accessed 2015] pour exprimer les patrons de spécification définis dans la section précédente ce qui permet d'obtenir une spécification exécutable. Comme illustré par la Figure 11, nous considérons un test du GAC dont l'entrée est un modèle de test et la sorte est le code généré. Le but est donc de détérminer si le code généré est valide, c'est à dire s'il correspond à la spécification.

Pour ce faire, les patrons de spécification sont exécutés sur le modèle de test. Pour chaque patron de spécification dont la condition d'application est satisfaite, les requêtes OCL et les commandes de répétition sont exécutées sur le modèle d'entrée pour produire un *pattern attendu* composé de texte verbatim et d'expressions régulières. Le résultat de l'exécution de l'ensemble des patrons de spécification est donc un ensemble de patterns attendus, chacun décrivant une portion de code qui doit exister dans le code généré. Nous pouvons alors déterminer la validité de la sortie du test en recherchant la correspondance des patterns attendus dans la sortie du test par des opérations de *Match* d'expressions régulières. Si tous les patterns attendus ont une correspondance dans la sortie du test, le résultat du test est un *Succès*, sinon le résultat du test est un *Echec*.

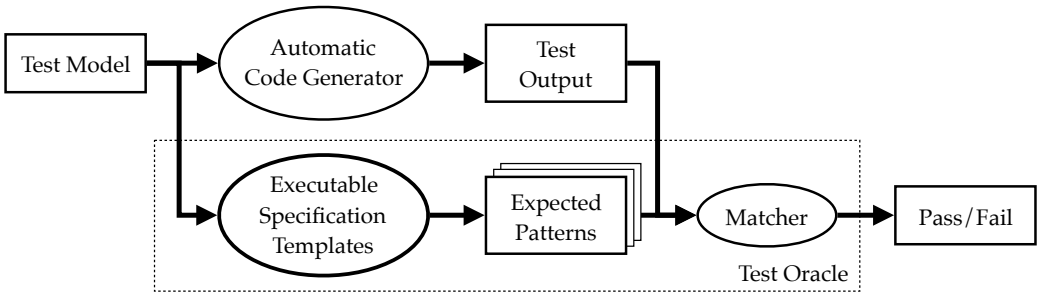



Figure 11: Utilisation des patrons de spécification comme oracles de tests

### 4.2.3 Example d'Utilisation: QGen

Nous avons appliqué notre approche de test à un générateur de code appelé QGen [AdaCore, accessed 2015] développé par AdaCore, le partenaire industriel de cette thèse. QGen prend en entrée des modèles Simulink et produit en sortie du

code C et/ou Ada. Nous avons développé l'outillage nécessaire à notre approche et nous avons spécifié le code attendu en sortie de QGen pour un sous-ensemble du langage d'entrée Simulink à l'aide de patrons de spécification. Par example, les trois patrons de spécification suivants définissent une partie du code généré pour un élément d'entrée Simulink de type **UnitDelay**.

Listing 2: Patrons de spécification de l'élément **UnitDelay**

```
1  [template public Compute(block : Block)
2  ? (block.BlockType() = 'UnitDelay' and block.OutDataType().IsScalar())]
3  [block.OutVar()/] = [block.MemVar().Optional_Cast()/];
4  [/template]
5
6  [template public Compute(block : Block)
7  ? (block.BlockType() = 'UnitDelay' and block.OutDataType().IsVector())]
8  for (i = 0; i <= [block.OutDataType().NumElements() - 1/]; i++) {
9    [block.OutVar().at('i')/] = [block.MemVar().at('i')/];
10 }
11 [/template]
12
13 [template public Compute(block : Block)
14 ? (block.BlockType() = 'UnitDelay' and block.OutDataType().IsMatrix())]
15 for (i = 0; i <= [block.OutDataType().NumElements() - 1/]; i++) {
16   for (j = 0; j <= [block.OutDataType().NumElements() - 1/]; j++) {
17     [block.OutVar().at('i', 'j')/] = [block.MemVar().at('i', 'j')/];
18   }
19 }
20 [/template]
```

Les patrons de spécification font appels à des fonctions réutilisables telles que `Optional_Cast()` qui retourne une expression régulière décrivant l'utilisation potentielle d'un conversion de donnée. Cette fonction est définie de la manière suivante.

```
1  [template public Optional_Cast(exp : String) post (trim())]
2  %<(>%(([Any_Type()/]) [exp/])%<|>%[exp/]%<)>%
3  [/template]
4
5  [template public Any_Type(arg : OclAny) post (trim())]
6  %<(GAINT8|GAINT16|GAINT32|GAUINT8|GAUINT16|GASINGLE|GAREAL|GABOOL)>%
7  [/template]
```

L'utilisation des expressions régulières permet de simplifier les patrons de spécification en éliminant des détails jugés non primordiaux. Par exemple le premier patron de spécification de **UnitDelay** décrit l'existence d'une affectation de variable qui comprend éventuellement une conversion de données. Cependant les condi-

tions précises menant à l'existence de la conversion de données n'ont pas à être explicitées car cet aspect est jugé non primordial dans cette partie de la spécification. Ainsi, grâce aux expressions régulières nous avons pu éviter de détailler les conditions d'occurrence de la conversion de données et nous avons obtenu une spécification simple.

En appliquant notre approche à un sous-ensemble des éléments Simulink pris en compte par QGen, nous avons pu obtenir des oracles de tests automatiques que nous avons appliqués à une base de tests existante. Les résultats de cette expérimentation concrète de notre approche seront présentés en Section 5.3.

# 5 Résultats Expérimentaux

Après avoir présenté les contributions principales de la thèse dans la section précédente, nous nous attardons à présent sur la validation et l'évaluation expérimentale des approches proposées. Pour rappel, le premier volet de la thèse a porté sur la problématique du test unitaire et du test d'intégration d'une chaine de transformation de modèles. Afin de produire un modèle de test d'intégration étant donné une exigence de test unitaire non-satisfaite, nous avons proposé de traduire l'exigence de test en amont le long de la chaine jusqu'au langage d'entrée. En supposant une spécification en ATL des transformations de la chaine, nous avons ainsi proposé une approche en deux étapes:

1. $ATL2AGT$: Traduction de la transformation dans la théorie des Transformations Algébriques de Graphes (AGT)
2. $Post2Pre$: Traduction arrière de la contrainte exprimant l'exigence de test unitaire en une précondition équivalente à l'aide de la construction théorique Weakest Liberal Precondition ($wlp$)

Ensuite, le second volet de la thèse s'est attardé sur les oracles de tests d'intégration d'un Générateur Automatique de Code (GAC), la problématique étant de concevoir un moyen de spécifier le code attendu en sortie du GAC au regard de sa structure textuelle. Nous avons alors proposé une approche de spécification à base de *patrons de spécification* décrivant les portions de code à générer. L'exécution des patrons de spécification sur un modèle de test permet de déterminer automatiquement si la sortie du test contient les portions de code spécifiées et constitue alors un oracle de test automatique.

Dans les sections suivantes nous résumons la validation expérimentale des approches proposées.

## 5.1 Validation de *ATL2AGT*

*ATL2AGT* traduit des transformations ATL en des transformations AGT équivalentes. Pour valider notre traduction nous l'implémentons en Java et nous adoptons une approche de vérification par test illustrée par la Figure 12. Plusieurs transformations ATL provenant de sources variées sont considérées lors de cette validation. Tout d'abord chaque transformation est traduite en AGT. Ensuite un ensemble de modèles de test est donné en entrée des deux versions ATL et AGT de la transformation. Les modèles résultants sont comparés avec EMFCompare[9]. Cette comparaison prend en compte l'ordre des éléments ce qui permet de valider les aspects de traduction relatifs au traitement des ensembles ordonnés.

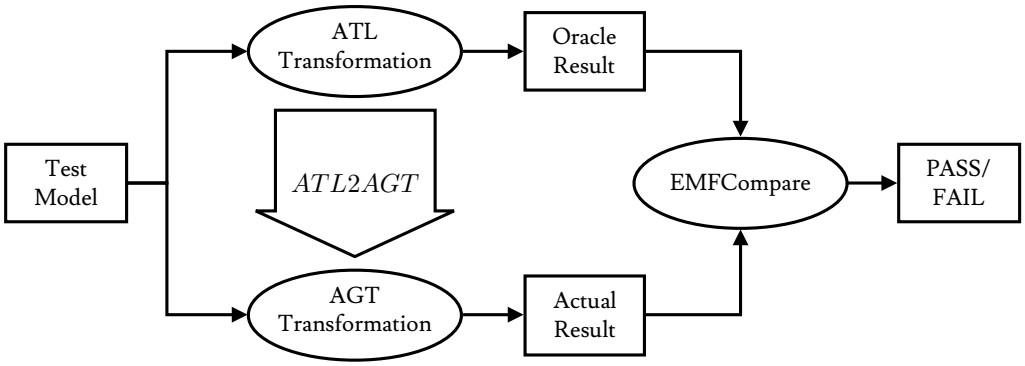



Figure 12: Validation de ATL2AGT

Pour cette validation nous avons considéré 3 catégories de transformations ATL:

1. Des transformations sélectionnées du Zoo ATL [ATL Zoo, accessed 2015] (Families2Persons, Class2Relational) et de publications existantes de l'état de l'art [Büttner *et al.*, 2012b]).

2. *SimpleCMG*: une version simplifiée d'une étape de transformation du générateur de code QGen développé par AdaCore. QGen n'est pas développé en ATL mais nous avons modelé cette étape de transformation en ATL.

3. Des transformations créées spécifiquement pour tester des fonctionnalités spécifiques d'ATL ou de la traduction vers AGT.

Par cette approche de validation nous avons montré que *ATL2AGT* traduit fidèlement les transformations ATL en des transformations AGT qui se comportent de manière identique à l'exécution. Les détails de la validation sont présentés dans le tableau Table 3 où nous indiquons pour chaque transformation de test les

[9]EMFCompare, `https://www.eclipse.org/emf/compare/`

fonctionnalités ATL employées ainsi que des métriques concernant le nombre de règles ATL en entrée et le nombre de règles AGT en sortie.

| | **Families2-Persons** | **Class2-Relational** | **ER2REL** | **QGen Code Generation** |
|---|---|---|---|---|
| **Metrics** | | | | |
| ATL rules | 2 | 6 | 6 | 6 |
| ATL bindings | 2 | 22 | 13 | 30 |
| Instantiation rules | 2 | 6 | 6 | 6 |
| Resolving rules | 8 | 23 | 15 | 32 |
| **ATL Features** | | | | |
| Default Resolve | X | X | X | X |
| `resolveTemp` | | | | X |
| `if-then-else` | X | | | |
| Helpers | X | X | | X |
| Attribute binding | X | X | X | X |
| Reference binding | | X | X | X |
| `OrderedSet{}` | | X | | X |
| `union()` | | X | X | X |
| `select()` | | X | | X |
| `collect(),at()` | | | | X |

Table 3: Liste des transformations de test et des fonctionnalités testées

A présent nous passons à la validation de la traduction de postconditions en préconditions: *Post*2*Pre*.

## 5.2 Validation de *Post*2*Pre*

*Post*2*Pre* est un ensemble de constructions qui transforment une postcondition $d$ d'une transformation $T$ en une précondition qui assure la satisfaction de la post-condition. Comme la construction Weakest Liberal Precondition $wlp$ est théoriquement infinie, nous avons proposé les constructions $wlp(T_{\leq N}, d)$ et $scoperWlp(T_{\leq N}, d)$ comme constructions alternatives bornées et nous avons démontré leurs propriétés théoriques. Nous avons également proposé des stratégies de simplification visant à améliorer le passage à l'échelle de nos constructions. Nous avons implémenté ces constructions en Java, et nous avons réalisé deux types de validations que nous présentons dans cette section: la validation fonctionnelle et l'analyse de passage à l'échelle.

### 5.2.1 La Validation Fonctionnelle

La validation fonctionnelle a deux objectifs. Premièrement il s'agit de vérifier que les préconditions calculées par notre implémentation exhibent bien les pro-

priétés théoriques de $wlp$ et $scopedWlp$. Deuxièmement, il faut montrer que les stratégies de simplification que nous proposons n'affectent pas la validité du résultat de l'analyse.

Puisque nos travaux sur *Post*2*Pre* sont limités aux aspects structurels des transformations et des contraintes, nous considérons pour cette validation les trois transformations ATL suivantes qui ne mettent pas en œuvre des manipulations d'attributs scalaires (*i.e.* entiers, chaines de caractères *etc.*.):

1. *SimpleATL*: Une transformation ATL simple composée de 2 règles ATL.
2. *PointsToLines*: Une autre transformation ATL simple composée de 3 règles ATL qui transforme un ensemble de points en un un ensemble de lignes connectant les points.
3. *SimpleCMG*: Une version simplifiée d'une étape de transformation de QGen, le générateur de code de Simulink vers C/Ada développé à AdaCore. Bien que QGen lui-même n'est pas développé en ATL nous avons modelé cette étape de transformation en ATL.

Pour chacune des transformations ci-dessus, nous avons considéré plusieurs postconditions pour lesquelles nous avons calculé les préconditions. Comme les constructions que nous validons sont définies en fonction d'un paramètre $N$, nous avons considéré différentes valeurs de $N$ lorsque cela ne pose pas de problèmes de passage à l'échelle. Les caractéristiques des tests de validation considérés sont résumé dans le tableau 4.

| $T$ | **Nombre de règles ATL** | **Nombre de règles AGT** | $N$ | **Nombre de postconditions** |
|---|---|---|---|---|
| *SimpleATL* | 2 | 5 | 1 | 13 |
| | | | 2 | 1 |
| | | | 3 | 1 |
| *PointsToLines* | 3 | 7 | 1 | 1 |
| | | | 2 | 2 |
| *SimpleCMG* | 6 | 24 | 1 | 2 |

Table 4: Tests de validation fonctionnelle de *Post*2*Pre*

Pour chacune des postconditions $d$ considérées nous calculons $wlp(T_{\leq N}, d)$ et $scopedWlp(T_{\leq N}, d)$ et nous validons par inspection manuelle que les préconditions obtenues correspondent aux définitions théoriques. Cette première validation est effectuée sans utiliser nos stratégies de simplification. Ensuite dans une seconde étape, nous activons nos stratégies de simplification une à une et nous effectuons de nouveau les analyses en vérifiant que les préconditions calculées sont identiques

ou équivalentes. Nous vérifions ainsi que nos stratégies de simplification ne détériorent pas la validité du résultat de l'analyse.

Après cette validation fonctionnelle, nous passons à présent à l'évaluation du passage à l'échelle de notre analyse, notamment du gain de performance apporté par nos stratégies de simplification.

### 5.2.2 L'Analyse de Passage à l'Echelle

Nous avons proposé dans cette thèse les quatre stratégies de simplifaction suivantes:

S1. Propriétés standards des Nested Graph Conditions (NGC) et de la logique de permier ordre.

S2. Selection des règles contribuant à la postcondition.

S3. Elimination des conditions en contradiction avec la sémantique ATL. Nous appelons cette stratégie le filtre *ATLSem*.

S4. Elimination des conditions contenant des éléments qui ne peuvent pas exister en précondition des règles en raison de la nature exogène de la transformation. Nous appelons cette stratégie le filtre *ElemCr*.

Afin de quantifier l'effet de nos stratégies, nous procédons à des exécutions différente de la même analyse en activant et désactivant les stratégies et en mesurant des métriques concernant la taille des préconditions calculées et le nombre de calculs nécessaires à l'analyse. Pour cela nous considérons la transformation $T$ et la postcondition $Post_1$ suivantes et nous calculons $wlp(T, 1_{\leq}, Post_1)$:

$$T = SimpleATL = R1_{Inst}\downarrow;\ R2_{Inst}\downarrow;\ R1_{Res}^{t1,refD}\downarrow;\ R1_{Res}^{t1,refE}\downarrow;\ R2_{Res}^{t,refD}\downarrow$$

$$Post_1 = \exists\left(\ \boxed{d{:}D} \xrightarrow{refE} \boxed{e{:}E}\ \right)$$

Nous effectuons 5 exécutions différentes de la construction avec les configurations suivantes des stratégies de simplification.

| **Identifiant de l'exécution** | **S1** | **S2** | **S3** | **S4** |
|---|---|---|---|---|
| *ExNoFilters* | ✓ | × | × | × |
| *ExATLSem* | ✓ | × | ✓ | × |
| *ExElemCr* | ✓ | × | × | ✓ |
| *ExBoth* | ✓ | × | ✓ | ✓ |
| *ExAll* | ✓ | ✓ | ✓ | ✓ |

S1 est toujours activée dans toutes les exécutions car cette stratégie n'est pas configurable dans notre implémentation. S3 et S4 sont évaluées séparément en

*ExATLSem* et *ExElemCr* puis ensemble en *ExBoth*. Et enfin *ExAll* permet d'évaluer S2.

Etant donné que le but des stratégies de simplifications est de réduire le nombre de calculs de recoupements de graphes (graph overlaps), nous mesurons durant chaque exécution le nombre de calculs $N_{Ov}$ effectué pour chaque règle de la transformation analysée.

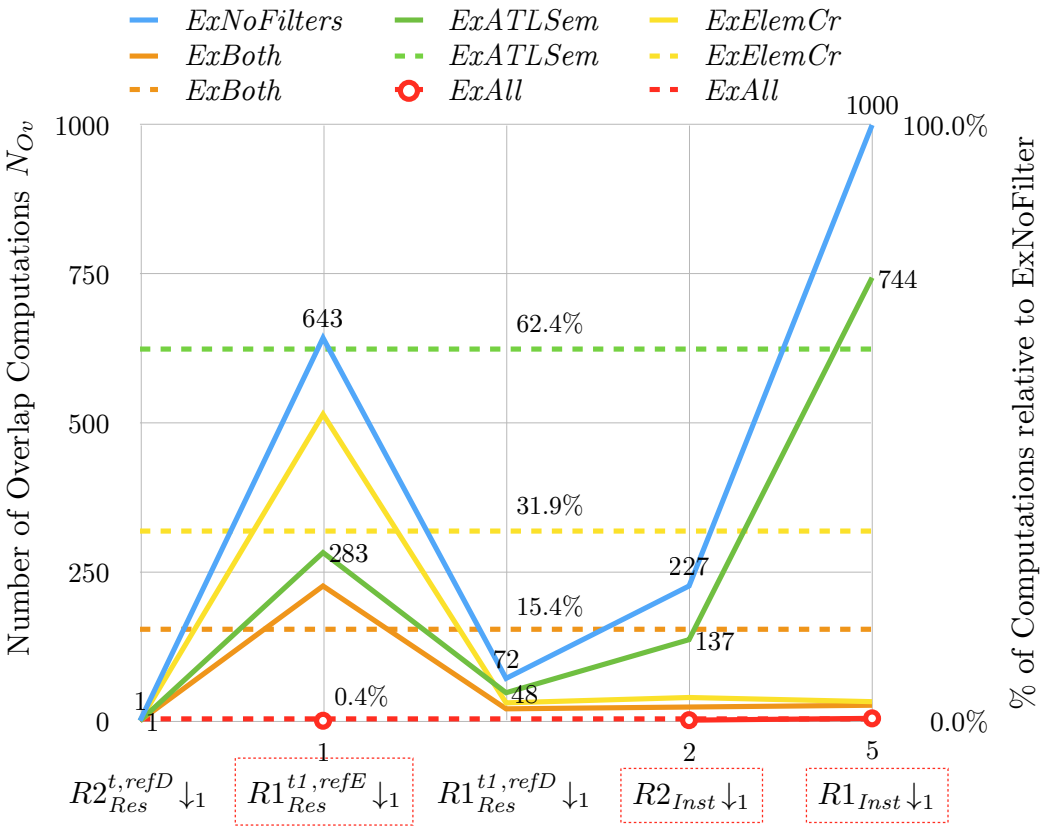



Figure 13: Nombre de calculs de recoupements de graphes $N_{Ov}$

Nous représentons les résultats des mesures de $N_{Ov}$ dans le graphe de la Figure 13. Les courbes en traits pleins représentent le nombre de calculs effectués pour chaque règle durant l'exécution tandis que les courbes en traits pleins représentent le pourcentage de calculs sur l'ensemble d'une exécution par rapport à l'exécution sans filtres *ExNoFilters*. Nous observons que pour *ExBoth* seulement 15.4% des calculs sont effectués pour aboutir au même résultat final ce qui constitue une réduction significative. L'effet est encore plus prononcé dans *ExAll* où le résultat est calculé avec uniquement 0.4% des calculs. Cela indique que nos stratégies sont très efficaces puisqu'elles permettent une réduction significative du nombre de calculs à faire durant l'analyse.

Par ailleurs, nous constatons que certaines exécutions qui sans l'intervention de nos stratégies mènent à l'épuisement de la mémoire après plus de 10 minutes, sont achevés en quelques secondes avec l'activation de nos stratégies avec une utilisation maximale de 15% de la mémoire totale, ce qui représente une amélioration significatives des performances de l'analyse.

Néanmoins, lorsque nous passons à des transformations et des postconditions plus grandes les performances ne sont pas toujours aussi satisfaisants. Ainsi avec une postcondition composée de 4 nœuds et 3 connections, et une transformation composée de 15 règles, l'analyse ne peut aboutir et se termine par la saturation de la mémoire malgré l'activation de nos stratégies. Ainsi dans la Figure 14 nous observons que malgré l'augmentation exponentielles du nombre de recoupements éliminés par nos filtres au fur et à mesure de l'avancement de l'analyse (l'échelle verticale est exponentielle), la taille de la précondition et le nombre de recoupements effectués conserve une évolution également exponentielle. Cela indique que malgré leur efficacité démontrée dans les paragraphes précédents, nos filtres ne modifient pas la nature exponentielle de l'algorithme et ne permettent pas un vrai passage à l'échelle.

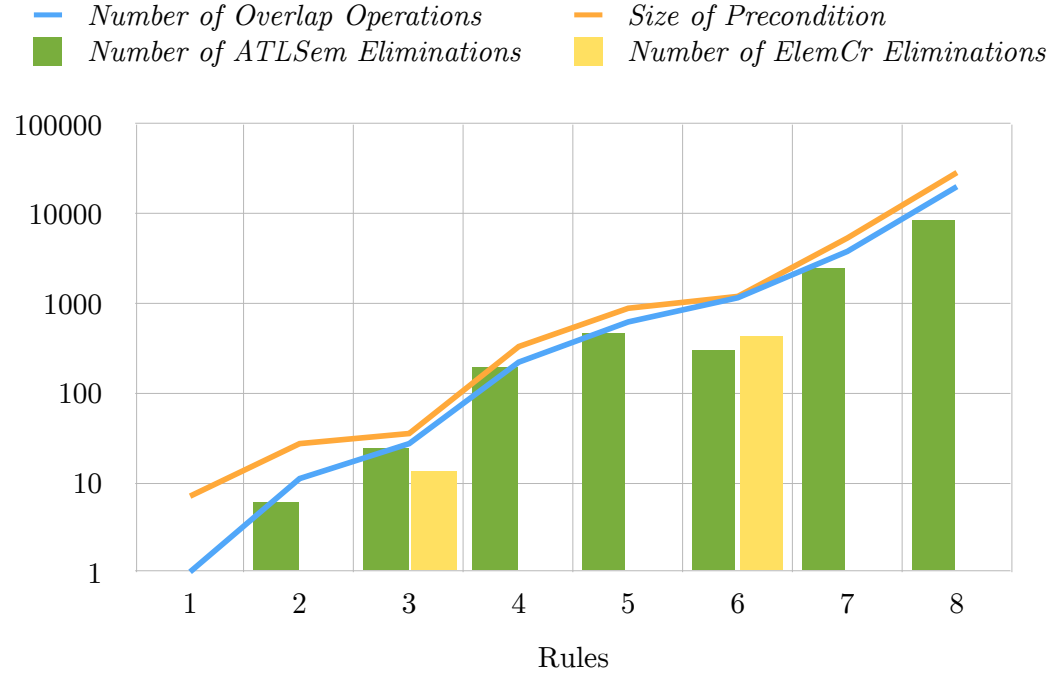



Figure 14: Métriques de l'exécution de *wlp* pour *SimpleCMG*

Une analyse plus détaillées des résultats ainsi qu'une discussion concernant la parallélisation de l'analyse et la gestion de la mémoire sont disponibles dans le chapitre 10 du manuscrit en Anglais. A présent nous passons à la validation de la seconde partie de cette thèse qui a porté sur les oracles automatiques de tests

d'intégration des générateurs de code.

## 5.3 Validation de l'Approche de Spécification et d'Oracles de Test Model-to-Text

Pour valider notre approche d'oracles de tests de transformations model-to-text, nous l'avons déployée au sein de l'équipe de développement du générateur de code QGen à AdaCore. Pour ce faire nous avons conçu et déployé un outil appelé *TOR Toolkit* basé sur l'environnement Eclipse et notamment la technologie Acceleo. A l'aide de cet outil 3 membres de l'équipe ont écrit les patrons de spécification décrivant le code requis en sortie de l'outil pour un sous-ensemble du langage d'entrée de l'outil. Ainsi, pour 38 types de blocs Simulink[10] nous avons obtenu une spécification que nous avons utilisé comme oracle automatique de test.

Au moment du déploiement de l'approche, l'équipe disposait déjà d'une base de tests significative contenant un large ensemble de modèles de test Simulink. Pour le sous-ensemble de tests concernant les 38 types de blocs considérés pour notre validation expérimentale, nous avons utilisé l'exécution des patrons de spécification et le *Match* des patterns attendus avec la sortie de l'outil pour valider les résultats de tests. Les oracles automatiques de notre approche ont ainsi permis la détection de plusieurs erreurs dans l'implémentation de QGen. Par exemple pour certaines configurations des blocs Simulink, les oracles ont détecté des incompatibilités entre le code généré et les patrons exigés par la spécification, ce qui a mené à la mise en évidence d'erreurs dans l'implémentation des blocs en question.

Cependant ce déploiement expérimental a aussi mis en évidence plusieurs limites de notre approche. Tout d'abord, l'application à Simulink a nécessité l'énumération manuelle d'un très grand nombre de patrons de spécification pour chaque type de bloc Simulink afin de tenir compte de tous les paramètres de configuration affectant la sémantique du bloc et donc le code à générer. Cette tâche est difficile à faire manuellement et est hautement sujette à l'erreur. C'est pourquoi pour envisager un déploiement réel de nos propositions dans l'avenir, il sera nécessaire de traiter l'aspect de la variabilité du langage d'entrée en s'inspirant par exemple des travaux de [Dieumegard *et al.*, 2012; Dieumegard *et al.*, 2014b; Dieumegard *et al.*, 2014a].

[10] au moment de la rédaction QGen prend en charge environ 120 types de blocs Simulink différents en entrée

# 6 Conclusion

Cette thèse a traité du sujet de la qualification des générateurs de code automatique (GACs), un sujet de grand intérêt dans le domaine de l'avionique aujourd'hui en raison de la réduction de coûts que la qualification peut apporter à la production de logiciels avioniques critiques. En effet, générer le code source d'une application critique avec un GAC qualifié permet l'élimination des vérifications coûteuses du code source généré. Cependant, la qualification des ACGs reste à ce jour un processus très coûteux qui impose de nombreuses contraintes sur le développement du GAC et nécessite des activités de vérification approfondie. Dans ce processus, nous avons mis l'accent en particulier sur la problématique du test des GACs et nous avons cherché à proposer des techniques de test efficaces autant par leur automatisme que par leur rigueur afin d'être compatibles avec les standards de qualification.

## 6.1 Rappel des Problématiques

Dans la première partie de la thèse nous avons cherché à assurer le même niveau de détection d'erreurs des tests unitaires dans les chaînes de transformation de modèles en n'utilisant que des tests d'intégration qui sont de mise en œuvre plus facile. Nous avons déterminé que cela est possible avec des techniques existantes permettant d'extraire des exigences de tests unitaires et des oracles de tests unitaires, et de vérifier leur satisfaction durant l'exécution des tests d'intégration. Cependant, une fois que des exigences de tests non satisfaites sont identifiées, les approches existantes ne fournissent pas un moyen de créer de nouveaux tests d'intégration pour les satisfaire. Ainsi, le premier problème abordé par cette thèse était de proposer une approche de traduction arrière des exigences de tests unitaires en des contraintes équivalentes sur l'entrée de la chaine de transformation permettant la création de nouveaux tests d'intégration.

Ayant investigué la production des tests d'intégration dans la première partie de la thèse, la deuxième partie s'est intéressé aux oracles de ces tests. Compte tenu du contexte de qualification et de certification, nous avons déterminé que ces oracles doivent être basés sur une spécification syntaxique du code source généré, qui est facilement compréhensible par les différentes parties prenantes. Par ailleurs, étant donné le grand nombre de tests nécessaires, il était nécessaire d'adopter des oracles automatiques. Ainsi, le deuxième problème abordé par cette thèse était de proposer une approche syntaxique d'oracles de test automatiques pour les GACs.

## 6.2 Résumé des Contributions

### 6.2.1 Traduction Arrière des Exigences de Test

Dans la première partie de la thèse, nous avons proposé une approche de traduction arrière des exigences de test unitaires d'une transformation en des contraintes équivalentes sur l'entrée de la transformation. Pour ce faire, nous nous sommes basés sur les concepts théoriques existants de la théorie des Transformations Algébriques de Graphes (AGT) [Ehrig *et al.*, 2006]. En supposant que la transformation est spécifiée dans le langage de transformation de modèle ATL [Jouault *et al.*, 2008], notre approche est composée de deux étapes:

1. Traduire la transformation ATL en un transformation équivalente dans la théorie AGT.

2. Considérer l'exigence de test comme une postcondition de la transformation, et la traduire en une précondition grâce à une analyse que nous appelons *Post*2*Pre* basée sur la construction théorique de *weakest liberal precondition* ($wlp$) en AGT.

Dans ce cadre général nous avons apporté plusieurs contributions:

**Traduction ATL vers AGT**

Nous avons définit une traduction du sous-ensemble purement déclaratif d'ATL vers le formalisme AGT. La difficulté de ce travail réside dans le traitement des mécanismes de *resolving* d'ATL ainsi que les ensembles ordonnés en OCL car ces sémantiques n'ont pas d'équivalents directs dans la théorie AGT. Cette contribution a fait l'objet d'une publication [Richa *et al.*, 2015] qui a reçu le *Prix du Meilleur Article* à la conférence internationale ICMT'15.

**Traduction de Postconditions en Préconditions**

Nous avons définit un ensemble d'analyses appelées *Post*2*Pre* basées sur la construction théorique $wlp$, pour les transformations purement structurelles.

Constatant que $wlp$ peut être théoriquement infinie, nous avons proposé une nouvelle notion d'*itération bornée* et nous avons défini la construction $wlp$ qui lui correspond afin d'obtenir une construction finie. Ensuite nous avons introduit une nouvelle construction *scopedWlp* qui rend le résultat de la construction bornée applicable à la transformation analysée qui contient des itérations non bornées. Pour tous les nouveaux concepts introduits, nous avons fourni les définitions et preuves théoriques qui ne sont pas limitées au seul contexte d'ATL, mais s'appliquent à des transformations AGT arbitraires. Par conséquent, ces résultats théoriques représentent une contribution à la

théorie AGT qui pourrait avoir des applications au delà du contexte de ce travail.

**Stratégies de Simplification**

Reconnaissant la complexité de calcul élevé de la construction $wlp$, nous avons proposé plusieurs stratégies pour y remédier. La complexité se manifeste concrètement par des préconditions calculées dont la taille augmente rapidement et finit par épuiser la mémoire disponible. Par conséquent, nous avons proposé tout d'abord des stratégies de simplification permettant l'élimination des parties non pertinentes des conditions calculées pour réduire leur taille. Certaines de ces stratégies sont spécifiques à ATL tandis que d'autres peuvent s'appliquer à des transformations AGT arbitraires. Ensuite, nous avons proposé une version alternative et équivalente de $wlp$ qui permet d'appliquer les stratégies de simplification le plus tôt possible dans le calcul et évite de réaliser des opérations inutiles. Grâce à ces propositions, plusieurs analyses *Post*2*Pre* qui auparavant aboutissaient à l'épuisement de la mémoire parviennent à s'exécuter correctement lors de notre évaluation expérimentale.

L'ensemble de nos contributions a été validé expérimentalement et nos implémentations sont disponibles sous forme d'un outil Open Source appelé *ATLAnalyser*[11].

### 6.2.2 Oracles de Test des Transformations Model-to-Code

Dans le cade de la seconde partie de cette thèse, nous avons proposé une approche d'oracles de tests de générateurs de code (ou transformations Model-to-Code) basée sur un nouveau concept de *patrons de spécification*. Chaque patron de spécification décrit un pattern de code que la transformation doit générer. Ce pattern est exprimé grâce à la juxtaposition de portions de texte verbatim, de requêtes au modèle d'entrée et d'expressions régulières. Cela permet de spécifier la structure du code généré de par sa syntaxe textuelle concrète.

L'exécution des patrons de spécification sur un modèle d'entrée de test produit un ensemble de *patterns attendus*. Ainsi une opération de *Match* de ces patterns attendus avec le code généré par l'exécution du test permet de valider le résultat et ainsi de déterminer la conformité de l'outil avec sa spécification.

Nous avons déployé et évalué cette approche au sein de l'équipe de développement du générateur de code QGen à AdaCore.

[11] ATLAnalyser, `https://github.com/elericha/atlanalyser`

## 6.3 Perspectives de Poursuite

Dans les perspectives de poursuite de ces travaux, l'analyse des transformations ATL est une piste prometteuse. En effet notre traduction d'ATL au formalisme AGT permet l'application de plusieurs analyses de la théorie AGT aux transformations ATL telles que la détection de conflits entre règles de transformation [Ehrig *et al.*, 2012b] et la preuve formelle de correction [Pennemann, 2009; Habel and Pennemann, 2009; Poskitt, 2013].

Enfin, après s'être intéressés à la qualification d'un outil dans cette thèse, il sera intéressant de se pencher sur la question de la qualification d'une ligne de produit plutôt que d'un seul outil, ou encore à la qualification d'un outil non seulement pour le domaine de l'avionique, mais aussi pour d'autres domaines critiques tels que l'automobile et le ferroviaire. En effet les fournisseurs d'outils tels que AdaCore doivent avoir des propositions satisfaisants des besoins très différents chez des clients d'industries variées. Malgré les avancées techniques de l'ingénierie des lignes de produits [Pohl *et al.*, 2005] le contexte du logiciel critique nécessite de prendre en compte les aspects de certification et de qualification ce qui soulève des questions de recherche difficiles [Hutchesson and McDermid, 2013]. Il serait intéressant d'intégrer ces facteurs dans la modélisation des lignes de produit afin de concevoir une famille d'outils qualifiés variant non seulement de par leurs fonctionnalités mais également de par les standards de sureté ciblés par leur qualification.

# Chapter 1

# Introduction

## Contents

## 1.1 General Context

Software embedded in aircrafts, trains and cars is said to be *critical* because its malfunction can have catastrophic consequences and endanger human lives. Given the high stakes, critical software is subject to strict safety standards such as the DO−178C certification standard in the avionics domain. This standard regulates all aspects of the development of the software, including planning, organisation, development and verification activities. As a result, developing certified critical software has a very high cost, the bulk of which resides in the extensive verification activities that are required for all development artifacts throughout the process.

Given the high cost of verification, manufacturers seek to eliminate some of the verification activities related to the source code by using Automatic Code Generators (ACGs) and showing that they produce source code exhibiting the required correctness properties by construction. This is only possible if the ACG itself is developed in conformance with the DO−330 tool qualification standard.

Qualifying an ACG is as rigorous and demanding as certifying critical embedded software. However the cost of qualification is compensated by its repeated use in the life cycle of a critical system, particularly in maintenance phases. At each use of the ACG the verifications avoided represent a return on the initial investment in the qualification of the tool. As a result, tool qualification is today a highly active topic of research and discussion within the community of critical system manufacturers.

Tool providers such as AdaCore, the industrial partner of this thesis, need to adopt efficient methodologies in the development and verification of ACGs in order to provide cost effective qualified tools. In this thesis, we focus in particular on the aspects of testing which proves to be highly expensive given the high criteria of thoroughness and coverage required for qualification.

In the next section, we identify the precise problems tackled by our work in the scope of testing ACGs for their qualification.

## 1.2 Problem Statement

Within the context of testing ACGs we identify two specific problems based on concrete observations in the practice of code generator development.

### 1.2.1 Unit Testing v/s Integration Testing

An ACG is typically designed as a Model Transformation Chain (MTC) which applies a series of model transformation steps sequentially. Testing such an ACG for its qualification requires both *unit testing* of each step of the MTC in isolation and *integration testing* of the complete tool as a whole. However industrial practice in code generator and compiler development[1] shows that unit testing is tedious to perform because of the complexity of the required test data (models at their intermediate representation) and stubs [Stuermer *et al.*, 2007]. Conversely integration testing is easier to carry out because it does not require stubs and test models are simpler to produce and maintain.

Fortunately, existing research suggests that it is possible to achieve the confidence of unit testing in a MTC using solely integration test data [Bauer *et al.*, 2011]. Combining this study with other existing works on the testing of model transformations, we can extract from each model transformation of the MTC a set of *unit test requirements* and a *unit test oracle*. Unit test requirements are constraints over the internal intermediate representations that the MTC operates on, characterising different test cases to be covered. The test oracle is an automatic procedure to validate the result of a unit test. With these two concepts, we can consider only integration tests, execute them, and assess the coverage of unit test requirements and the validity of oracle outputs automatically. This method provides the same confidence of unit testing by relying on oracles for unit tests, but uses only integration test models.

However, when *non-satisfied* unit test requirements are identified, the method stops short of providing a way to create new integration tests to satisfy them. Producing such tests manually is not straightforward because a unit test requirement expresses a constraint over an internal language of the MTC while the integration test model should be in the input language of the chain. Therefore producing such a model requires reasoning inversely over several steps of the MTC which is difficult to achieve manually.

Consequently the first problem tackled in this thesis is:

*Translating a non-satisfied unit test requirement backwards along a model transformation chain into an equivalent constraint over the input language of the chain.*

If we can propose a solution to this problem, then with the support of existing test generation approaches, we can produce new integration tests that satisfy the remaining non-covered unit test requirements. As a result, we would be able to

[1] compilers also consist of chains and raise issues similar to ACGs

provide a complete testing strategy that complies with the coverage requirements of qualification, provides the same confidence as unit testing, and uses solely integration test data.

### 1.2.2 Specification-based Oracles of Code Generator Integration Tests

Having so far discussed the production of integration test models, we also need to investigate the corresponding *test oracles i.e.* the means to determine if an integration test passes or fails. Since integration tests aim at covering unit test requirements of all the transformations involved in the MTC, then there will likely be a large number of tests. It is therefore desirable if not necessary to have automatic test oracles determine the validity of test outputs.

In an integration test of an ACG the input is a model and the output is source code. In the literature we find that most test oracle approaches aim at validating the *semantics* of source code, *i.e.* verifying that the behavior of the generated code is compatible with the semantics of the input model. However in the context of qualification and certification, the *syntax* of the generated source code is also of importance and must be shown to comply with the applicable code standard. This syntactic aspect is less developed in the literature of ACG testing. Additionally, the qualification standard requires test oracles to be based on the specification, which in turn is subject to several reviews by different stakeholders. As a result, it is important that this specification be readable and easily understandable. Consequently, the second problem that we tackle in our work is the following:

*Devising a specification and automatic test oracles approach focusing on the syntax of the generated source code and the readability of the specification.*

Having defined the two main problems tackled by this study, we present in the next section a summary of the solutions that we propose to address them.

## 1.3 Summary of Contributions

To address the first problem of backward translation of test requirements we propose to transpose it to the formal framework of Algebraic Graph Transformation which provides the means to translate and reason on constraints. As for the second problem of determining the verdict of integration tests, we propose a specification and test oracles approach based on the textual concrete syntax of the generated source code. In the following sections we develop our solutions and the conceptual and concrete contributions that they provide.

### 1.3.1 Backward Translation of Test Requirements

Concerning the first problem of backward translation of test requirements, we note first that MTCs can include several steps (*e.g.* typically ∼15 for code generation). As a result we propose an *iterative* approach that performs the backward translation of test requirements step by step. Solving the sub-problem of backward translation for one model transformation step of the chain would theoretically unlock the solution for the complete chain.

In the scope of this thesis, we propose and validate an approach to the one-step backward translation of a unit test requirement into an equivalent constraint over the input of the preceding transformation. To do so, we propose to rely on existing theoretical concepts in the formal framework of Algebraic Graph Transformation (AGT) [Ehrig *et al.*, 2006]. Assuming that the transformation is specified in the ATL model transformation language [Jouault *et al.*, 2008], our approach is composed of two steps:

1. Translate the preceding transformation from ATL to the formal framework of AGT.
2. Consider the unit test requirement as a postcondition, *i.e.* a constraint on the output of the transformation, and translate it into an equivalent precondition, *i.e.* a constraint over the input of the transformation. The translation uses a set of analyses called *Post*2*Pre* that we propose based on the formal construction of the *weakest liberal precondition* (*wlp*) in AGT.

With these two steps, we obtain a constraint over the input of the preceding transformation that ensures the satisfaction of the unit test requirement. Since the result is a constraint, it can again be considered as a postcondition of its preceding transformation, and thus the analysis can be iterated.

In this general approach, we put forward several contributions:

**Translation of ATL to AGT**

We define a translation of purely declarative ATL transformations to equivalent AGT transformations. The challenge in this translation is to support semantical features such as the ATL resolving mechanisms and OCL[2] ordered sets which do not have direct equivalents in the AGT framework. This contribution was the subject of a publication at ICMT'15 [Richa *et al.*, 2015] which received the *Best Paper Award* of the conference.

**Translation of Postconditions to Preconditions**

We define a set of analyses called *Post*2*Pre* based on the theoretical construc-

[2]OCL, `http://www.omg.org/spec/OCL/`

tion of *wlp*, within the scope of purely structural transformations. As a result this contribution does not yet support the transformation of object attributes.

Noting that *wlp* can in fact be theoretically infinite, we propose a new *bounded iteration* construct and define its corresponding *wlp* construction to obtain a finite construction for a bounded version of the analysed transformation. Then we introduce a new construction called *scopedWlp* which extends the previous one and makes it applicable to unbounded iteration. Thus with *scopedWlp* we obtain a finite precondition that ensures the satisfaction of the postcondition for the original unbounded transformation. For all these new concepts we provide formal definitions and proofs of correctness that are not limited to ATL but also apply to arbitrary AGT transformations. As a result, this set of theoretical results constitutes a contribution to the theory of AGT that can have applications beyond the context of this work.

**Simplification Strategies**

Acknowledging the high computational complexity of the *wlp* construction, we put forward several strategies to alleviate it. The complexity concretely manifests through very large computed preconditions that ultimately exhaust the available memory. Consequently, we first propose simplification strategies allowing the elimination of irrelevant portions of computed preconditions to reduce their size. Some of these strategies are specific to ATL while others can apply to arbitrary AGT transformations. Then we propose a modified construction of *wlp* which is equivalent to the original one but allows early simplification of conditions and avoids unnecessary computations. With the combination of these proposals, we manage to successfully perform analyses which were previously infeasible due to the high complexity.

Our proposals are validated experimentally in this thesis and their implementation is available in the form of a tool called *ATLAnalyser*[3].

With this first set of contributions, we have tackled the general problem of producing new integration tests to cover particular unit test requirements. We now move to the solution of the second problem which is determining the outcome of these integration tests with specification-based automatic test oracles.

### 1.3.2 Specification and Test Oracles of Model-to-Code Transformations

To address the second problem raised in this thesis, we introduce the concept of *specification templates* and propose a specification approach based on it. Specification templates express the patterns of code that should be generated in terms of

[3] *ATLAnalyser*, `https://github.com/elericha/atlanalyser`

verbatim textual code interspersed with queries to the input model and regular expressions. This allows to specify the structure of the generated code in terms of its concrete syntax, and thus satisfies the needs of qualification.

We propose a test oracles approach based on the *execution* of the specification. Executing a specification templates over an input model yields so-called *expected patterns* which are automatically matched in the output of the test to determine if the test passes. Beyond the automation of test oracles, this approach is very appealing in the context of qualification because it establishes the specification as the direct decider of the outcome of tests.

At the request of AdaCore, the industrial partner of the thesis, this solution was developed and implemented for the specific needs of the Simulink® [4] to C code generator, QGen[5], developed within the company. Our proposals were validated through an experimental deployment of the approach within the QGen development team. The proposals were not generalised beyond that context in the scope of this thesis, however we believe that the core concepts of our approach can be extended to arbitrary model-to-text transformations.

## 1.4 Document Organisation

The remainder of this document is organised as follows.

**Background and Current Advances**

First in Chapter 2 we give an overview of the industrial context of qualification and certification and the relationship between the two processes, highlighting the challenges of testing in qualification. Then in Chapter 3 we report on the current advances in the testing of model transformations and model transformation chains to identify approaches that are relevant to our context and assess their adequacy to our needs.

**Problem Statement and General Approach**

In Chapter 4 we define precisely the two problems tackled in our work concerning the production of integration tests and the specification-based test oracles of these tests. We highlight the main challenges and explain why existing techniques do not provide satisfactory solutions. Then in Chapter 5 we give an overview of the solutions we proposed to the identified problems.

**Contributions**

The contributions addressing the first problem in this thesis are detailed in 3

[4] Simulink®, `http://www.mathworks.com/products/simulink`
[5] QGen, `http://www.adacore.com/qgen`

chapters. Chapter 6 presents the translation of ATL transformations to AGT transformations. Then in Chapter 7 we present the analysis that we propose for the resulting AGT transformations. We detail the constructions involved in *Post*2*Pre* and demonstrate their theoretical properties. In Chapter 8 we investigate the implementation of the analysis and propose simplification strategies to tame its complexity.

As for the contribution addressing the second problem, it is presented in Chapter 9 which details the syntactic specification and test oracles approach that we propose.

**Experimental Validation**

The experimental validation of our proposals is detailed in Chapter 10. First we validate the translation of ATL to AGT by comparing ATL and AGT executions of transformations. Then we assess the translation of postconditions to preconditions, first with a functional validation focusing on the correctness of the resulting preconditions, and then with a scalability analysis assessing the efficiency of our simplification strategies. Finally, our syntactic specification and test oracles approach is assessed through an experimental deployment within the team developing the code generation technology at AdaCore.

# Chapter 2

# Background: Qualification and Certification in the Avionics Domain

## Contents

## 2.1 Introduction

Software embedded in aircrafts trains and cars is said to be *critical* because its malfunction can have catastrophic consequences and endanger human lives. For this reason critical software is subject to strict safety standards that regulate all aspects of its development. In our work we focus particularly on the avionics domain where the safety standard in effect is the DO−178C *certification standard*. Certification requires extensive verifications of all development artifacts to ensure that no errors are introduced during the process. This includes independent reviews of artifacts and extensive requirements-based testing that are very costly to perform and constitute the bulk of the overall cost of certification.

The verification cost can be reduced by producing development artifacts automatically with tools that guarantee that their result is correct by construction. For example, the specification of the critical software can be defined in the form of models given as input to an Automatic Code Generator (ACG) which automatically produces the implementation source code. In that case the verification of the generated source code can be avoided by verifying instead that the ACG ensures by construction that the generated code is correct. This verification of the ACG itself is called *qualification* and is also regulated by a standard, the DO−330 *tool qualification standard*, in the avionics domain.

Qualifying an ACG is as demanding and rigorous as certifying critical embedded software. The qualification standard regulates all aspects of the tool development process including organisational and technical aspects. Like certification, it also requires extensive verification of all development artifacts of the ACG including independent reviews and thorough requirements-based testing. AdaCore, the industrial partner of this thesis, is developing a qualifiable[1] Simulink to C source code ACG called *QGen*. This thesis was launched in that context with the objective of investigating techniques to support the qualification process. Ultimately we focused on the specific aspect of the testing involved in the qualification of code generators which is particularly costly.

In the following sections we will give an overview of the certification and qualification processes and their relationship, emphasizing lastly the main focus of this thesis which is the testing of code generators for their qualification.

[1] as will be explained, a tool cannot be qualified independently of its usage context

## 2.2 Certification of Critical Airborne Software

Critical avionics software must be developed in conformance with the DO−178C certification standard. The standard regulates all aspects of the development process, defining the roles involved, the development and verification activities that must be carried out, as well as the artifacts and documents that must be produced as certification evidence. This evidence is assessed by a certification authority to verify that all aspects of the standard were applied, and a certificate is issued allowing the software to be deployed. Certification authorities are for example the Federal Aviation Administration (FAA) for the USA and the European Aviation Safety Agency (EASA) for Europe.

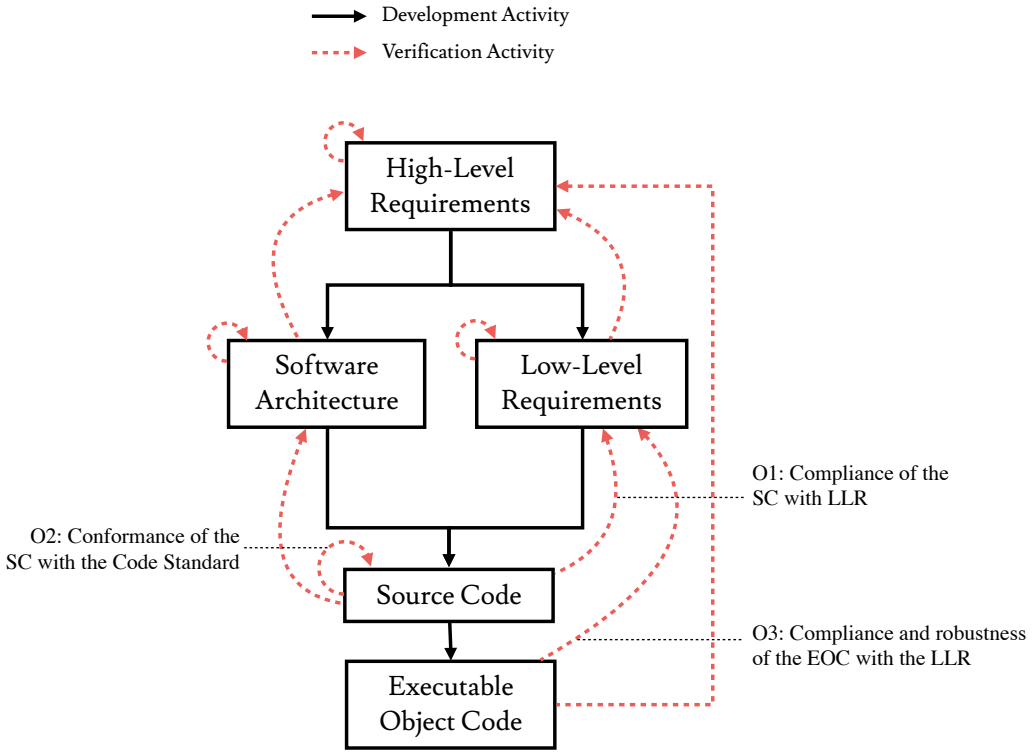



Figure 2.1: Simplified DO−178C compliant development process

### 2.2.1 Planning and Development Processes

Certification starts with the planning process which defines and coordinates all development and verification activities. Among other things this phase defines a set standards to be followed during the development. The *Code Standard* will be of particular interest in our context. It is a set of constraints on the way source code will be developed, defining complexity restrictions on the degree of coupling between

software components, the nesting levels of control structures and the complexity of logical and numeric expressions. These restrictions must be followed during coding and we will discuss verification activities addressing that aspect.

After the planning process, the development process outlined in Figure 2.1 is started. The first development activity is the specification of *High-Level Requirements* (HLR) of the software which describe all the required functionality of the software. The second step is the design of the software which results in a *Software Architecture* (SA) and a set of *Low-Level Requirements* (LLR) which specify the component architecture and the interfaces and the detailed functionalities of each component. Then architecture and requirements are implemented into *Source Code* (SC). Compiling the SC results in the final *Executable Object Code* (EOC) which is the actual embedded software.

### 2.2.2 Verification Activities and Verification Objectives

Verification activities are carried out after or concurrently with the development process when appropriate. Verification activities which are depicted by dashed arrows in Figure 2.1 concern all development artifacts. In general, each artifact should be verified (looping arrows) for accuracy, consistency and conformance to the design standards defined earlier in the planning phase of the life cycle. Artifacts must also be verified for compliance with the artifacts from which they were developed (upward arrows). We will focus particularly on the verifications concerning the SC and the EOC. The objectives of these verifications include many aspects such as the compliance with the LLR and SA, conformance to code standards, stack and memory usage, fixed point arithmetic overflow and resolution *etc.* We will focus particularly on the following verification objectives, also highlighted in Figure 2.1:

O1. *Compliance of the SC with LLR* must be shown typically via *independent* review of the SC. Independence is achieved when the verifier is a different person than the developer of the item being verified.

O2. *Conformance of the SC with the Code Standard* must be shown typically with a combination of analysis tools and manual reviews.

O3. *Compliance and robustness of the EOC with the LLR* must be shown via LLR-based testing. Testing is conducted by analysing the LLR and developing *test cases*. Each test case defines a range of inputs (or specific inputs) to the software component under test, and the expected output or behavior. Both *normal range* test cases and *robustness* test cases must be developed. The former use inputs within normal ranges defined by requirements while the

latter use abnormal inputs to test the response of the software to abnormal conditions (which should also be specified by robustness LLRs). Test cases should exercise equivalence classes and boundary values of numeric inputs.

Testing is subject to a Structural Coverage Analysis which determines which code structures were actually executed. For the highest levels of criticality, the criteria of this coverage analysis is the *Modified Condition/Decision Coverage* MC/DC criteria.

**The Modified Condition/Decision Coverage Criteria** This coverage criteria is used on several occasions in the standard when dealing with *decisions* expressed as boolean formulas over *conditions*. For example, for a decision $D = (a \wedge b) \vee c$ is composed of 3 conditions $a$, $b$ and $c$. The MC/DC criteria states that three requirements must be satisfied:

1. The decision must take all possible outcomes (*true* and *false*) at least once.
2. Each condition of the decision must take all possible outcomes at least once.
3. Each condition must be shown to independently affect the outcome of the decision, *i.e.* varying the condition alone while keeping the others fixed affects the outcome of the decision.

In general, for a decision composed of $N$ independent conditions the criteria can be satisfied with $N + 1$ test cases [Hayhurst *et al.*, 2001]. Applying MC/DC criteria to source code consists in analysing the decisions controlling the flow of execution of the program (*if* statements, *while* statements *etc.*) and analysing the outcomes that they take during the execution of tests to determine whether the above requirements of the MC/DC criteria are met. This is typically done using specialised coverage analysis tools.

Industrial experience shows that the bulk of the cost of certification resides in verification activities such as the above due to the thoroughness required to ensure the safety of the critical software. Consequently industrials seek to reduce this cost. Focusing on the above verification activities, we discuss next how they can be eliminated or reduced by using a *qualified* automatic code generator.

## 2.3 Claiming Certification Credit with Qualified Tools

When the SC is developed manually, the verification activities must all be performed. However the situation is different when the SC (or part of it) is generated

automatically using an Automatic Code Generator (ACG). For example, to generate code from the LLR, they must be expressed in a precise and machine-readable formalism. In this thesis we consider a Model-Driven Engineering (MDE) approach where LLR are expressed as models with precise structure and semantics. LLR models can then be automatically implemented into SC via model transformation by an ACG. This scenario is depicted in Figure 2.2.

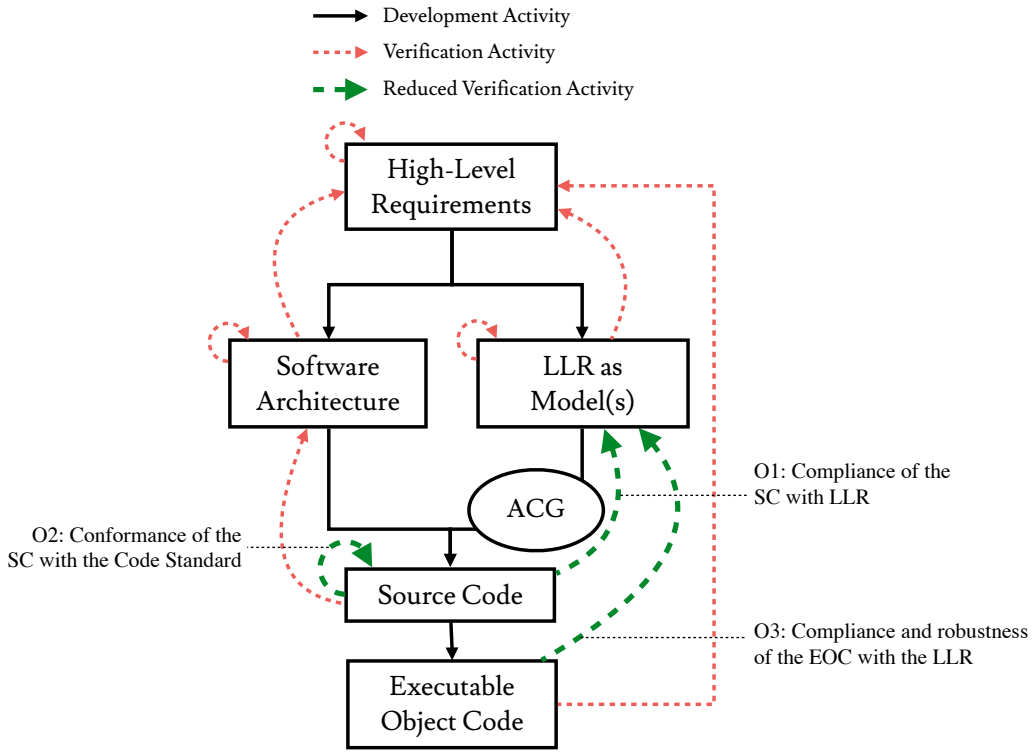



Figure 2.2: Simplified DO−178C compliant development process with a qualified ACG

Using an ACG naturally reduces the development cost, however our aim is to reduce the verification cost which is more significant by eliminating some verification activities. In DO−178C if a tool is used to produce artifacts of the software and its output is not verified, then the tool must be *qualified* to show that the tool ensures by construction the properties to be verified, thus eliminating the need of verification. This is called *claiming certification credit* for a verification objective from the use of a qualified tool. For example, in Figure 2.2 if the ACG is used to claim credit for the conformance of the SC with the Code Standard (O2) and its compliance with the LLR (O1), then its qualification must demonstrate that the generated SC always conforms with the Code Standard and always complies with the LLR. If the compliance of the EOC with the LLR (O3) is also to be removed, then the qualification must also cover that aspect. Therefore as we will detail later, qualification

activities depend on the claimed certification credit.

At this stage, to avoid confusion between the software to be certified and the tool to be qualified we distinguish them with the following terminology:

- the *application* is the embedded software that must be *certified*.
- the *tool* (an ACG in our case) is *qualified* in order to eliminate verification activities in the certification of an application.
- In qualification, the term *operational* (*e.g.* operational context, operational requirements) refers to the usage context of the tool which is the certification of the application.

In the avionics domain, tool qualification must be performed according to the DO−330 qualification standard. The qualification process is very similar to the certification process, and for ACGs it is arguably as demanding and rigorous as certification. However the cost of qualification is justified by the reduced certification cost. In fact when coding is done manually, it is not uncommon to perform SC verifications multiple times during the life cycle of the certified software. This can occur in advanced stages of the process if verifications (*e.g.* integration testing) uncover errors requiring code alterations. In that case SC verifications have to be performed again after each modification or set of modifications. Moreover, in the maintenance phase following deployment, code modifications may also occur and would also require reverification of the SC. As a result, SC verification is often a recurrent activity. Replacing the recurrent verification with a one-shot qualification of an ACG provides a significant reduction of the certification cost. In the next section we detail the qualification process of an ACG.

## 2.4 Qualifying an ACG

Tool qualification is defined by the DO−330 qualification standard. Like certification, the standard defines planning, development and verification activities that must be carried out, and artifacts and documents to be provided as qualification evidence that the standard was applied faithfully.

### 2.4.1 Coupling of Certification and Qualification – *Qualifiable* Tools

The purpose of qualification is to obtain certification credit for the verification activities eliminated in the application life cycle. This credit can only be granted within the context of the certification of the application. This means that a tool may not be qualified on its own. It is only qualified in the context of a project certification and

for a precise certification credit. If a subsequent project uses the same tool without any change, then the previous qualification evidence may be reused as is, provided that the new usage context is shown to be equivalent to that of the previous qualification. Otherwise, if the usage context is different or if changes are needed in the tool, an impact analysis of the changes must be performed to determine any needed re-verification activities.

Consequently, tool vendors such as AdaCore cannot provide *qualified* tools independently of a specific certification project. Instead they can provide *qualifiable* tools for which most of the qualification evidence has been developed by the tool vendor with assumptions regarding the usage context. When an avionics constructor acquires a qualifiable tool and its qualification evidence, he needs to assess that his usage context is compatible with the existing qualification evidence, and take necessary action otherwise. This may include modifications to the qualification documents or even the tool itself, and may trigger re-verification activities.

### 2.4.2 Tool Qualification Planning

The first phase of qualification is the planning, starting with the assessment of the role of the tool in the application life cycle and the precise identification of the claimed *certification credit*. This determines a *Tool Qualification Level* (TQL) specifying the level of rigor (TQL–1 the highest, to TQL–5 the lowest) required in the qualification. Certain verification activities are required for TQL–1 but not in TQL–5. In our context, the tool is an ACG whose output is part of the airborne software. The ACG must be qualified at TQL–1, the most rigorous level, since it may introduce errors in the critical software, and all verification activities are necessary.

The planning process also defines the tool development standards to be followed. A Tool Requirements Standard specifies the methods, notations and tools used to develop the various requirements of the qualified tool. A Tool Code Standard specifies the programming languages used as well as coding rules, naming conventions, complexity constraints *etc.*. Requirements and code must be developed in conformance with these standards and verification activities will be deployed to verify this conformance.

### 2.4.3 Tool Development

Then the tool development process is initiated. It is depicted in Figure 2.3 and is evidently very similar to the certification process. First, the *Tool Operational Requirements* (TOR) are defined. They specify the required operation of the tool as a black

box from the user's perspective. TORs must include enough detail regarding the functionality of the tool to support the claim of certification credit for the eliminated certification activities. For example, if credit is claimed for the compliance of the SC with LLR in the certification process of Section 2.2, then the TOR must describe precisely what the generated code structure should be and how elements of input LLR models should be implemented by the ACG into the generated SC.

The second step is the development of *Tool Requirements* (TR) which also describe the function of the tool but this time in a more detailed manner, addressing nominal and failure modes. Each TR should trace to one or more TORs. However there may be so-called *derived* TRs that do not trace to TORs and that specify for example additional tool functionality unused in the operational context.

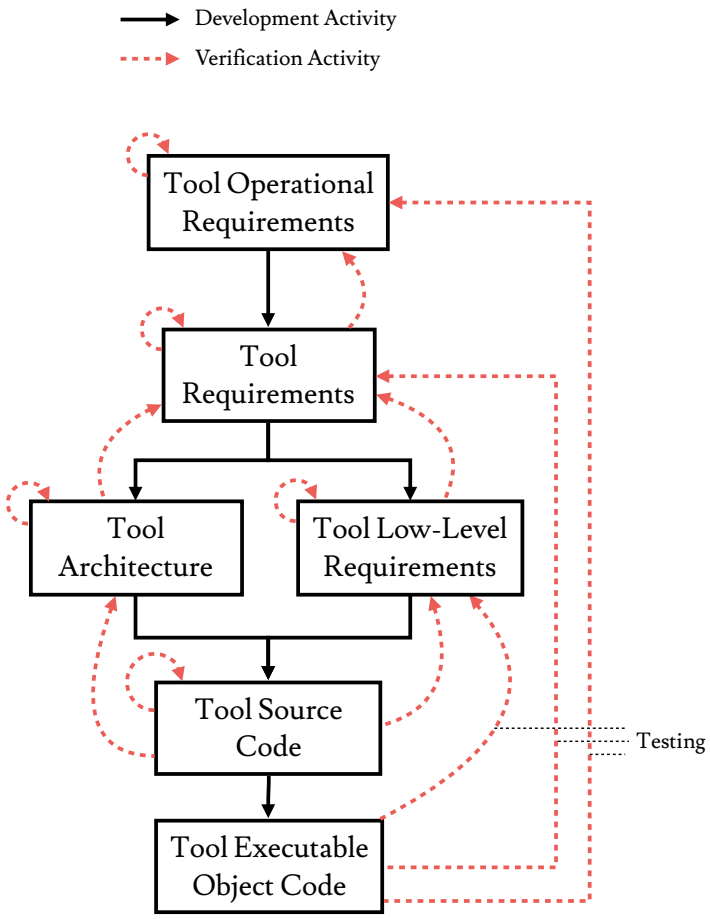



Figure 2.3: DO−330 compliant tool development process

Then the *Tool Architecture* (TA) is defined and specifies the various software components of the tool and their interfaces, and the *Tool Low-Level Requirements* (TLLR) specify the functionality of each component. TA and TLLR are defined in a way to jointly achieve the functionality required in the TR. If needed, there can be several

levels of requirements refinement before reaching the final TLLR level. Each TLLR must trace back to one or more TRs such that all components functionality serves higher level tool functionalities and there is no unintended functionality. Finally, the TA and TLLR are implemented into the *Tool Source Code* (TSC) which when compiled and linked yields the *Tool Executable Object Code* (TEOC) *i.e.* the actual resulting tool.

*Traceability* is an important aspect in this development process. Each artifact is required to have trace links of some form to the artifacts from which it was developed. Trace data is then used in various ways by verification activities to show that no unintended (*i.e.* untraceable) artifacts were inadvertently introduced, but also to support impact analyses when artifacts need to be modified.

### 2.4.4 Tool Verification

Several verification activities must be conducted on the development artifacts and are depicted as dashed arrows on Figure 2.3. Like in the certification process, each artifact should be verified for accuracy and consistency, and conformance with the requirements and code standards defined in the planning phase. This kind of verification is depicted as looped dashed arrows. Artifacts must also be verified for compliance with the artifacts from which they were developed (relying on trace data). These verifications are depicted as upward dashed arrows. For the TEOC the verification of compliance with TLLR, TR and TOR is performed through requirements-based testing.

Many of the above verifications must be performed with *independence*. In all verifications of compliance, the person verifying an artifact must be different from the person who developed the artifact. Independence may be achieved granularly over individual artifacts. For example, the person implementing a component $A$ of the SC can review a component $B$ of the TSC for compliance with the TLLR if he/she did not participate in the coding of $B$.

#### Tool Operational Verification and Validation

In the aforementioned verifications, those concerning the TOR are of particular importance and constitute the *Tool Operational Verification and Validation* (TOV&V). TOV&V includes the following objectives:

1. The TORs are sufficient to automate the operational development activities and eliminate the operational verification activities within the claimed certification credit.

2. The TEOC complies with the TORs.

Consequently, TOV&V depends on the claimed certification credit. For example, if credit is claimed for the compliance of the generated code with a Code Standard (objective O2 in Section 2.2), then first a review of the TOR must ensure that the specification of the TOR complies with the Code Standard. Then, TOR-based testing should verify that the generated code complies with the TOR, and by transitivity with the Code Standard.

However if credit is claimed for the compliance of the EOC with LLR (objective O3 in Section 2.2) then the TOV&V is more complex. In a traditional certification process, O3 is demonstrated with LLR-based testing that must satisfy the MC/DC structural coverage criteria. If LLR-based testing is to be removed, it must be replaced with an equivalent verification in TOV&V. For O1 the reviews of the TOR must ensure that the generated code as specified by the TOR implements the model semantics correctly, and for O3 TOR-based testing should be equivalent to LLR-based testing, *i.e.* it should involve compiling the generated code itself using the same compiler used for the certified application, executing the generated EOC itself to show its compliance with the model from which it was generated, and perform a coverage analysis to show MC/DC structural coverage of the generated EOC. Only then can the LLR-based testing be confidently eliminated.

Given the complexity of the testing involved in tool qualification the next section will detail this aspect further.

## 2.5 Requirements-Based Testing in Qualification

Testing is one of the most costly aspects in qualification. This is because the standard requires high levels of exhaustiveness in the selection of test cases as well as in the coverage of the software structure. For this reason we have decided to focus this thesis on this specific aspect. Given its large scope of applicability, the standard does not require specific representations for requirements and test artifacts since this can vary greatly depending on the qualified tool. However it provides the general steps that should be followed and the artifacts that should be produced.

#### Test Cases Development

In line with the requirements-centered philosophy of qualification, the selection of test cases and the definition of expected results is based solely on the requirements: TOR, TR and TLLR. From each requirement a set of *test cases* is developed where each test case identifies the set of inputs to the tool or the tested component,

and the expected result or the pass/fail criteria. As with all qualification artifacts, test cases must be traced to the requirements from which they were developed. The development of test case is required to be systematic and thorough.

For requirements involving numeric data, equivalence classes and boundary values should be used for the definition of test cases. To give a rough example, if a requirement relies on a numeric parameter $p$ and distinguishes different behaviors if $p < 0$, $p = 0$ or $p > 0$, then 3 test cases must be developed, one for each range of $p$. For requirements involving logic formulas the MC/DC criteria applies to these formulas: test cases should be developed in a way to show that each individual condition independently affects the outcome of the logic formula. Robustness test cases must also be developed using inputs outside of the nominal ranges of the tool to test the capability of the tool to handle abnormal inputs and issue appropriate error messages. For example for an ACG it is very important that the tool avoids generating wrong or partial output in the event of errors.

All test cases are implemented into *test procedures* which specify how the testing environment is setup, how stubs are setup, how the test is executed and how the pass/fail verdict is determined. As explained in the previous section, TOR-based testing depends on the claimed certification credit. When claiming credit for compliance of the compliance of EOC with LLR in the operational context, then TOR-based test procedures should include compilation and execution of the generated code itself, and not only of the tool.

#### Analysis of Requirements-based Testing

After executing all tests, an analysis of the test results and the trace data is performed. First test failures are investigated to determine and correct errors. Then the trace data is analysed to ensure that all test procedures are executed and that test coverage of all requirements is achieved: each requirement is associated to at least one test case, and normal and robustness test cases were defined in the manner explained earlier.

Then structural coverage analysis of the TSC is performed to determine if there are code structures that were not exercised by the requirements-based testing. The coverage criteria required for source code is also MC/DC. When non-covered code structures are found, analysis should determine the reason. If the non-covered code is due to a shortcoming in the definition of test cases, then additional test cases should be developed. If the non-covered code corresponds to dead code or unused functions of the tool, it must either be removed or justified by explaining why it does not affect the functionality of the tool.

#### Tooling

We have discussed that the standard does not require specific representations for requirements and test artifacts as long as they comply with the definitions of the standard. This opens the door to using structured representations of requirements and test cases allowing to apply systematic methodologies in testing activities and even automatic tools to support the generation of test cases and test data. Such tools *may* need to be qualified themselves depending on their role and the confidence put in their output. However since they do not contribute to the actual software, these tools would be qualified at a much less rigorous TQL than the ACG which is not very costly.

Given these possibilities and the high cost involved in testing, we have decided to focus this thesis on investigating methodologies and tooling to support the testing activities involved in the qualification of ACGs. First in the next section we will present the general scope and assumptions that we adopt for this investigation before proceeding in Chapter 3 to an overview of existing works within the identified scope.

## 2.6 Scope of the Research: Model Transformation Chains

#### QGen

On the industrial side, the context of this work is the *QGen* toolkit which is a code generation and model verification toolkit developed at AdaCore for Simulink and Stateflow models. It supports a wide subset of Simulink and Stateflow elements that are relevant for the development of critical software. The main product of the toolset is an ACG targeting C and Ada source code which can be configured with optimisation options and custom behavior. QGen also integrates this code generation tool with other AdaCore compiler, emulation, and static analysis technology to support testing the generated code on target architectures as well as performing static analysis of Simulink/Stateflow models.

More specifically, the context of this thesis is the ACG of the QGen toolset. AdaCore seeks to provide a *qualifiable* ACG of Simulink models to C source code. This means providing customers with not only the tool, but also the evidence (or a significant part of it) necessary for its qualification, including planning documents, requirements documents and verification documents complete with test cases, test execution results *etc.* Many difficult questions arise in that context. The needs of customers can differ, requiring potentially different configurations of the tool and as a result different artifacts in the qualification evidence. How different qualifi-

cation evidence can be produced in a flexible and efficient process that remains compliant with the standard is a very difficult research question that we do not try to answer in this thesis. Instead we will assume a smaller objective which is the qualification of one configuration of the ACG and investigate ways to optimise that process, focusing specifically on the testing activities. In the remainder of the thesis, we will use the term *QGen* to refer to the Simulink to C code generator intended for qualification.

**Model Transformation Chains**

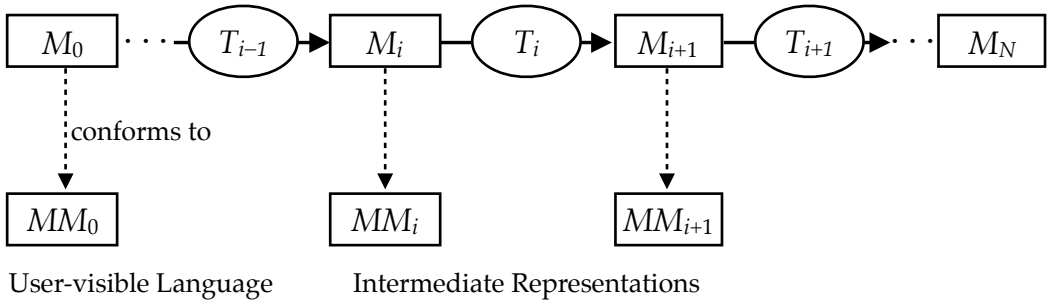



Figure 2.4: Model transformation chain

Even though the industrial context of our work is QGen, our work will not be limited to that tool but will extend to the general family of code generators and other software sharing the same kind of architecture. Like many model-based tools, an ACG is typically designed as a *Model Transformation Chain* (MTC). The kind of MTCs that we will study have the structure depicted in Figure 2.4. A chain takes as input a model $M_0$ conforming to the metamodel $MM_0$ used by the users of the chain, and applies a series of transformations $M_i \xrightarrow{T_i} M_{i+1}$. The input model is thus transformed step by step into intermediate models conforming to internal intermediate metamodels which are not exposed to the user. Finally, the last transformation produces the output $M_N$ of the chain which in the case of an ACG is the source code.

In the literature, MTCs can have more complex structures with for example alternative branches implementing configurable behaviors of the software. However we adopt this simpler form because it corresponds better to the context of qualified tools. As explained earlier an ACG is qualified in a specific context with a particular set of requirements. Therefore there is no need to consider configuration aspects. Additionally, given the high cost involved in qualification, it is advisable to simplify as much as possible the qualified tool.

In our MTCs we assume that transformations are *model-to-model* meaning that they transform an input model into an output model, except for the last transformation which is *model-to-text*. Since the input of the chain is often a text file, the first transformation is typically a *text-to-model* parsing transformation, however we will not consider that kind of transformations and will assume that the chain starts with a parsed model.

A transformation is said to be *endogenous* if the input and output metamodels are the same and it is *exogenous* if the input and output metamodels are different. Both kinds of transformations can be involved in an ACG. For example the QGen chain starts with a series of endogenous transformations involving flattening of nested structures, resolving of references, and various refinements of the Simulink model while remaining within the same metamodel. Then an exogenous transformations translates the Simulink model to a so-called *code model* which is similar to the abstract syntax tree of source code. The code model is again refined by a series of endogenous transformations including expansion and optimisation of the model before the ultimate model-to-text printing transformation.

Given this scope and assumptions, in the next chapter we proceed to a review of existing work in the literature that is relevant to our context. Namely, we are interested in strategies and methodologies for the testing of model transformations and model transformation chains.

## 2.7 Conclusion

In this chapter we have set the stage for the research conducted in this thesis. Critical airborne software is subject to the rigorous constraints of the DO−178C certification standard. This makes their development extremely costly due to the extensive verification and testing activities that are required, in particular the reviews of source code and the low-level requirements-based testing. Manufacturers seek to reduce the cost of certification by using a *qualified* Automatic Code Generator (ACG) that provides enough confidence in the generated code allowing to eliminate part or all of its verification.

Qualifying an ACG is also a rigorous process defined by the DO−330 standard and arguably as costly as certification. It requires deploying a heavy requirements-driven process involving traceability and verification of all development artifacts. A significant portion of the cost of qualification is related to the extensive testing required on the ACG, including the development of a large number of carefully designed test cases with high criteria of requirements and source code coverage to satisfy. This lead us to focus this thesis on the investigation of testing techniques

and methodologies for ACGs which essentially consist of model transformation chains. As a result, the next chapter will be dedicated to a literature review of the current scientific advances on the testing of model transformations and model transformation chains.

# Chapter 3

# Literature Review on the Testing of Model Transformations and Code Generators

## Contents

## 3.1 Introduction

Having identified the testing of model-based code generators as the focus of this thesis, we present in this chapter a literature review covering existing works on the testing of Model Transformations (MTs) and Automatic Code Generators (ACGs).

Before we proceed with the literature review, it is important to acknowledge that testing is one way to verify the correctness of model transformations, but correctness can also be shown via formal methods relying on mathematical abstractions. So it is interesting to explain first the distinction between the two verification methods and justify why only testing is considered in our work.

**Formal verification** consists in using a mathematical model of the transformation to be verified and demonstrating the desired correctness property either for all possible inputs of the transformation (*i.e.* unbounded verification) or for a subset of the possible inputs (*i.e.* bounded verification). In both cases, the verification is *exhaustive* meaning that correctness is shown for *all* possible elements of the verification scope.

**Testing** consists in selecting particular instances (*i.e.* test models) from the set of possible inputs of the transformation under test and verifying that the transformation executes correctly for these instances. The verification is *non-exhaustive*, hence the underlying hypothesis is that if the test models are sufficiently varied and representative of the set of possible inputs, then the transformation is assumed to be correct for any instance of the set.

Compared to testing which validates a transformation using selected instances of the input domain (*i.e.* the set of all possible inputs), formal verification is in a sense stronger than verification by testing because the correctness is demonstrated for *all* instances of the verification scope and not only for *some* selected test instances. However often in practice the formal verification result can be compromised by implementation factors. This is because formal verification tools reason on the mathematical model of the transformation which may differ from the actual implementation of the transformation: the execution engine may include bugs or hidden divergences from the formal semantics of the language. Therefore the result of the formal verification is only valid as long as the execution engine is true to the language semantics and is bug free. In that regard testing has the advantage that it exercises the actual implementation and not a formal representation of it. The verification result is therefore undeniably valid (on the test instances) for the final software.

For this reason, in the industrial context of qualification, even though formal methods may be applied in certain cases, testing remains the preferred and most trusted verification method, and strict coverage criteria are required to ensure its thoroughness.

As a result, the first section of this chapter gives a brief overview of the formal verification of MTs while the rest of the chapter is dedicated to their testing.

## 3.2 Formal Verification of Model Transformations

Formal verification approaches typically consist in deriving a mathematical model of the relationship between the input and output domains of a transformation based on its semantics. Various mathematical models have been proposed differing in the underlying formalisms and the abstractions used [Anastasakis *et al.*, 2007b; Cabot *et al.*, 2010b; Büttner *et al.*, 2011; Büttner *et al.*, 2012b; Büttner *et al.*, 2012a]. For example several of these works rely on the notion of a *transformation model* [Bézivin *et al.*, 2006] which is composed of the union of the input and output metamodels of the transformation, with trace links relating input and output elements. The transformation model includes a set of constraints expressing the relationship between input and output elements according to the semantics of the transformation. Such a formal model is then submitted to a bounded or unbounded verification tool with a target property to be verified. Various properties can be verified including functional correctness, *i.e.* checking that a postcondition always holds on the output model under the assumption of a precondition on the input model, or other properties such as determinism or bijectivity.

Bounded verification uses tools such as Alloy [Alloy, accessed 2015] based on SAT[1] solvers and EMFtoCSP [Gonzalez *et al.*, 2012] based on constraint logic programming. Such tools can check if a property is always satisfied within a bounded scope and can provide counter examples if the property is violated. Roughly, the scope of the verification is defined by the maximum number of objects in the input and output models.

Unbounded verification employs tools such as *ocl2smt* [Soeken *et al.*, 2010] relying on automatic theorem provers, *i.e.* SMT[2] solvers, or interactive theorem provers such as HOL–OCL [Brucker and Wolff, 2008] which allows an assisted proof of correctness guided by the user.

[1] SAT: boolean satisfiability problem; finding an assignment of boolean variables that satisfies a given boolean formula.

[2] Satisfiability Modulo Theories

The SAT and SMT solvers that support formal verification often suffer from scalability issues when dealing with large formal models comprising many concepts and constraints. This hinders the application of formal analysis to large scale industrial use cases at the current stage. As for interactive proof approaches, they require a special kind of expertise in highly theoretical concepts, which is not widely available in the industry. Finally as explained in the introduction, the correctness result is only valid as long as the execution technology is compatible with the theoretical model used for the verification, which may not be the case in the event of implementation bugs or corner cases. The risk of discrepancy increases when the verification model undergoes transformations in the purpose of analysis [Leveque *et al.*, 2011], which may alter its semantics and jeopardise the applicability of the verification result to the actual executed software.

For all these reasons, formal verification is not yet popular in the industry, and testing remains the preferred verification method. Consequently, the rest of this chapter will be dedicated to the testing of MTs.

## 3.3 Model Transformation Testing Foundations

Testing a MT consists in (i) producing a set of input test models, (ii) executing the transformation over each model, and (iii) checking that the resulting model is correct with respect to a certain specification. If the transformation behaves correctly during testing, it is then assumed to behave correctly for all possible inputs. Evidently, this claim only holds if the set of test models is sufficiently rich and representative of all possible inputs, and it is highly important to ensure that the test set exhibits these properties. As a result, 3 activities are necessary in the testing of MTs [Baudry *et al.*, 2010]:

1. *Producing test models*: The first step in MT testing is producing test models to be used as input to the transformation under test. Test models must conform to the input metamodel and must satisfy any validity constraints associated with the metamodel. This is a challenging task given the complexity of metamodels and associated constraints which makes manual production tedious and error prone. Even automatic production is not straightforward as it can be both time and memory consuming for complex metamodels.
2. *Defining test adequacy criteria*: It is not possible to test a MT with all possible inputs. A test adequacy criterion [Zhu *et al.*, 1997] helps select a set of test models, *i.e.* a subset of all possible inputs, which is considered sufficient for testing. Such a criterion should ensure that the test set is sufficiently rich and representative of the input domain, or that it exercises different execution

scenarios of the transformation. A test set that satisfies the chosen criterion is said to be *adequate*. Various criteria may be used. For example, given a MT consisting of a set of transformation rules, a simple *rule coverage criterion* may be used stating that each rule should be activated at least once by the test set. More complex criteria assess the effectiveness of the test set at detecting errors. For example the so-called *mutation criterion* consists in introducing intentional errors in the transformation under test and assessing whether the test set is able to detect these errors [Mottu *et al.*, 2006]. The assumption here is that if a test set is capable of detecting intentional errors, then it is likely to detect non-intentional errors as well and is therefore an effective test set. A test adequacy criterion can be used either to assess the quality of an existing test set, or alternatively to drive the generation of a test set which would be adequate by construction.

3. *Test oracles*: The third challenge in MT testing is determining the verdict of tests, *i.e.* assigning a PASS or FAIL verdict to each test execution. Test oracles can take various forms. For example, a simple form of oracles consists in comparing the test output with an expected output, manually validated as being correct. Other more complex oracles may check correctness properties expected to hold on the output such as the existence or the absence of certain model elements in the result.

The body of work on MT testing is organised around these three challenges, and our presentation of existing works is organised accordingly. First we present works addressing the definition of test adequacy criteria in Section 3.4, then we present works on test model generation in Section 3.5, and finally works addressing test oracles in Section 3.6. Some works addressing two or more of the challenges jointly may be mentioned in more than one section, each time focusing on a different part of their contribution.

Before moving to the presentation of existing works, we find it relevant to define common general concepts that can be found in several works, often with different names. We thus present the following two general schemes that we extracted based on existing works.

### 3.3.1 General Scheme for Test Adequacy Criteria and Test Generation

Several works addressing the first two aspects of test adequacy criteria and test model generation implicitly share a common underlying process illustrated in Figure 3.1. On the left side of the figure, the first step is to define a test adequacy criterion as a procedure that takes as input a variety of sources of information (or

a combination of them). A test adequacy criterion is said to be *black-box* if it is based only on information from the specification of the transformation under test, and it is said to be *white-box* if it considers information from the implementation. The result of applying a test adequacy criterion is a set of so-called *test requirements* that expresses concretely the constraints that a test set should satisfy in order to be *adequate*.

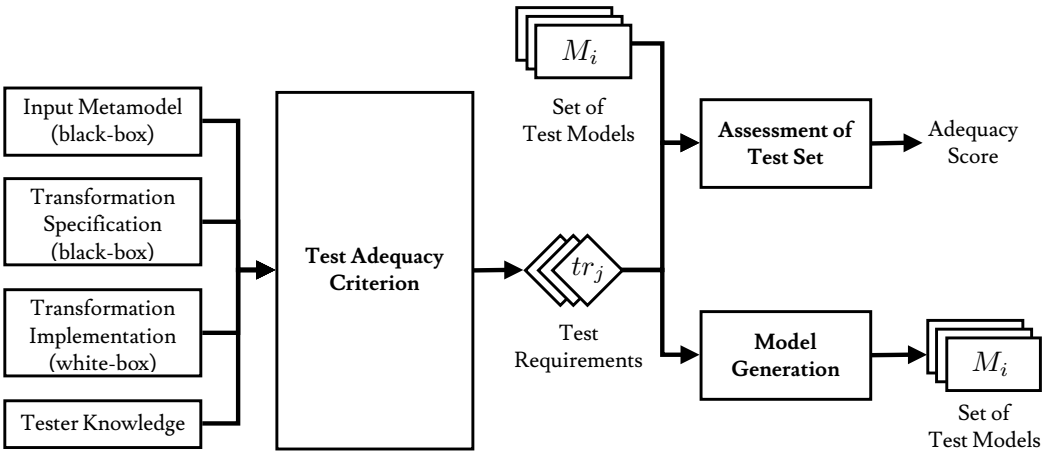



Figure 3.1: General scheme for test adequacy criteria and test generation

A test requirement is a *constraint* over input models of the transformation which must be satisfied at least once during testing. Test requirements express combinations of objects and attribute values that are required to exist in the set of test models. If a test set contains all such combinations, then it is considered *adequate* with respect to the test adequacy criteria. It is not necessary to have a *one-to-one* relationship between test requirements and test models. One test model could satisfy multiple test requirements, and one test requirement could be satisfied by multiple test models, as long as each test requirement is satisfied at least once.

Test requirements can be expressed in various forms. In some approaches they take the form of so-called *model fragments* (detailed in Section 3.4.2) which express combinations of objects and specific ranges for scalar numeric or string attributes. In other approaches test requirements are *OCL constraints* that express more complex patterns that are required to exist in test models (detailed in Sections 3.4.3 and 3.4.4).

Once a set of test requirements is obtained it can be used in two ways. The first, depicted by the upper branch of the process in Figure 3.1, is to consider an existing set of test models and assess whether each test requirement is satisfied by at least one test model. An *adequacy score* is then computed as the ratio of satisfied test requirements over the total number of test requirements. A test set is adequate

with respect to the criteria if all test requirements are satisfied. The second use of test requirements, depicted in the lower branch of the process in Figure 3.1, is to guide the automatic generation of test models. Several techniques exist and will be detailed in Section 3.5. Roughly model generation techniques take as input a test requirement and automatically generate a test model that satisfies it. Thus the resulting test set is adequate by construction.

The general process of Figure 3.1 for the assessment or the generation of test sets applies to most test adequacy criteria except for the *mutation criterion* which is detailed in Section 3.4.1. The reason is that the mutation criterion assesses a test set by evaluating its ability to detect errors intentionally introduced in the transformation under test. This cannot be easily expressed as a set of test requirements and therefore the mutation criterion follows a fundamentally different process that will be detailed in Section 3.4.1.

### 3.3.2 General Scheme for Test Oracles

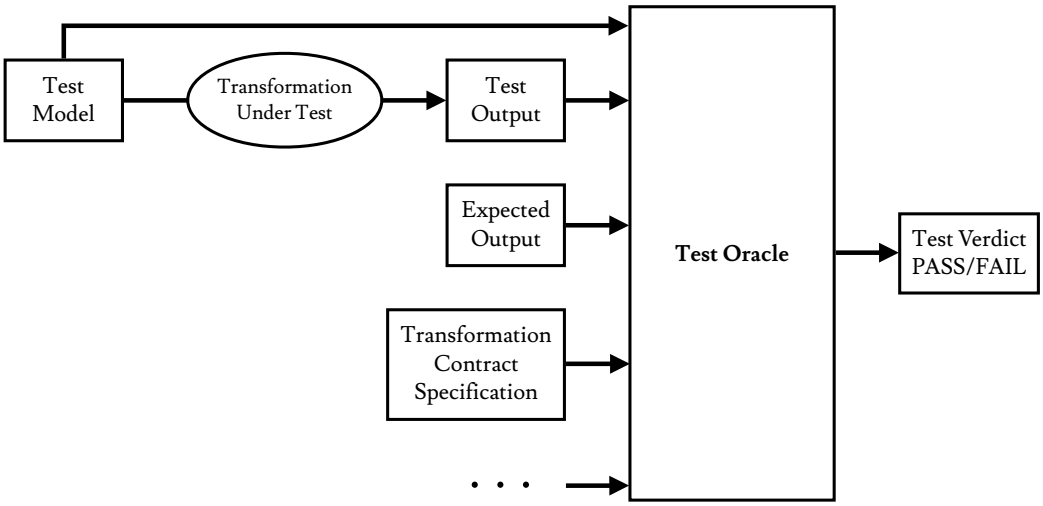



Figure 3.2: General scheme for test oracles

With the aspects of test adequacy criteria and test model production covered by the previous scheme, the last aspect is the test oracles which validate the output of a test execution. Approaches addressing test oracles follow the general scheme of Figure 3.2 based on [Mottu *et al.*, 2008] which defines and compares several forms of test oracles for MT. Each test model is given as input to the transformation under test, and the role of a test oracle is to decide whether the test output is valid or not, yielding the verdict of the test accordingly: PASS or FAIL. As discussed in [Mottu *et al.*, 2008] several kinds of oracles can be used. Oracles may be simple, consisting of a comparison with an expected output manually validated to be correct, or rely on

a contract-based specification of the transformation which can be evaluated over the pair (Test Model, Test Output) to assess whether the result complies with the specification. The most prominent approaches of test oracles will be detailed in Section 3.6.

Having laid down a general terminology for test adequacy criteria, test model generation and test oracles, we now proceed with the review of existing works on MT testing starting with approaches that propose test adequacy criteria.

## 3.4 Test Adequacy Criteria

### 3.4.1 Mutation Criterion

As mentioned previously in Section 3.3.1, the mutation criterion does not adhere to the general scheme of test adequacy criteria presented earlier. However we start with this criterion because it is subsequently used in several other approaches as a way to assess the quality of other test adequacy criteria.

#### Basic Principle

*Mutation analysis* is a testing technique originally proposed for general purpose software as a way to estimate the quality of a test set in terms of its *fault revealing power*, *i.e.* its ability to detect faults. In [Mottu *et al.*, 2006] the authors adapt *mutation analysis* to the testing of MTs. The approach consists in inserting intentional errors in the transformation under test to create faulty versions of the transformation called *mutants*. Each mutant is the original transformation modified by the injection of a single fault. Then each test model is given as input to the original transformation and to each mutant and the outputs are compared. A mutant is said to be *killed* if for some test the output of the mutant is different than the output of the original transformation, *i.e.* the test set was able to reveal the fault in the mutant. Conversely, a mutant is said to be *alive* if no tests reveal a difference between its output and the output of the original transformation, *i.e.* the test set was not able to reveal the fault in the mutant or the mutant is equivalent to the original transformation. The more mutants are killed by the test set, the better the test set is at revealing faults. It is then assumed that if the test set is able to detect intentional faults, then it is likely to detect real involuntary errors. A *mutation score* is computed for the test set as the proportion of revealed mutants among the non-equivalent mutants.

### Mutation Operators

For the generation of mutants, [Mottu *et al.*, 2006] proposes a set of *mutation operators* dedicated to MTs. Mutation operators represent typical errors that MT developers may introduce. For example, the *Collection Filtering Change with Deletion* (CFCD) operator deletes a filter on a collection, thus modeling the situation where a developer forgets to add a filter on a collection before using it in the transformation. The fact that mutation operators model real developer errors reinforces the pertinence of mutation analysis as an assessment of a test set's ability of detecting involuntary errors. The proposed mutation operators are language independent making the analysis applicable to any MT language. A precise modelling of mutation operators was later proposed in [Aranega *et al.*, 2014] which allowed to extend mutation analysis with a method to generate new test models to improve the mutation score. We will detail this extension later in Section 3.5.3 when we discuss model generation techniques.

### Mutation Analysis as a Metric

Beyond assessing a particular test set, mutation analysis is often used to assess other test adequacy criteria and automated test generation approaches associated with them. A criterion is assessed by evaluating the mutation scores of test sets generated based on it. In this way different test adequacy criteria can be compared with each other as will be mentioned in the following sections.

### Discussion

Despite its scientific value, mutation analysis cannot be used in the context of *qualification* as a main method for the monitoring of testing quality. In a qualification process tests are developed based on the requirements of the software, *i.e.* high-level Tool Operational Requirements (TOR), Tool Requirements (TR) and Tool Low-Level Requirements (TLLR). The main criteria for assessing the quality of testing is a *coverage* criteria: all requirements should be covered by tests, logical formulas in requirements should have MC/DC coverage, and source code should have MC/DC coverage. Even if mutation analysis is used to provide more confidence in the test set, it may not replace the coverage-based criteria required by the standard.

However, it would be interesting to compare the quality of coverage oriented testing in qualification with mutation driven testing in terms of the fault revealing power. Even though we have not conducted such a study, experimental data in a coverage oriented approach in [Mottu *et al.*, 2012] tends to show that a coverage-based approach based on a detailed specification of the software under test results in a hight fault revealing power. This confirms that the thoroughness of coverage

criteria required in qualification is necessary to ensure the quality of testing. This aspect will be further detailed once we present coverage-based criteria in the next section.

### 3.4.2 Input Metamodel Coverage Criteria

Input domain coverage is a classical criteria in the testing of general software. In [Fleurey *et al.*, 2004; Brottier *et al.*, 2006; Fleurey *et al.*, 2007] the authors define several coverage criteria for the input metamodel of a model transformation. These approach adhere to the general scheme of test adequacy criteria that we presented in Section 3.3.1. In that scheme, a test adequacy criteria is a procedure that takes as input information from various sources (*e.g.* the input metamodel, the MT specification *etc.*) and generates a set of test requirements to be satisfied. The set of test requirements is then used either to assess the quality of an existing test set or to drive the generation of a test set from scratch.

In all the approaches detailed next, the main idea is to ensure that all elements of the metamodel are instantiated at least once in the test set, and that there is a sufficient variety in the way objects are related and in the values assign to their attributes in the test models.

#### Partitioning

The approaches start with a partition analysis inspired from the *category-partition* method of testing classical programs [Ostrand and Balcer, 1988]. The method consists in dividing the input domain of a program into non-overlapping subsets called *equivalence classes*. The set of equivalence classes is called a *partition*. A test set can then be built by selecting one test datum from each equivalence class. The assumption is that if the program executes correctly for the selected datum, it should also execute correctly for any other instance of the equivalence class. This technique is first adapted to metamodels in [Fleurey *et al.*, 2004] where partitioning is applied to scalar attributes of metaclasses and for reference multiplicities. For example an integer attribute *attr* of a metaclass *C* yields a partition of 3 equivalence classes: $\{\{<0\},\{0\},\{>0\}\}$. Similarly, references yield partitions in terms of the number of objects contained in the reference. This partitioning is arbitrary and may not be relevant for the transformation under test. For this reason a better suited *knowledge-based partitioning* is also proposed which relies on singular values of particular significance to the transformation. Such values are either provided by the tester or automatically extracted from literals in the specification of the transformation (*e.g.* its pre- and post-conditions). For example if the precondition compares

*attr* with a literal value of 3, then the following partition is more relevant for the transformation: $\{\{<3\},\{3\},\{>3\}\}$.

#### Model Fragments

Considering each partition independently on its own is not sufficient. This is why after partition analysis, the identified equivalence classes are selected and combined into so-called *model fragments*. A model fragment is a set of equivalence classes selected from the partitions identified earlier. A model fragment indicates that at least one of the test models must contain objects with attribute values and references that are part of the selected equivalence classes. Each model fragment is therefore a constraint that must be satisfied by at least one test model: it represents a test requirement in the general scheme that we proposed.

#### Black-box Adequacy Criteria

In [Fleurey *et al.*, 2007] model fragments are generated automatically according to different strategies. Each strategy combines equivalence classes in a different way and constitutes a different test adequacy criterion. All the proposed criteria are black-box because they are only based on the input metamodel and not on the implementation of the MT.

The *AllRanges* criteria is the simplest and requires simply that each equivalence class be satisfied once, yielding one model fragment for each equivalence class. Another simple criteria, *AllPartitions*, requires all equivalence classes of each partition to be satisfied in the same model, yielding one model fragments per partition such that all equivalence classes of the partition are selected in the model fragment. This ensures the co-existence of equivalence classes of the same partition in the same model, but does not guarantee the interaction of different partitions. Eight other more complex criteria are proposed to combine equivalence classes of different partitions derived from the same metaclass or from different metaclasses, taking into account inheritance relationships.

#### White-box Footprint-driven Criterion

In contrast with the above black-box criteria which are only based on the input metamodel, in [Mottu *et al.*, 2012] a different coverage criterion is proposed based on a white-box static analysis of the implementation of the transformation. A *footprint* of the transformation is extracted from its implementation and identifies metamodel features that are closely related because they are used jointly within the same operation of the transformation. The idea is then to build model fragments by combining partitions of closely related features based on the footprint.

**Assessment of a Test Set**

Once the set of model fragments is generated based on a black-box or a white-box coverage criteria, it can be used either to assess the quality of an existing test set, or to generate a new test set from scratch. We only discuss the former aspect of assessment at this stage while the latter model generation for Section 3.5.

The assessment of a test set is performed by the Metamodel Coverage Checker (MMCC) tool discussed in [Fleurey *et al.*, 2007]. In fact the tool implements several aspects of the approach: (1) it performs the partitioning of a metamodel, (2) it generates model fragments according to the adequacy criteria discussed above, and (3) it assesses whether each model fragments is satisfied at least once in a set of test models.

**Comparison of Coverage Criteria**

In [Sen *et al.*, 2009] automatic test generation is performed guided by the *All-Ranges* and *AllPartitions* criteria and mutation analysis is used to assess the quality of the resulting test sets in terms of their fault revealing power. Both criteria were found to yield relatively close mutation scores on average ( 82%), both higher than a random unguided test generation approach. In [Mottu *et al.*, 2012] the white-box footprint-based criterion was compared to the previous two criteria and found to yield significantly higher mutation scores reaching 98%. This indicates that a partitioning and coverage criteria combined with detailed information of the transformation under test (*e.g.* a white-box footprint) can yield a high quality test set.

**Discussion**

This body of work explores the notion of metamodel coverage to a large extent, providing several criteria to define metamodel coverage and the means to achieve it. Such criteria can be very useful in the industrial context of qualification because they focus on the notion of partitioning and coverage which is central to the testing philosophy of qualification processes. For example the qualification standard explicitly requires testing boundary values of integer parameters which is what the presented approaches do. Such approaches are therefore good candidates to drive the testing of qualified software.

In particular, the white-box approach of [Mottu *et al.*, 2012] showed a 98% mutation score, a higher score than black-box approaches. However in qualification the production of tests is based on requirements, and thus is necessarily black-box. Does this mean that we cannot benefit from the white-box criteria in qualification?

In fact we believe that the high score reached by the white-box approach is due to the combination of two factors: (1) an input domain partitioning method and (2) detailed knowledge of the software under test driving the combination of partitions. Qualification demands a high level of detail in the specification of requirements, a level close to that of the implementation. Therefore we believe that the knowledge in (2) can be extracted from the requirements instead of the implementation, yielding a black-box approach compatible with qualification and with a high fault revealing power.

In line with that last idea, we present in the next section criteria based on the specification of a MT.

### 3.4.3 Specification-based Criteria

#### Specification Language

In [Guerra, 2012; Guerra and Soeken, 2015] the authors propose to use the specification of the MT to guide its testing. The approach deals jointly with all three aspects of testing: test adequacy criteria, test model generation and test oracles; however we focus on the first aspect at this stage. This approach also adheres to the general scheme presented in Section 3.3.1 whereby a test adequacy criteria generates a set of test requirements that should be satisfied by the test set. In this case, the input of the test adequacy criteria is the specification of the MT.

The specification language in this approach is PAMOMO [Guerra *et al.*, 2010], a formal pattern-based declarative language. The specification is composed of three kinds of elements:

1. *Preconditions* that all input models should satisfy.

2. *Postconditions* that all output models should satisfy.

3. *Invariants* that express a different kind of postconditions that should be satisfied jointly by the input model and the output model after the transformation. Invariants are composed of a *source pattern* and a *target pattern* and express properties of the form: *if the source pattern appears in the input model then the target pattern should exist (or should not exist) in the output model*.

All the above specification elements are represented as graph patterns optionally accompanied with OCL constraints.

### Specification Coverage Criteria

Since postconditions only concern the output model and preconditions must be satisfied by all input models, only invariants are considered for defining the specification coverage criteria. The approach proposes to extract from each invariant the source pattern concerning the source model and translate it to an OCL constraint requiring the existence of the source pattern. This OCL constraint is called a *property* and the result of this first step is a set of properties $e_i$. Then seven levels of coverage criteria are proposed, each criteria combining the extracted properties in a different way.

We illustrate a few of the specification coverage criteria in Table 3.1 where we list for each criteria the resulting set of test requirements, assuming there are 3 properties in the specification. The first criteria, *property coverage*, is the simplest because it requires that each property should be satisfied at least once by a test model. In contrast, *t-way coverage* requires that all possible combinations of $t$ properties should be covered jointly. This ensures that the interaction between different parts of the specification is tested and verified to be correctly implemented. The so-called *closed* versions of the criteria complement the set of test requirements with the negation of each test requirement.

| **property** | **closed property** | **2-way** | **closed 2-way** | ... |
|---|---|---|---|---|
| $e_1$ | $e_1$ | $e_1$ and $e_2$ | $e_1$ and $e_2$ | |
| $e_2$ | $e_2$ | $e_1$ and $e_3$ | $e_1$ and $e_3$ | |
| $e_3$ | $e_3$ | $e_2$ and $e_3$ | $e_2$ and $e_3$ | |
| | not $e_1$ | | not $e_1$ | |
| | not $e_2$ | | not $e_2$ | |
| | not $e_3$ | | not $e_3$ | |

Table 3.1: Test requirements for different specification coverage criteria applied to 3 properties (excerpt from [Guerra, 2012] )

### Assessment and Comparison of Criteria

As customary, mutation analysis is performed in [Guerra and Soeken, 2015] to compare the quality of test sets generated with the above criteria: property coverage, closed property coverage, 2-way coverage and closed 2-way coverage. Since these criteria are specification-based, a particular aspect of this assessment is that it considers how the degree of completeness of the specification affects the quality of the resulting test sets. It is found that when the specification is incomplete, *closed* versions of the criteria outperform the other versions, but for complete specifications, the criteria were found to yield test sets of identical quality. This highlights

the importance of ensuring a complete and detailed specification when adopting a specification-based testing approach.

Additionally, the highest attained mutation score across all criteria is 84%, however this metric is to be considered with care. The mutation analysis set up for this evaluation is slightly different than the classical mutation analysis presented in Section 3.4.1. In addition to the specification-based adequacy criteria, this approach also proposes specification-based oracles that will be detailed in Section 3.6.2. As a result the mutation analysis conducted in this work uses these specification-based oracles instead of the classical comparison with the non-mutated transformation. Therefore the resulting metric assesses the quality of both the test adequacy criteria and the test oracles simultaneously. This prevents a comparison with the scores of other test adequacy criteria of the literature such as the ones based on input domain coverage that we presented in Section 3.4.2. In fact, the authors explain that the score of 84%, which is relatively low compared to the 98% reached with the white-box footprint-based metamodel coverage criteria of Section 3.4.2, is due to the inability of the oracles to detect mutants rather than to the quality of the test set itself. This aspect will be further detailed when we discuss test oracles in Section 3.6.2.

**Discussion**

This kind of approach seems very interesting from the industrial qualification viewpoint because it is based on the specification which is the underlying philosophy of requirements-based testing in qualification. The approach seems to be applicable beyond the PAMOMO specification language since the proposed criteria could be applied to any form of specification organised as a collection of rules or invariants. The combinatorial nature of the proposed criteria is interesting from a qualification perspective because it allows to exercise the interaction of different parts of the specification jointly, thus pushing the coverage of requirements to a further extent.

The quality of test sets resulting from such a specification-based approach is evidently sensitive to the level of detail and completeness of the specification. Given the multi-level requirements scheme in qualification, *i.e.* high-level TORs, intermediate TRs and low-level TLLRs, we expect TLLRs to have the necessary completeness and level of detail necessary to ensure the quality of testing in this approach.

Finally, the choice of OCL to express *test requirements* in this approach is more expressive than the model fragments used in the approaches of Section 3.4.2. It allows in particular expressing negative constraints prohibiting the existence of certain elements which was not possible with model fragments. In addition we believe OCL

is a good candidate for describing *test cases* in a qualification process because its expressiveness would allow the combination of automatic and manual specification of test cases. Since test cases are expected to characterise both the input and the output of a test, OCL test requirements can represent the characterisation of the input within a test case.

In the next section we discuss approaches that propose coverage criteria based on OCL expressions that occur either in the specification (black-box) or in the implementation (white-box).

### 3.4.4 OCL Structural Coverage Criteria

#### General Principle

In [González and Cabot, 2012; González and Cabot, 2014] the authors propose to analyse OCL expressions to identify the various ways in which an OCL expression may be executed and ensure that testing covers all possible executions. For example, the expression `obj.ref->select(e|e.someAttr)` where `ref` is a reference to objects of type `T` could be tested under several situations:

1. There are no objects of type `T`:
   ```
   T::allInstances()->isEmpty()
   ```
2. There are objects of type `T` but none have `someAttr` set to `true`:
   ```
   T::allInstances()->select(e|e.someAttr)->isEmpty()
   ```
3. There are objects of type `T` and some of them have `someAttr` set to `true`:
   ```
   T::allInstances()->select(e|e.someAttr)->notEmpty()
   ```
4. ...

Each of the listed constraints is a test requirement that should be satisfied at least once in the test set.

Two different analyses are proposed in [González and Cabot, 2012] and [González and Cabot, 2014] for the generation of test requirements. In [González and Cabot, 2012] the analysis is inspired from traditional control flow and data flow coverage strategies [Myers *et al.*, 2011] involving the construction and traversal of a *dependency graph* constructed for OCL expressions (similar to a control flow graph for traditional programs). Classical strategies such as *Condition Coverage* and *Multiple-Condition Coverage* are used in the traversal of the graph to generate a set of test requirements.

In [González and Cabot, 2014] an ad-hoc analysis of OCL constraints is proposed to partition the input domain of the transformation based on a similar un-

derlying principle: enumerating the different ways in which an OCL constraint can be satisfied, each corresponding to a different region of the input domain. Test requirements are then generated based on different strategies to combine regions.

#### White-box/Black-box

In [González and Cabot, 2012] the approach is white-box because the analysed OCL expressions are taken from the implementation of the transformation whereas in [González and Cabot, 2014] OCL metamodel invariants are analysed making the approach black-box. However both approaches may be black-box or white-box depending on the origin of the analysed OCL expressions.

#### Discussion

From a qualification viewpoint only a black-box usage of these OCL analyses would be useful since testing should be solely requirements-based. Moreover the *Condition Coverage* and *Multiple-Condition Coverage* (MCC) criteria discussed in these approaches are closely related to the *Modified Condition/Decision Coverage* (MC/DC) criteria required in qualification.

The MC/DC criteria that we presented in Section 2.2.2 is in fact a simplified version of the MCC criteria (also known as *exhaustive testing*) [Hayhurst *et al.*, 2001]. For a decision composed of $N$ independent conditions, MCC requires $2^N$ tests which is very difficult to achieve for large decisions. This lead to the adoption of MC/DC which requires only $N + 1$ and still ensures a high combination of conditions.

In qualification, MC/DC is required for source code, but also in the definition of test cases based on requirements expressed as boolean formulas. The OCL coverage approaches that we discussed in this section are therefore highly relevant if requirements involve OCL expressions since they would allow to automatically ensure the level of coverage required by the standard.

This concludes the review of test adequacy criteria proposed in the literature. We now move to the review of test model generation approaches which are typically guided by the test requirements produced based on test adequacy criteria.

## 3.5 Test Model Generation

In this section we give an overview of test model generation approaches of the literature. The first two approaches in Section 3.5.1 and Section 3.5.2 adhere to the general scheme for model generation that we described in Section 3.3.1. According

to that scheme a model generation approach takes as input a set of test requirements, and generates a set of test models that satisfy each test requirement at least once. The third approach of Section 3.5.3 does not adhere to that scheme because it is based on the mutation criteria.

### 3.5.1 Ad-hoc Model Generation Based on Model Fragments

One of the first model generation algorithms was proposed in [Brottier *et al.*, 2006] to generate models satisfying test requirements expressed as model fragments. Given a set of model fragments derived from a metamodel coverage criteria (see Section 3.4.2), the algorithm generates a set of models that satisfy the model fragments. Several configuration parameters control the size and characteristics of the generated models. For example the maximum number of objects in each model and the minimum number of model fragments to be included in the same model can be specified.

However, this algorithm does not take into account metamodel validity constraints and transformation preconditions. There is no guarantee that models produced by the algorithm will satisfy both the metamodel constraints and the preconditions, and models violating these constraints cannot be used for testing. If such violating models are encountered, the tester must manually investigate the model fragments that produced them to determine if the fragments are in conflict with validity constraints and preconditions and should be discarded. This manual activity is highly inconvenient. Furthermore, another limitation of this approach is the limited expressiveness of model fragments when it comes to generating models to satisfy constraints manually provided by the tester. Model fragments cannot express arbitrary constraints (such as the absence of object patterns) which limits the scope of testing knowledge that the tester can manually introduce. Both limitations led to the wide spread adoption of model generators based on Constraint Satisfaction Problem (CSP) solvers which are presented next.

### 3.5.2 Model Generation with Constraints Satisfaction Problem Solving

Several approaches propose to describe model generation as a Constraint Satisfaction Problem (CSP) and use SAT solving and SMT[3] solving. The main advantage of such approaches is the ability to *include any needed constraints to the model generation* such that generated models are guaranteed to satisfy all constraints simultaneously.

[3]satisfiability modulo theories: SAT with predicates on non-boolean variables subject to a set of rules called a theory, *e.g.* linear arithmetics for real numbers, arrays, lists *etc.*

This allows to drive the model generation with knowledge combined from various sources:

– Metamodel structure and semantics (*i.e.* multiplicity constraints, containment references constraints *etc.*)

– Metamodel validity constraints (OCL)

– Transformation preconditions (OCL)

– Test requirements automatically generated based on a test adequacy criteria (*e.g.* model fragments for metamodel coverage in Section 3.4.2, OCL test requirements for specification coverage in Section 3.4.3 and Section 3.4.4)

– Arbitrary testing constraints provided manually by the tester

Several such approaches exist relying on various CSP paradigms and technologies. The underlying principle is to encode the structure of the metamodel in the CSP paradigm. Then all constraints over the metamodel are transformed into equivalent constraints over the CSP structure. This includes both metamodel validity constraints and transformation preconditions. Finally, each test requirement is translated to a constraint in the CSP and the solver is invoked to find an instantiation of the structure that satisfies all constraints: the test requirement as well as the validity constraints and transformation precondition. If a solution is found, it is translated from the CSP paradigm back to a model. If a solution is not found, then either there is a conflict between the test requirement and the other constraints making their combination unsatisfiable, or the bounds (if any) chosen for the solver to limit the search space are too restrictive.

Several such tools were proposed in the literature based on various technologies. The following is a non exhaustive list:

1. Pramana [Sen *et al.*, 2008] based on Alloy [Alloy, accessed 2015] and SAT solvers.
2. EMFtoCSP [Gonzalez *et al.*, 2012] (formerly UMLtoCSP) based on the ECL$^{i}$PS$^{e}$ Constraint Programming System[4] [ECL, accessed 2015].
3. UML2Alloy [Anastasakis *et al.*, 2007a] based on Alloy and SAT solvers.
4. Snapshot generation in the UML-based Specification Environment (USE) [Gogolla *et al.*, 2005]
5. ocl2smt model finder based on SMT solvers [Soeken *et al.*, 2010].

In the context of our work, we have mostly investigated the first approach based on the Alloy language and tool. This approach has been applied in several works,

[4]not to be confused with the Eclipse IDE platform

each combining testing knowledge from different sources and formalisms: [Sen *et al.*, 2008; Sen *et al.*, 2009; Sen *et al.*, 2012; Mottu *et al.*, 2012]. Testing knowledge is combined into a CSP expressed in *Alloy* [Alloy, accessed 2015] and the *Alloy Analyzer* is invoked to find model instances satisfying the various constraints simultaneously. Next, we outline the principles of constraints satisfaction in Alloy.

**Bounded Model Finding in Alloy**

Alloy is a language allowing to describe a *model* as a core *structure* and associated *constraints* which implicitly define a set of possible instantiations of the structure (*i.e.* those satisfying the constraints). Then the Alloy Analyzer can be used to enumerate instances of the structure satisfying the constraints, which is referred to as *model finding*. At a first glance the Alloy language has a semantics close to an object oriented language. An Alloy model is defined as a collection of *signatures* (similar to classes) with *extension* relationships (similar to inheritance) between them, where each signature can have *relations* (similar to inter-class references) to other signatures with multiplicity constraints. An Alloy model is to a certain extent similar to a metamodel. However at a lower abstraction level the semantics is in fact set theoretical. Signatures are sets, and extension relationships indicate the inclusion of one signature/set in another. A relation between two signatures is in fact a binary relation between the corresponding sets, and arbitrary n-ary relations can be defined in Alloy. Based on this core structure of signatures and relations, an Alloy model also includes the definition of *facts* which are structural constraints that must always be true. The language used to specify facts is based on first-order logic, with the addition of a (reflexive) transitive closure operator.

Given a model composed of a structure and a set of constraints, the Alloy Analyzer can find instances of the structure that satisfy all constraints. The analysis operates by transforming the Alloy model into a corresponding boolean logic formula and then invoking a SAT-solver to find a solution to the formula which is then translated back to an instance of Alloy model. In order to ensure the finiteness of the analysis, Alloy performs model finding within a restricted *user-defined scope*. The user specifies bounds to the number of elements in each signature of the model.

**Alloy-based Test Generation**

Test generation is performed by translating the problem to an Alloy model. The metamodel structure is translated to a set of Alloy signatures and relations, with facts enforcing metamodel semantics such as containment references (*i.e.* an object can only exist in one containment relationship and there cannot be containment cy-

cles). Metamodel validity constraints and transformation preconditions are translated *manually* from OCL to Alloy facts. Automatic translation of OCL to Alloy is possible but was not conducted in these approaches because it raises a number of challenges. The translation was however tackled in other works [Anastasakis *et al.*, 2007a; Anastasakis *et al.*, 2010]. Then to this core of necessary constraints, we can add testing knowledge to guide the model finding.

In [Sen *et al.*, 2009; Mottu *et al.*, 2012] testing knowledge is automatically provided in the form of model fragments derived from metamodel coverage criteria or from transformation footprinting as detailed in Section 3.4.2. Model fragments are translated to Alloy constraints and incorporated in the CSP. For each model fragment, the Alloy Analyzer is invoked to find a model that satisfies the model fragment and all other metamodel and precondition constraints. In [Sen *et al.*, 2008] testing knowledge is added manually. The tester describes patterns of objects that he deems interesting to test directly in the form of Alloy constraints to drive the model finding. In [Sen *et al.*, 2012] a radically different approach is proposed for tester's manual input of testing knowledge. The idea is that instead of designing full test models, the tester can specify only the interesting part of the test model in the form of a *partial model* which is less tedious to construct. Each partial model is translated to an Alloy constraint, and the analyzer completes the partial model into a full test model satisfying also validity and precondition constraints. Using mutation analysis, the authors go on to show that tests based on partial models are sufficient and comparatively effective to tests developed fully manually. This semi-automatic approach is made possible by the versatility of CSP model generation which can incorporate arbitrary constraints in the model finding.

#### Other Approaches

The specification-based criteria approach of PAMOMO (see Section 3.4.3) has used EMFtoCSP[5] with SAT solving at first in [Guerra, 2012] and then moved to ocl2smt with SMT solving in [Guerra and Soeken, 2015]. In both approaches PAMOMO preconditions are compiled to OCL, and combined with OCL test requirements automatically generated based on a specification coverage criteria. The resulting OCL constraint is given as input to EMFtoCSP or ocl2smt to guide the model finding.

#### Bounds Issues

SAT-based model finders operate within bounds on the size of the considered models so that the search always terminates. When a SAT-based model finder cannot find a model instance satisfying all the given constraints, it is either because the

[5] UMLtoCSP at the time

set of constraints is unsatisfiable, or because the chosen bounds are too restrictive. In the former situation it means that the test requirement chosen to drive the model generation contradicts metamodel constraints or transformation preconditions and should be discarded. In the latter situation, more appropriate bounds should be used. Determining which situation is the culprit is not an easy task despite the feedback given by tools such as Alloy[6]. Moreover, finding appropriate bounds is not straightforward since they must be minimal (to avoid a very large search space) and compatible with the metamodel structure.

#### Discussion

The main benefit of the CSP-based model generation approach is its flexibility. Testing knowledge can be incorporated from various sources while always honoring metamodel validity constraints and transformation preconditions. This is a major advantage over the ad-hoc approach in Section 3.5.1 which did not take into account validity constraints and preconditions. This flexibility is made possible by the expressiveness of CSP languages (typically first-order logic with additional features) which allows a large spectrum of testing knowledge to be incorporated in the model generation. However, with bounded SAT-based solvers finding appropriate bounds is still a manual and non-straightforward task. Despite the limitations, these model generation approaches would be interesting in a qualification process to reduce the cost of manually producing the large quantity of test models needed to achieve the required coverage criteria.

### 3.5.3 Semi-Automatic Model Generation based on Mutation Analysis

In [Aranega *et al.*, 2014] a significant extension to the mutation analysis presented in Section 3.4.1 is proposed. While classical mutation analysis only assesses the mutation score of a test set, this work proposes a method to create new test models to improve a test set's mutation score if it is deemed insufficient.

#### Improved Mutation Analysis

As in the classical analysis, mutation operators are applied to the transformation under test to create mutants which differ from the original transformation by the introduction of a single fault. However in this approach, a precise modelling of the mutation operators is defined in a transformation-language-independent manner based on the metamodels manipulated by the transformation. Then, a sophisticated traceability model is introduced to keep track of the analysis process: each

[6] Alloy provides a so-called minimal UNSAT core which is a minimal, but still often large and tedious to investigate, subset of unsatisfiable constraints

test model is traced to the mutants it manages to kill and to the mutation operators that these mutants resulted from. Both mutation operator modelling and the mutation traceability model serve the generation of new test models in the next step.

#### Generation of New Test Models

After the first analysis step, the remaining *alive* mutants are considered, and an approach is proposed for the semi-automatic production of new test models targeting the remaining mutants. A set of patterns and heuristics based on the traceability model are proposed. Patterns represent known situations preventing mutants from being killed. Each pattern also proposes recommendations on how an existing model can be modified to potentially kill the mutant. When known patterns are detected in the analysis results, the associated recommendations can sometimes be applied *automatically* to create new test models. Otherwise, analysis indications and modification suggestions are reported to the tester to assist him in *manually* creating a new test model, hence the semi-automatic nature of the process. The new test models allow to kill the remaining alive mutants and thus increase the fault revealing power of the test set.

#### Discussion

This approach pushes mutation analysis beyond the simple assessment of test sets towards the generation of new test models. However, the approach relies on an existing test set and cannot be used on its own to create a test set from scratch. Nonetheless, the method can be easily combined with other automatic model generation approaches as a way to improve a test set initially produced with another automatic method.

However, as previously discussed in Section 3.4.1, in a qualification process the criteria required for testing are coverage-based and mutation-based approaches can only be complementary but may not be used as a main criteria.

## 3.6 Test Oracles

The purpose of a test oracle is to validate the output model resulting from a test execution and determine the outcome of the test: PASS or FAIL. As explained in the general scheme of test oracles in Section 3.3.2, a test oracle may be based on various information and [Mottu *et al.*, 2008] details and compares several kinds of oracles. We will highlight the most prominent ones. The first test oracle is the

classical one based on manually inspected expected results. The second test oracle is based on a contract specification.

### 3.6.1 Manually Validated Expected Results

The simplest form of test oracles consists of comparing the *actual model* generated by the test execution with an *expected model* known to be correct. Therefore one expected model per test is necessary. The expected model must be provided manually by the tester which is highly inconvenient given the typical large number of tests that is necessary. A popular workaround is to run each test a first time when no expected model is initially available, and validate the result by manually verifying it is correct. If found correct, this result is saved as the expected model. While this eliminates the manual production of expected models, manual validation is tedious and highly error prone when models are large and complex as is usually the case.

Furthermore, MTs can evolve during their development. Assuming at a certain point a test suite using expected results as oracles has been developed, modifying the behavior of the transformation often requires updating a large number of the expected results to reflect the new correct behavior. Updating expected models is just as tedious and impractical as producing them for the same reasons mentioned earlier.

### 3.6.2 Contracts as Partial Oracles

Applying the classical principles of software design by contract, several approaches have proposed contracts for MTs [Cariou *et al.*, 2004; Guerra *et al.*, 2010] as a means of specification. As will be explained contracts may then act as automatic partial oracles [Cariou *et al.*, 2009; Guerra *et al.*, 2013; Guerra and Soeken, 2015] with several benefits that will be discussed shortly.

Contracts typically consist of three parts:

1. *Preconditions*: constraints that should be satisfied by the input model in order for the transformation to be applicable.
2. *Postconditions*: constraints that should be satisfied by the output model for any execution of the transformation.
3. *Transformation constraints*[7] relating the input and output models: constraints describing the relationship between elements of the output model and ele-

[7] these are called *invariants* in the PAMOMO specification approach of [Guerra *et al.*, 2010] that we already discussed in Section 3.4.3

ments of the input model. For refinement transformations that perform in-place transformation of elements, these constraints describe the evolution of model elements as a result of applying the transformation.

Given this specification, a transformation implementation is correct if for any input model satisfying the precondition, it produces an output model that satisfies the postconditions and the transformation constraints.

#### OCL Contracts

In [Cariou *et al.*, 2004] contracts are written as OCL constraints which makes it possible to execute them as automatic test oracles [Cariou *et al.*, 2009]. Given an input test model and the output model resulting from the test execution, the OCL contract can be executed to automatically determine whether the pair of models honors all constraints and is therefore correct with respect to the specification. Validity constraints of the output metamodel (irrespective of the particular transformation) may also be checked. If a constraint is violated, this indicates a discrepancy between the transformation implementation and its specification: the test fails.

#### Pattern-based Contracts in PAMOMO

As previously introduced in Section 3.4.3, PAMOMO [Guerra *et al.*, 2010] is a formal pattern-based language for the specification of MT contracts. Apart from preconditions and postconditions, so-called *invariants* correspond to the *transformation constraints* described above in (3) and express specification constraints roughly in the form: *if a certain* source pattern *appears in the input model then a certain* target pattern *should exist (or should not exist) in the output model*. In [Guerra *et al.*, 2010; Guerra and Soeken, 2015] the authors propose to compile PAMOMO specifications to a semantically equivalent set of OCL constraints over the input and output models of the transformation. They can thus be evaluated automatically by a standard OCL evaluator to serve as an automatic oracle function. In [Guerra *et al.*, 2013] an alternate compilation of PAMOMO to QVT is proposed so that contracts are not only checked, but detailed feedback is provided regarding which constraints were violated (if any) and the location of the violating elements.

#### Advantages

Contract-based oracles have several advantages. First, no separate test oracles needs to be developed since the specification itself plays that role. Moreover the specification is a single test oracle that applies to all tests. Second, if the transformation needs to be modified, its specification is necessarily updated (if a proper development process is followed), therefore no additional maintenance is required

for the test oracle. And finally, since testing mainly aims to show the conformance of an implementation with its specification, then using the specification itself to validate test executions is clearly a powerful test oracle.

#### Partial Oracles and Effectiveness

Contract-based oracles are called *partial oracles* because they only assess the properties that have been formalized in the contract. Any other properties not included in the contract will not be checked. Therefore this kind of oracles is only as good as the level of detail of the specification. This is experimentally observed in [Guerra and Soeken, 2015] where several test generation criteria (that we detailed in Section 3.4.3) are assessed via mutation analysis using contracts as test oracles. The maximum attained mutation score is 84% which seems relatively low compared to the maximum score of 98% reached with with metamodel coverage-based approaches detailed in Section 3.4.2.

In fact, the mutation analysis of metamodel coverage-based approaches employed a different test oracle which was a model comparison with the output of a reference implementation considered to be correct. Therefore the slightest divergence could be detected. Conversely, in [Guerra and Soeken, 2015] the authors trace the low mutation score to the inability of their contract-based oracles to detect a particular class of faults because of lacking constraints in the PAMOMO specification. Furthermore they go on to show that complementing the contract with the missing constraints allows to obtain a mutation of 100% *i.e.* all introduced faults are detected. We conclude that the fault revealing power of contract-based test oracles highly depends on the completeness and thoroughness of the specification.

#### Discussion

Qualification requires test oracles to be based on requirements. Since requirements can be specified as contracts, then the contract-based test oracles presented in this section are very interesting for a qualification process. Qualification also requires intensive verification (through review) of the completeness and thoroughness of requirements. If conducted properly, this verification would ensure that the situation discussed above, where the incompleteness of the specification caused errors not to be detected, is avoided.

## 3.7 Testing of Model Transformation Chains

So far we have reviewed a first set of works on MT testing, and we concluded in the various discussions that several of the existing approaches would be very

interesting in a qualification process. The coverage-oriented test adequacy criteria seem relevant and similar to the MC/DC requirements and code coverage criteria demanded in qualification. Automatic CSP-based model generation driven by such coverage criteria (or a combination of them) would allow to considerably reduce the testing effort and increase the confidence in the testing quality because of the systematic and exhaustive nature of the approaches. And finally, the cost of manual test oracles may be avoided with specification-based test oracles, provided that enough detail is included in the specification.

However, all of these approaches consider one MT in isolation. In contrast, complex model-based tools such as code generators typically consists of a chain of transformations rather than a single one. For this reason we now investigate works that have tackled MT chains to identify the challenges and existing solutions in that context.

### 3.7.1 Test Suite Quality for Model Transformation Chains

In [Bauer *et al.*, 2011] the authors propose an approach to assess the quality of a test suite for a Model Transformation Chain (MTC). The chains considered in this work are composed of several transformations. Each transformation takes as input the output of a preceding transformation, however the structure of the chain may have several branches as depicted in Figure 3.3.

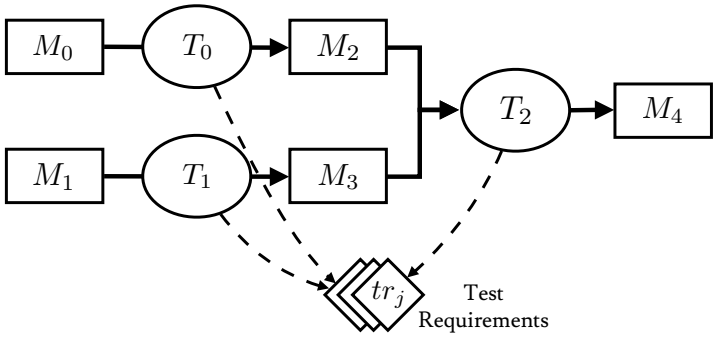



Figure 3.3: Example model transformation chain handled in [Bauer *et al.*, 2011]

The first step of the approach consists in deriving a set of test requirements for each transformation of the chain. Test requirements are derived from a combination of metamodel coverage criteria (similar to Section 3.4.2), contract coverage criteria (similar to Section 3.4.3) and manually written test requirements (in OCL). In a way, the test adequacy criteria here is a combination of several criteria applied to each transformation of the MTC. The result is the set of test requirements $\{tr_j\}$ visible in Figure 3.3.

| | $tr_1$ | $tr_2$ | $tr_3$ |
|---|---|---|---|
| test case 1 | 0 | 1 | 0 |
| test case 2 | 2 | 1 | 0 |
| test case 3 | 2 | 1 | 0 |
| test case 4 | 1 | 2 | 0 |

} ← redundant test requirement

↑ unsatisfied test requirement

Table 3.2: Footprint analysis for test suite quality assessment

The second step is to consider an existing test set and assess its adequacy with respect to the set of test requirements. Each test case is a pair $< M_0, M_1 >$ of models given as input to the chain. During each test, the models manipulated in the chain ($M_i$ for $0 \leq i \leq 4$) are inspected (automatically) to assess if they satisfy any test requirements and whether a test requirement is satisfied multiple times by the same model. This information is recorded in a so-called *footprint*[8] shown in Table 3.2. For each test case, the footprint is a vector of integer counters corresponding to the number of times each test requirement is satisfied during the execution of the test case. Each footprint constitutes a rough summary of the behavior of the corresponding test case. Analysing this information determines the following quality aspects of the test suite:

1. *Adequacy*: the test suite must satisfy each test requirement at least once to be considered adequate. *Unsatisfied test requirements* can be identified by looking for counters that are always 0 in all footprints *i.e.* the corresponding test requirements have never been satisfied.

2. *Minimality*: If two footprints have the exact same counters for all test requirements, then they are likely to exercise the same execution paths in the chain. They are thus considered redundant since they test the same functionality, and one of the them can be safely discarded. The test suite is minimal when it no longer contains redundant test cases.

**Discussion**

This work is highly relevant in our context since code generators typically consist of a chain of MTs. A qualification process applied to a chain of MTs typically yields a large number of test requirements to cover. It is therefore very convenient to automate the coverage analysis of test requirements as proposed in this work.

However, there are two aspects that this method does not address. First, the approach assesses an existing test suite but does not propose a way to create such a

[8] this notion of *footprint* is different from the transformation *footprint* used in Section 3.4.2

test suite from scratch. Second, when an unsatisfied test requirement is identified, the approach does not propose a way to generate new test models to cover that test requirement. It is entirely up to the tester to manually analyse the unsatisfied test requirement and design a new test model. That can be very tedious when dealing with a large transformation chain. As will be detailed in Chapter 4, this is one of the main problems tackled by this thesis.

## 3.8 Testing Code Generators

Several works of the literature address Automatic Code Generators (ACGs) specifically, therefore it is important to review them. We identified two prominent works that consider different aspects of source code: the first considers the *semantics* of source code while the second one considers its *syntax*.

### 3.8.1 Semantic Testing of Code Generators

The output of an ACG is executable source code. One way to verify the correctness of the generate code is by verifying its execution semantics, *i.e.* that it behaves correctly upon execution. When the input model of the ACG also has execution semantics, then the aim of this kind of testing is to show that the generated code has the same (or sufficiently similar) execution semantics as the input model. Therefore the test oracle of such approaches relies on comparing the executions of the input model and the output source code.

In [Sturmer and Conrad, 2003; Stuermer *et al.*, 2007] a general test architecture is proposed for model-based ACGs and illustrated for the optimisation part of a Simulink to C/Ada ACG. The approach is a semantical one, relying on the comparative execution of the input Simulink model and the generated source code.

#### First-Order Testing

The approach, illustrated in Figure 3.4 roughly consists in two levels of testing. First a partitioning analysis is conducted based on a specification of the ACG to produce a set of so-called *first-order test cases*. First-order test cases are Simulink models containing combinations of computation elements with parameter values chosen according to the partition analysis to trigger different optimisation rules in the specification. The ACG is invoked for each first-order test case to automatically generate the corresponding source code. At this point, the oracle that validates the generated source code is in fact a set of *second-order test cases*.

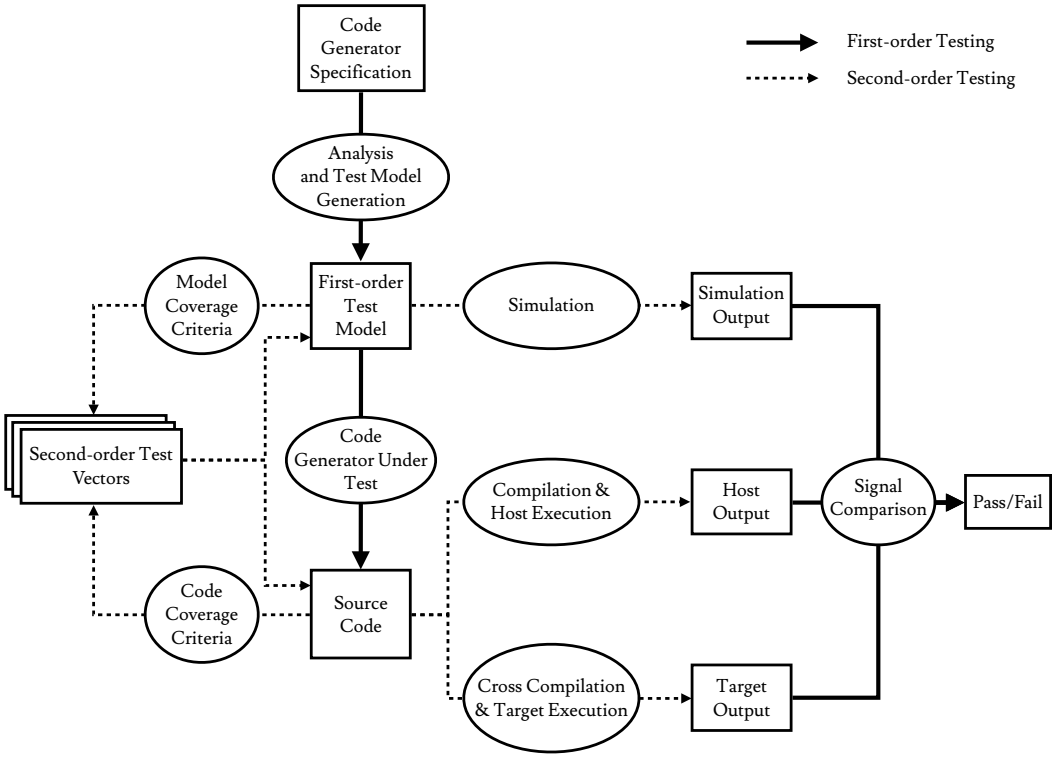



Figure 3.4: Testing model-based code generators

**Second-Order Testing**

Second-order testing consists of a comparative execution of the first-order Simulink test model and its corresponding generated source code over the same numerical input. Consequently second-order test cases are vectors of numerical data given as input to both a simulation of the Simulink input model and the generated code.

As depicted in Figure 3.4, second-order test cases are generated automatically using white-box structural coverage criteria on both the Simulink test model and the corresponding generated code. This ensures that all the model structure and all the generated code structure is executed during second-order testing since model and code may have different structures due to optimisations.

The same test vectors are given as input to model simulation and the generated code executed on the host modeling/development platform and/or on a separate embedded platform (to take into account target platform specificities). The output is compared with an acceptance threshold $\epsilon$ that masks differences due to known factors such as heterogeneous numerical precisions across the simulation and execution platforms. If second-order testing passes, then the corresponding first-order test passes: the generated code has the same execution semantics as the model from which it was generated.

If differences are detected between the simulation and the generated code execution(s), they must be analysed manually to determine the reasons which can be manifold:

1. an error in the implementation of the code generator.
2. a problem in the environment of the code generator *e.g.* operating system primitives or hardware.
3. an error in the test model.
4. an error in the simulator of the test model.
5. an error in the specification of the code generator.

**Discussion**

These works are highly interesting in the context of the qualification of code generators because they address one of the major objectives of qualification: showing that the code generator translates the model semantics into the generated code faithfully. What is particularly interesting is the thoroughness of the second-order testing which demonstrates the behavioral equivalence of source code with original models taking into account *both* model and code coverage criteria. Furthermore, the generated code is cross-compiled and executed on the embedded target platform which provides increased confidence that the model semantics is preserved on the target platform even after cross-compilation. As explained in Section 2.4.4 the qualification standard requires the involvement of the cross-compiler in the loop to claim certification credit for, and thus eliminate, the testing of the generated code.

Even though showing that the behavioral semantics of input models is preserved in the generated source code is a highly valuable result, this approach does not address the aspects of code structure. Indeed, as will be detailed in Chapter 4, the syntactical structure of source code is important for certification, and as a result it is important to show that the code generator generates source code that conforms to specific syntactical constraints. Since code with different structure can have the same semantics, then the above behavioral-oriented testing approach is insufficient to address structural aspects. For this reason, the structural aspects of code generator verification will be one of the main problems tackled in this thesis as detailed in Chapter 4.

The syntactic aspect of source code is also investigated in the next ACG testing approach.

### 3.8.2 Syntactic Testing of Code Generators

Most approaches of MT testing have considered model-to-model transformations. In contrast in [Wimmer and Burgueño, 2013] the authors propose an approach to specify contracts for model-to-text (M2T) and text-to-model (T2M) transformations. When we consider an ACG as an M2T transformation, then this approach allows to specify and verify the syntax of the generated source code. In contrast with the previous syntactic work, this approach only covers the aspect of test oracles and not of test model generation and test adequacy criteria.

The approach proposes a generic metamodel to represent any kind of text artifacts: it describes a hierarchy of *Folders* and *Files* composed of *Lines* of text. With this generic metamodel, the model-to-text (and text-to-model) specification problem is transposed back to a model-to-model specification problem, and contracts similar to the ones presented in Section 3.6.2 can be written to specify the transformation. The approach thus proposes to use OCL to write contracts specifying all aspects of the generated files such as the folder hierarchy, file naming patterns, and textual content.

For example the following contract excerpt for a UML to Java code generation transformation specifies that for each *Class* a *File* with the same name should exist with the extension `'java'`, and that for each derived *Attribute* of the *Class*, only a getter method should be generated, named after the *Attribute*'s name. Note that regular expressions are used to specify patterns of text. This contract specifies syntactic aspects of the source code such as the existence of a method with a particular name.

Listing 3.1: UML to Java M2T contract excerpt from [Wimmer and Burgueño, 2013]

```
1 Class.allInstances->forAll(c | File.allInstances->exists(f |
2   f.name = c.name and f.extension = 'java' and
3   c.attributes->select( a | a.isDerived)->forAll(a |
4     not f.content().matchesRE(a.type + '.*?' + a.name + '.*?;')
5     and f.content().matchesRE(a.type + '\\s+get' + toFirstUpper(a.name)))))
```

The OCL M2T contracts are then used as automatic test oracles, in combination with a simple parser that transforms the hierarchy of text files resulting from a test execution into a model conforming to the aforementioned generic text metamodel. Once the actual test result is parsed, a regular OCL evaluator can assess if the OCL contracts are satisfied and determine the PASS/FAIL verdict.

**Discussion**

This approach is appropriate for the specification and testing of the syntactic aspects of an ACG. In Section 2.4.4 we discussed that when qualification aims to claim certification credit for the compliance of the generated code with a code standard, then testing needs to address syntactic aspects of the source code. In that context the model-to-text contracts approach is relevant.

In fact as will be explained in Chapter 4, the syntactic specification of source code is one of the main problems tackled in this thesis. However we approach this problem from the specific needs of the qualification of QGen, the Simulink to C code generator developed at AdaCore (see Section 2.6). The solution that we propose in Chapter 5 bears some similarities with the approach of Wimmer *et al.* presented above, and a comparison will be discussed. Moreover, the approach of Wimmer *et al.* was published in late 2013, well after the inception of this thesis in early 2012. Therefore even though similarities exist, the two works were developed independently, both conceptually and in their tooling.

## 3.9 Conclusion

In this chapter we have given an overview on the body of existing works related to the testing of model transformations (MTs), model transformation chains (MTCs) and Automatic Code Generators (ACGs). Existing approaches target the three major challenges in testing which are (1) defining test adequacy criteria, (2) generating test models and (3) deciding the verdict of tests with test oracles.

Several of these approaches were found highly interesting and useful in the context of qualification. The test adequacy criteria based on metamodel and specification coverage are similar to the testing criteria required by qualification. These approaches can help automatically assess the adequacy of tests to the criteria of qualification. Automatic model generation approaches based on constraint solving reduce the effort needed to create test models. Finally, specification-based test oracles were found highly appealing given the strong requirements-based philosophy of testing in qualification. Additionally, we have identified relevant work addressing the testing of MTCs which are often the basis of code generators. The identified approach allowed to assess the adequacy of a test set with respect to all the transformations of a MTC instead of only one transformation in isolation. As for the specific case of ACGs, we identified an approach which verifies the semantics of the generated source code by comparing its execution with a simulation of the input model in very thorough manner. As for the syntax of source code, we identified an approach which proposes model-to-text contracts as a test oracle for

ACGs. Overall, we conclude that several existing approaches already address various testing challenges in qualification.

However we identified two areas of improvement in the literature. The first one pertains to the testing of MTCs. Even though existing work allows to identify non-satisfied test requirements of a MTC, it lacks a method for producing new test models to cover these test requirements. The other area of improvement was in the syntactic specification of ACGs where we expressed a need for a solution specific to the qualification of QGen at AdaCore.

In the next chapter we define the precise problems that we chose to tackle in this thesis, explain how the existing approaches identified above can help address them and where new solutions are needed.

# Chapter 4

# Analysis and Problem Statement

## Contents

## 4.1 Introduction

In Chapter 2 we explained the challenges of the qualification of Automatic Code Generators (ACGs) and decided to focus this thesis on the issues of testing. In Chapter 3 we gave an overview of the existing body of work on the testing of Model Transformations (MTs), Model Transformation Chains (MTCs) as well ACGs. In this chapter, we detail the concrete problems that we chose to address in this thesis, explaining where existing approaches can be reused and highlighting the key issues that we tackle.

Our work addresses two main challenges that are detailed in turn in Section 4.2 and Section 4.3.

## 4.2 Unit Testing and Integration Testing of Model Transformation Chains

Qualifying an Automatic Code Generator (ACG) requires extensive testing to show the compliance of the implementation with its requirements. Both the testing of components in isolation (*i.e. unit testing*) and the testing of the tool as a whole (*i.e. integration testing*) are required. As will be explained in the following sections, a code generator typically consists of a chain of successive model transformations which makes unit testing particularly difficult to conduct. Conversely integration testing is easier to apply. Since both are required for the qualification of the ACG, the problem is then to design a testing approach relying solely on integration testing but providing the same confidence as unit testing. The next sections will motivate these claims and detail the problem further.

### 4.2.1 Unit Testing and Integration Testing in Qualification

In a qualification process testing should be based on the requirements developed at all refinement stages of the tool design: Tool Operational Requirements (TORs), Tool Requirements (TRs) and Tool Low Level Requirements (TLLRs). For simplification let us consider two of these stages, TOR and TLLR[1].

#### Integration Testing and Unit Testing of a MTC

TORs consider the tool as a whole. Therefore tests based on TORs define test data given as input to the tool and validate the output of the tool. We will refer

[1] depending on the complexity of the qualified tool, it is acceptable to omit the Tool Requirements stage if deemed unnecessary

to such tests as *integration tests* since they exercise the tool as a whole. Conversely, TLLRs specify subcomponents or *units* of the tool which have a specific role in the overall operation. Tests based on TLLRs define test data given as input to a subcomponent and validate the output of that subcomponent. We will refer to such tests as *unit tests* since they exercise one subcomponent. Both unit and integration tests are required for the qualification of an ACG.

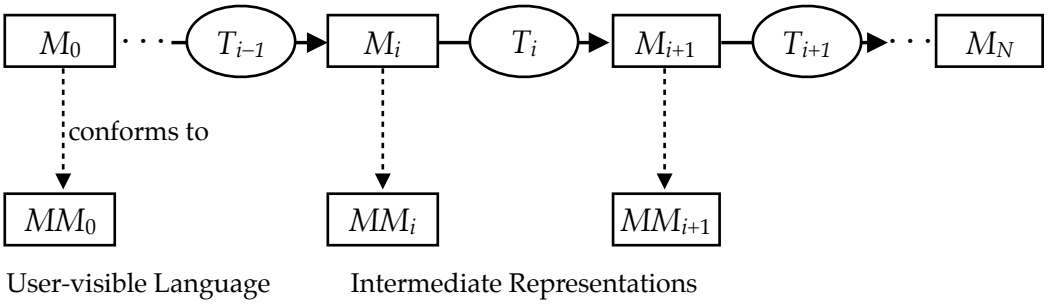



Figure 4.1: Model transformation chain

As shown in Figure 4.1 an ACG is typically designed as a chain of $N$ model transformations $M_i \xrightarrow{T_i} M_{i+1}$ for $0 \leq i < N$ where an input model $M_0$ is processed by $N$ successive transformation steps $T_i$ into intermediate models $M_i$, and ultimately into the final output model $M_N$ which is in fact source code in the case of an ACG. $M_0$ conforms to the input metamodel of the chain $MM_0$ which is the language exposed to the users of the ACG. The intermediate models $M_i$ conform to intermediate metamodels $MM_i$ which are internal to the tool. For several transformation steps $T_i$, we can have $MM_i = MM_{i+1}$ meaning that $T_i$ is an endogenous refinement transformation within the same language.

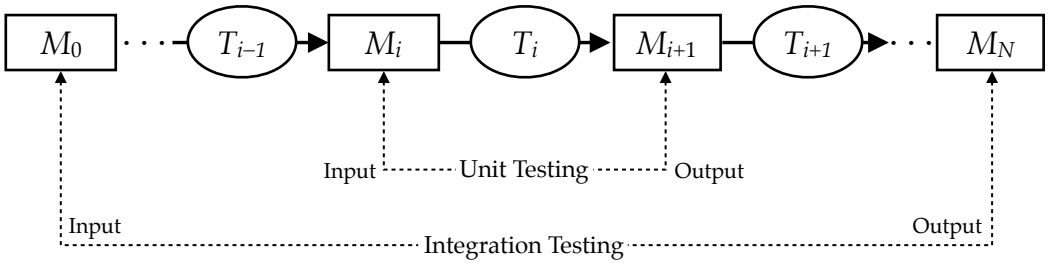



Figure 4.2: Unit testing and integration testing of a model transformation chain

Given that architecture of the ACG, a *unit* is a transformation step $T_i$. As depicted in Figure 4.2 *Unit testing* then consists of producing test models $M_{i,j}$ in the intermediate language $MM_i$, executing $T_i$ over these test models, and validating the resulting models $M_{i+1,j}$ with a suitable oracle. Conversely, *integration testing* considers the complete chain, producing test models $M_{0,j}$ in the input language

$MM_0$, executing the complete chain, and validating the final result $M_{N,j}$ with a suitable oracle.

**Observations in Practice**

While both unit and integration testing are required by qualification standards and general best practices, we observe in development practice that integration testing is widely preferred to unit testing which is perceived as tedious and cumbersome. This feedback comes from developers of the QGen code generator at AdaCore as well as developers of the GCC compiler toolchain which has a similar chain-based architecture. In both projects, the vast majority of tests are integration tests while unit tests are nearly non-existent. Additionally, this problem is confirmed in existing work on code generator testing in [Stuermer *et al.*, 2007] (detailed in Section 3.8) where the authors highlight the difficulty of unit testing for code generators due to the difficulty of providing test stubs and drivers for the kind of components involved in code generators.

The main reason for the unpopularity of unit testing is that it relies on test data in the intermediate languages of the chain. Intermediate languages typically become more and more complex down the transformation chain, making it difficult to produce and maintain unit test models. This is because the ACG starts with the high-abstraction user model $M_0$ and transforms it step by step into intermediate models $M_i$ with increasing detail and decreasing abstraction level. One element in the input model, *e.g.* a Simulink computation block, is typically transformed into several elements in the intermediate representation that together implement its semantics, *e.g.* a variable to store the result and several code statements that perform the computation. Moreover, intermediate languages typically do not have dedicated editors since they are internal representations. Producing, validating and maintaining unit test models manually is therefore a tedious and error-prone task.

As was detailed in Section 3.4 and Section 3.5 existing approaches can automatically generate test models based on test adequacy criteria. However they typically yield a very large number of test models which remain difficult to maintain because they are in the intermediate languages. Maintaining unit test models is an issue because intermediate languages often evolve during the development life cycle of the tool. Therefore assuming that a set of unit test models was produced, whenever the intermediate representation changes, identifying which unit tests are affected and propagating the necessary changes is not trivial.

In contrast, integration testing is widely preferred because it relies on test models expressed in the input language of the chain. The input language is the one used by users of the ACG, therefore it typically has a high level of abstraction, a good

editor, and does not evolve often. Producing test models, even manually, is therefore conceivable and the maintenance problem is reduced thanks to the stability of the language.

We conclude from the above observations that unit testing is a major hindrance in the verification process of an ACG and that it would be desirable to perform *only* integration testing. However how can we achieve the same confidence as unit testing to adhere to qualification requirements? Let us investigate in detail the steps required by unit testing and whether we can ensure them with integration testing.

### 4.2.2 Formalising Unit Testing

In this section we aim to formalise the steps and artifacts needed for the unit testing of each transformation step $T_i$ of the MTC. This will allows us to discuss in a later stage whether we can achieve the same level of confidence of unit testing through other means.

#### Step 1: Defining Unit Test Requirements

Considering a transformation $T_i$, the first step of unit testing is to produce a set of *unit test requirements* that characterise the inputs with which the transformation should be tested.

**Definition 4.1 (Unit Test Requirement).** A unit test requirement[2] $tr$ of a transformation step $T_i$ is a constraint over its input language which must be satisfied at least once during the testing campaign. Assuming $MM_i$ is the set of input models of $T_i$, we formalise a test requirement as a predicate indicating whether a given model $M \in MM_i$ exhibits a particular characteristic expressed by the test requirement.

$$\begin{aligned} tr \ : \ MM_i &\longrightarrow \mathbb{B} = \{\text{True}, \text{False}\} \\ M &\longmapsto tr(M) \end{aligned}$$

A test requirement $tr$ is said to be *satisfied* or *covered* by a model $M$ if $tr\,(M) = \text{True}$. This is denoted as $M \models tr$.

□

As explained in Section 3.3.1, test requirements are a way to express *what situations should be tested* in order to ensure the compliance with a certain *test adequacy criteria*. We presented various test adequacy criteria in Section 3.4, some based on

[2] the prefix *unit* is often omitted for convenience

the partitioning and coverage of the input model, and others based on coverage of the specification. With these well defined, test requirements can be generated automatically. If each test requirement is satisfied at least once during the testing campaign, then we have satisfied the test adequacy criteria.

It is not necessary to satisfy test requirements independently. This means that multiple test requirements can be satisfied by the same test model. If two test requirements express properties that need to be tested independently, then this must be explicit in each of them: each test requirement should contain the negation of the other test requirement to enforce independent testing.

Test requirements may also be specified manually by the tester based on his knowledge of the transformation. If the tester would like to test a particular feature, he can express the conditions that trigger it into a test requirement.

The result of this first step is a set of unit test requirements that must be satisfied: $UTR_i = \{tr_{i,j}\}$

**Step 2: Defining Unit Test Oracles**

A test oracle is a procedure to determine if the verdict of a test is *PASS* or *FAIL*. As discussed in Section 3.6 several forms of test oracles exist, and the preferred form in qualification is specification-based oracles.

**Definition 4.2 (Unit Test Oracle).** We assume that each transformation $T_i$ has an executable contract-based specification that constitutes its test oracle $to_i$. Therefore we assume that all unit tests of $T_i$ are validated with the same test oracle $to_i$. We formalise this test oracle as a predicate over the input test model and its corresponding output model that determines if the pair complies with the specification of the transformations:

$$\begin{aligned} to_i \ : \ MM_i \times MM_{i+1} &\longrightarrow \mathbb{B} \\ \langle M_{in}, M_{out} \rangle &\longmapsto to_i\,(M_{in}, M_{out}) \end{aligned}$$

□

**Step 3: Developing Unit Test Models**

Unit test models are created either manually by the tester or automatically with model generation approaches from the literature detailed in Section 3.5. The goal of this step is to satisfy all unit test requirements, however one test model can satisfy

multiple test requirements at the same time, as explained earlier. The result is a set of unit test models: $UTM_i = \{M_{i,k}\}$

**Step 4: Executing Unit Tests and Evaluating Oracles**

The transformation $T_i$ is executed for each test model $M_{i,k}$ to produce a result $T(M_{i,k})$. The validity of this result is determined by evaluating $to_i(M_{i,k}, T(M_{i,k}))$ and the PASS/FAIL verdict of the test is decided accordingly.

**Objective and Effectiveness of Unit Testing**

After performing unit testing of all transformations of the MTC, the general objective is to cover all test requirements and ensure that all tests pass. If the above steps are followed systematically for each transformation, then this objective should be ensured for the complete MTC. We formalise it as follows.

**Definition 4.3 (Objective of Unit Testing of a MTC).** The objective of unit testing for a complete MTC is to ensure that each unit test requirement is satisfied at least once and that all tests pass.

$$\forall i \in [0, N[ \; . \; \begin{pmatrix} \forall tr \in UTR_i \; . \; \exists M \in UTM_i \; . \; M \models tr \\ \wedge \; \forall M \in UTM_i \; . \; to(M, T_i(M)) = \text{True} \end{pmatrix}$$

□

Having formalised the steps and general objective of unit testing, we note the following. Unit testing is effective at detecting errors because test requirements are generated using systematic test adequacy criteria and test oracles are based on the detailed specification of each transformation step. This ensures thoroughness of both the test data and the test oracles. However we argue that developing unit test models can be avoided while maintaining the confidence provided by unit test requirements and oracles. We develop this argument in the next section.

### 4.2.3 Achieving Unit Testing Confidence Through Integration Tests

Taking inspiration from the work in [Bauer *et al.*, 2011] introduced in Section 3.3 we notice the following: an integration test exercises the complete tool, *i.e.* all intermediate transformation steps $T_i$. During the execution of an integration test, the intermediate models $M_i$ manipulated along the way can cover unit test cases of the intermediate transformations. This interesting property of transformation chains would allow us to use *only* integration testing to cover unit test cases.

**Covering Unit Test Cases with Integration Tests**

| Integration Test Models | Test Requirements and Oracles | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | $T_0$ | | | | $T_1$ | | | $T_2$ | | |
| | $tr_{0,0}$ | $tr_{0,1}$ | $tr_{0,2}$ | $to_0$ | $tr_{1,0}$ | $tr_{1,1}$ | $to_1$ | $tr_{2,0}$ | $tr_{2,1}$ | $to_2$ |
| $M_{0,0}$ | T | F | F | T | F | T | T | F | F | T |
| $M_{0,1}$ | F | T | F | T | T | F | T | T | F | T |
| $M_{0,2}$ | F | F | T | T | F | T | T | F | T | T |

Table 4.1: Integration testing with unit testing confidence

This is illustrated in Table 4.1 where we consider a chain of 3 transformations $T_0$, $T_1$ and $T_2$. We assume that only the first two steps of unit testing of these transformations was done, and yielded 3 test requirements for $T_0$, 2 test requirements for $T_1$ and 2 test requirements for $T_2$, as well as a unit test oracle for each each transformations. We also assume that 3 integration test models $M_{0,0}$ , $M_{0,1}$ and $M_{0,2}$ have been developed in the input language of the chain $MM_0$. We execute the complete chain over each model, as part of integration testing. However, during each execution over $M_{0,k}$ we do the following:

1. Before executing each transformation step $T_i$ we evaluate all of its unit test requirements $tr_{i,j}$ over the input intermediate model $M_{i,k}$, and record the results of the evaluation.

2. After executing each transformation step $T_i$, we evaluate its unit test oracle $to_i$ on the pair $\langle M_{i,k} \ , \ T_i\,(M_{i,k})\rangle$.

The evaluations may also be done *a posteriori* by dumping the intermediate models computed by each step of the MTC and evaluating test requirements and oracles after the execution. In any case, we record the results of these evaluations in a so-called *footprint*[3] of the execution. In Table 4.1 each row corresponds to the footprint of an integration test execution. "T" indicates a result True of the evaluation and "F" indicates False.

We can observe in Table 4.1 that all test requirements are satisfied at least once, and all test oracles always evaluate to True. This means that even though unit test models have not been explicitly identified as in unit testing, we have achieved the objective of unit testing defined in Definition 4.3. Therefore we have performed a testing campaign based solely on integration test models, and ensured the same confidence as unit testing.

[3]the concept is similar to the footprints in [Bauer *et al.*, 2011]

**Identifying Non-Satisfied Test Requirements**

We now consider a different scenario where instead of the integration test model $M_{0,1}$ we use a model $M_{0,4}$. The execution footprint of $M_{0,4}$ is shown in Table 4.2 and we can see that $tr_{1,0}$ is not satisfied. By analysing the footprints of all tests, we determine in fact that $tr_{1,0}$ is never satisfied during the testing campaign. In this case we need to develop a new integration test model that leads to the satisfaction of $tr_{1,0}$. The approach of [Bauer *et al.*, 2011] that was the basis for this analysis does not provide a way to produce such a model. This brings us to the core problem that we try to solve in this thesis: *How can we produce a new integration test model to cover a non-satisfied unit test requirement?* We develop this problem next.

| | **Test Requirements and Oracles** | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Integration Test Models** | $T_0$ | | | | $T_1$ | | | $T_2$ | | |
| | $tr_{0,0}$ | $tr_{0,1}$ | $tr_{0,2}$ | $to_0$ | $tr_{1,0}$ | $tr_{1,1}$ | $to_1$ | $tr_{2,0}$ | $tr_{2,1}$ | $to_2$ |
| $M_{0,0}$ | T | F | F | T | F | T | T | F | F | T |
| $M_{0,4}$ | F | T | F | T | F | F | T | T | F | T |
| $M_{0,2}$ | F | F | T | T | F | T | T | F | T | T |

Table 4.2: Identification of non-satisfied test requirements

### 4.2.4 The Problem: Producing Integration Test Models to Cover Unit Test Cases

We have demonstrated in the previous sections that based on existing work in the literature, it is possible to assess the satisfaction of unit test requirements during the execution of integration tests, and identify non-satisfied unit test requirements. We now need a way to produce new models to cover these unit test requirements. Given a non-satisfied test requirement $tr_{i,j}$ in Figure 4.3 how can we produce a test model $M_0$ in the input language of the chain such that upon execution of the test, $tr_{i,j}$ is satisfied?

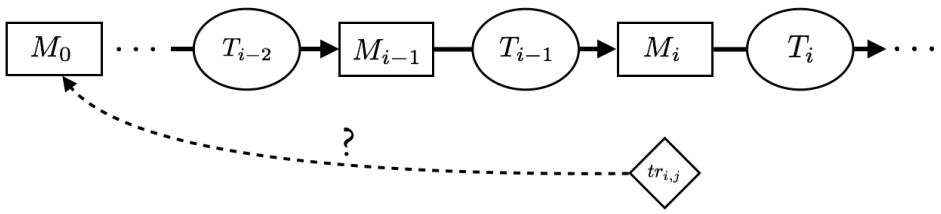



Figure 4.3: Satisfying unit test requirements

Producing $M_0$ manually would require the tester to start with $tr_{i,j}$ and reason in *reverse* on all the preceding transformation steps. This is a difficult task because

there may be many preceding transformation steps (the typical length of the chain is ∼15 steps) which are often non-bijective and therefore non-reversible. Therefore we have to turn to automation.

To automate this task, we first notice that it is in essence a *Constraint Satisfaction Problem* (CSP) where $tr_{i,j}$ is a constraint that we wish to satisfy with an appropriate model instance. Using the approaches detailed in Section 3.5.2, we can encode the model generation problem in the form of a CSP. Different solvers (SAT/SMT-Solvers) may then be used to generate an instance $M_i$ in the intermediate language $MM_i$ that satisfies $tr_{i,j}$. This is shown by arrow 1 in Figure 4.4. However this is not what we are looking for because we do not want to maintain unit test models in intermediate languages. Instead, we want to produce a model $M_0$ in the input language of the chain.

In the literature we find approaches dealing with bidirectional transformations [Stevens, 2008] that provide a *backward* execution of transformations which would allow to produce $M_0$. However the transformations of our chain are not bidirectional by nature and are not designed as such, therefore their backward execution is not feasible.

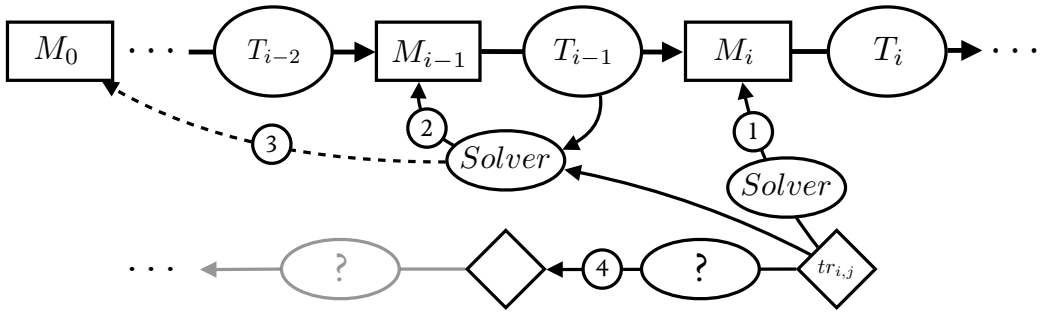



Figure 4.4: Existing model generation solutions

We turn to the notion of a *transformation model* that we briefly discussed in Section 3.2 of our literature review where we gave an overview of formal verification approaches. Even though originally designed for verification, this concept can be useful in solving our problem. A transformation model combines the input and output metamodels of the transformation and trace elements linking input and output metaclasses, with a set of constraints describing the relationship between input and output elements according to the semantics of the transformation. In [Büttner *et al.*, 2012b] the transformation model is translated to a CSP in Alloy, and the Alloy Analyzer (relying on a SAT solver) is used to find instances of the transformation model that violate a postcondition. Such an instance includes the input model that leads to the violation of the postcondition. Therefore what this approach has es-

sentially managed to do is find an input model which leads to the satisfaction of a constraint[4] on the output model.

Going back to our problem, this means that by encoding the transformation $T_{i-1}$ itself as a transformation model in a CSP and specifying $tr_{i,j}$ as a constraint on the output, we can generate an instance $M_{i-1}$ that leads to the satisfaction of $tr_{i,j}$. This approach is illustrated by arrow 2 in Figure 4.4. However this is still not satisfactory because the instance is in the intermediate language $MM_{i-1}$ while we seek test models in the input language of the chain. Moreover, having obtained the *instance* $M_{i-1}$, we cannot iterate the transformation model approach again for $T_{i-2}$ because what we need as input to the CSP solver is a *constraint* and not an *instance*.

With a similar approach, we could theoretically generalize the notion of the *transformation model* to include more than one transformation. This means that instead of modeling only $T_{i-1}$ in the transformation model, we could model all of the preceding chain of transformations: $T_{i-1} \circ \cdots \circ T_1 \circ T_0$. In this manner we could include all preceding steps in the resulting CSP which would theoretically allow us to generate a satisfactory $M_0$ as shown by arrow 3 in Figure 4.4. However given the large number of steps in the chain the resulting size of the CSP would prevent a realistic analysis since CSP solvers have well known scalability issues.

What we need instead is an iterative solution which can tackle the problem step by step instead of dealing with all preceding transformation steps at once. This requires an analysis that takes as input a *constraint* and produces also a *constraint* as shown by arrow 4 in Figure 4.4 allowing for an iterative reasoning. With such an analysis we would be able to propagate the test requirement step by step backwards along the chain, up to an equivalent test requirement over the input language. Then using classical model generation techniques we can produce an integration test model $M_0$ which upon execution of the chain covers the unit test requirement.

Hence the first problem that we tackle in this thesis is:

*Translating a non-satisfied unit test requirement backwards along a model transformation chain into an equivalent constraint over the input language of the chain.*

This concludes the discussion of the first problem which stemmed from the general goal of ensuring the coverage of unit test cases with integration tests. We now move to the second problem which concerns the oracles of these integration tests.

[4] the negation of the postcondition

## 4.3 Specification-based Test Oracles for Model-to-Code Transformations

In the previous section we advocated the use of integration tests to cover unit testing needs and focused on the problem of producing them. We now move to the second part of this thesis where we focus on the problem of defining the oracles of these integration tests in the case of a code generation chain. As will be explained this is closely related to the problem of specifying requirements in the DO−330 qualification process.

Before we proceed with our presentation, it is important to note the following. This thesis is a CIFRE[5] collaboration with the company AdaCore. For AdaCore, qualification is a critical aspect in the development of QGen, the Simulink to C code generator. As a result, this second part of the thesis was explicitly required to focus on specific needs of the qualification of QGen. Consequently the challenges identified in the upcoming presentation and the solutions proposed in subsequent chapters are less general than the previous discussions and more specific to the qualification of QGen.

Several exiting works on test oracles for model-to-code and model-to-text transformations have been presented in Chapter 3, however the context of qualification raises specific issues. As will be detailed next, qualification requires test oracles to be based on the requirements of the tested software, therefore we seek an approach that combines requirements specification and requirements-based test oracles. Given the certification credit that we target by the qualification of the code generator, the requirements must address the syntax as well as the semantics of the generated code. Since the semantic aspect is sufficiently covered in the literature we focus on syntactic specification and test oracles of code generation. The following sections detail and motivate these claims.

### 4.3.1 Test Oracles and Requirements in Qualification

As previously detailed in Chapter 2, the qualification process of a tool such as an ACG is strongly centered around *requirements*. They specify the required operation of the tool and are the basis for its design and implementation as well as its testing. In testing activities, the results of tests should be decided based on the requirements which specify the required behavior and outputs of the software. For this reason when discussing test oracles in a qualification process, it is necessary to consider

[5]Convention Industrielle de Formation par la REcherche

*requirements-based* test oracles, also referred to as *specification-based* test oracles in the academic literature.

We are concerned with so-called *integration tests* of an ACG as defined in Section 4.2: they exercise the ACG as a whole. The requirements that specify the tool as a whole and that should be the basis for these tests are the *Tool Operational Requirements* (TORs). As detailed in Section 2.4, TORs describe the operation of the tool as a whole and are the main justification for the elimination of *a posteriori* verification activities of the output of the tool. For this reason TORs must include a specification of the output of the tool, precise enough to justify the elimination of verification activities.

Even though certification and qualification processes traditionally rely on textual requirements in natural language, there is no actual constraint on the representation chosen for requirements as long as it is justified and well defined in the *Tool Requirements Standard*. Given the level of detail required for TORs and the fact that they are the basis of integration test oracles, we would like to explore the idea of formalising TORs in a precise and machine readable representation that can support automatic TOR-based test oracles. Now let us investigate what aspects should be specified by the TORs.

### 4.3.2 Requirements Specification of a Code Generator

In the qualification of an ACG, TORs are the main justification for the elimination of verifications of the output of the ACG. Therefore the content of the TORs and the subsequent TOR-based review and testing activities depend on the eliminated verifications, *i.e.* the *certification credit* claimed from the qualification of the ACG. In general, the TORs must specify the behavior of the tool and define the generated Source Code (SC). In some cases only the syntactic aspect of SC is relevant while in other cases both syntactic and semantic aspects of SC are important.

To illustrate this, we recall the following verification objectives for which certification credit may be claimed through the qualification of the ACG, as detailed in Section 2.2.2:

O1. Conformance of the Source Code (SC) with the Code Standard

O2. Compliance of the Source Code (SC) with Low-Level Requirements (LLR)[6] and the Software Architecture

O3. Compliance and robustness of the Executable Object Code (EOC) with the Low-Level Requirements (LLR)

[6] the LLR are formalised as the input model of the ACG

For objective O1, what is important is the *syntax* of the generated code. The Code Standard describes syntactic rules such as constraints over the level of nesting of control structures (loop and conditional constructs) or the number of statements allowed in subprograms. If the qualification of the tool claims credit for O1, then the TORs must include a precise syntactic specification of the generated code allowing to justify its compliance with the Code Standard. TOR-based testing of the ACG should also focus on syntax: test oracles of the ACG should inspect the generated code and show that it conforms to the syntactic specification of the TORs.

For objectives O2 and O3, it is necessary to consider the *semantics* of the SC, *i.e.* its behavior. In O2 the compliance of the SC with the input model (LLR) encompasses both the syntax and the semantics of the SC. The goal is to show that the behavior of the SC implements the behavior expressed by the LLR which is typically done by an independent[7] manual review. In O3 it is only semantics that matters since the EOC results from the compilation of the SC, at which point syntax is no longer relevant. O3 is demonstrated via LLR-based testing. When O2 and/or O3 are to be eliminated through the qualification of the ACG, then the TORs must specify the semantics of the generated code and TOR-based testing of the ACG should encompass semantics: test oracles of the ACG should verify that the behavior of the generated code conforms to the semantics of the input model.

### 4.3.3 Semantic and Syntactic Test Oracles

#### Semantic Test Oracles

In the context of our work the input of the ACG is a Simulink model and the output is C SC. To claim credit for objectives O2 and O3 we need to demonstrate the semantic compliance of the generated code with Simulink. In that task, the ACG testing approach proposed in [Sturmer and Conrad, 2003; Stuermer *et al.*, 2007] and detailed in Section 3.8.1 does a very thorough job. In this approach, for each Simulink test model SC is automatically generated, and the test oracle consists in executing the generated SC over vectors of numerical data (so-called *second-order tests*) and comparing the computation results of the SC with the computation results of the Simulink simulator. With this approach, the test oracle demonstrates that the behavior of the SC generated by the ACG complies with the semantics of the input Simulink model. The approach also takes into account model and SC coverage as well as different execution platforms during second-order testing which makes for a thorough verification that we consider sufficient for qualification needs.

[7] the reviewer is a different person than the developer of the reviewed artifact

Note that the above approach is not strictly specification-based as required by the qualification standard because nowhere was the Simulink semantics actually *specified*. In fact the original problem is that the Simulink semantics is only specified in the extremely verbose natural language form of the Simulink user documentation. For this reason the Simulink simulator is widely regarded as the reference for the Simulink execution semantics, as was done in the approach described above. This lack of formal specification of Simulink and of data flow languages in general is an open problem from an academic and a qualification point of view and is actively investigated in works such as [Dieumegard *et al.*, 2014a; Dieumegard *et al.*, 2014b; Dieumegard *et al.*, 2012]. Nonetheless, we consider that from a testing viewpoint, the works on semantic test oracles presented above are sufficient and we focus on syntactic test oracles.

**Syntactic Test Oracles**

There are two ways to consider the syntax of source code:

(a) Consider SC as a structured artifact based on its Abstract Syntax Tree (AST)

(b) Consider SC as plain text based on its Concrete Syntax

In (a) SC is viewed as a model consisting of its AST and conforming to an appropriate metamodel of its AST. The ACG becomes a model-to-model transformation of the input model to the output code model and we can apply existing specification-based test oracle approaches such as the ones presented in Section 3.6.2. TORs can be expressed as contracts of the transformation: preconditions, postconditions and transformation invariants relating input and output elements. Given an input test model and its corresponding output via the ACG, such TORs can be executed to determine if the contract is honored, thus constituting a specification-based automatic test oracle.

However such an approach is not compatible with the readability needs of qualification. TORs are the subject of several reviews by different stakeholders. When a tool is qualified in the context of a certified project, the certification applicant, *i.e.* the tool user, has the responsibility of reviewing the TORs and ensuring they justify the claimed certification credit. The AST of SC is a complex structure and contracts expressed on this AST will undoubtedly grow complex and become incomprehensible to non-experts of the technology. Thus we believe that specifying TORs in terms of the AST of the generated SC is not a suitable approach.

In that regard, (b) is a more promising investigation track because it considers code in its most common and well known form: the concrete syntax. This implies considering the ACG as a model-to-text transformation and applying model-to-text

test oracle techniques. To the best of our knowledge, only one approach has proposed model-to-text test oracles in [Wimmer and Burgueño, 2013]. We detailed this approach in Section 3.8.2 and explained that it consists in viewing the generated textual artifacts as a rough model composed of *Folders* containing *Files* containing *Lines*. However the generated content is not modeled beyond this level of detail to preserve its textual nature. Based on this general model, contracts can be written to specify the generated SC in terms of portions of actual code that should be generated verbatim, combined with regular expressions expressing textual patterns that should be found in the generated code. TORs can thus be written as model-to-text contracts which can also be executed to determine whether the output of a test complies with the specification, thus constituting a specification-based automatic test oracle.

While this approach enhances readability for non-experts because it presents code in its textual form, we believe it needs to be adapted to our context before it can be used. Contracts in this approach are OCL constraints that can arbitrarily mix constraints over input elements with constraints over output artifacts and their content in a manner that can again compromise readability. It is therefore necessary to organise contract specification with guidelines and usage patterns to ensure uniformity in TORs and allow their presentation as a formal qualification document.

Moreover, at the time of inception of this thesis, the work of Wimmer *et al.* was not yet published. As a result, it could not be applied to our context at the time leading us naturally to seek our own solution to this problem that specifically targets the needs of AdaCore in qualification. Even though our approach bears some conceptual similarities with the approach of Wimmer *et al.*, it does not purposefully reuse aspects from that approach. A comparison of both approaches will be discussed in Chapter 9.

### 4.3.4 The Problem: Devise a Syntactic Specification and Test Oracles Approach for an ACG

The analysis in the previous sections has shown that test oracles of integration tests of an ACG are strongly related to the specification of TORs in the context of qualification. We've explained that both semantic and syntactic aspects of code generation are important. While the former semantic aspect already has a satisfying solution in the state of the art, the latter syntactic aspect does not have a readily usable solution that satisfies the readability needs of qualification. Therefore the second problem tackled in this thesis is:

*Devising a specification and automatic test oracles approach for ACGs, focusing on the syntax of the generated source code and the readability of the specification.*

## 4.4 Conclusion

In this chapter we have defined the main problems tackled by this thesis. Noticing the difficulties of conducting unit testing for model transformation chains such as Automatic Code Generators (ACGs), we have shown that with existing works it is possible to achieve the confidence of unit testing using only integration test models that are easier to develop and maintain. However, even though existing approaches can identify non-satisfied unit test requirements, producing new integration tests targeting the non-satisfied test requirements remains an open problem. We have determined that this can be done by propagating non-satisfied unit test requirements into equivalent constraints over the input language of the chain, something that is not straightforward with the current state of the art. This lead us to tackle this particular problem as a first challenge in this thesis:

*Translating a non-satisfied unit test requirement backwards along a model transformation chain into an equivalent constraint over the input language of the chain.*

Having advocated the use of integration tests, we then investigated the oracles of such tests and determined that they must be based on the specification of the ACG defined by the Tool Operational Requirements (TORs) of the qualification process. While TORs and their associated tests must cover both semantic and syntactic aspects of code generation, we have determined that existing work on the semantic aspect is sufficient while existing work on the syntactic aspect does not correspond to the needs of qualification. In particular, TORs must be readable because they are presented as qualification evidence that is subject to thorough reviews. Since TOR specification is a critical aspect of the qualification of QGen for AdaCore, the second challenge tackled in this thesis is specific to that context:

*Devising an ACG specification approach focusing on the syntax of the generated source code and the readability of the specification, and supporting automatic test oracles.*

In the next chapter, we give an overview of the solutions we propose to each of the identified challenges and highlight the main contributions of this thesis.

# Chapter 5

# General Approach

## Contents

## 5.1 Introduction

In this chapter, we give an overview of the approaches we propose to address the challenges identified in the previous chapter. In a first part we propose an approach to the backward translation of test requirements, and in a second part we propose a syntactic specification and test oracles approach for model-to-code transformations. As explained earlier, the first part of our work targets the general scope of model transformation chains that was discussed in previous chapters, while the second part is focused on the needs of AdaCore in the context of the qualification of QGen.

## 5.2 Backward Translation of Test Requirements

The first problem tackled in this thesis emerged in the context of the relationship between unit testing and integration testing of a model transformation chain. In the previous chapter we explained that it is possible to satisfy unit testing needs using only integration tests. This can be done by extracting from each transformation of the chain a set of unit test cases, each characterised by a *test requirement*. Then using only integration tests, it becomes possible to assess the satisfaction of unit test requirements during the execution of integration tests and to identify *non-satisfied* unit test requirements. At this stage we need to produce a new integration test that satisfies the unit test requirement.

We determined that producing new tests can be done by propagating the non-satisfied test requirement step by step backwards along the chain, into an equivalent constraint over the input language of the chain. This is the main problem that we address in the next sections.

### 5.2.1 Core Principle : Backwards Translation of Constraints

We propose to solve the problem with the approach illustrated in Figure 5.1. Given a non satisfied test requirement $tr_{i,j}$, we propose to consider $tr_{i,j}$ as a *postcondition* of the previous transformation step $T_{i-1}$, and design a construction *Post*2*Pre* that transforms the postcondition into an equivalent *precondition* that ensures the satisfaction of the postcondition. We call this precondition the *equivalent test requirement* $etr_{i,j,i-1}$ of $tr_{i,j}$ at step $T_{i-1}$.

The resulting precondition is also a constraint, therefore the same construction can be iterated again for preceding steps until an equivalent test requirement $etr_{i,j,0}$ over the input language of the chain is reached. At that point existing constraint

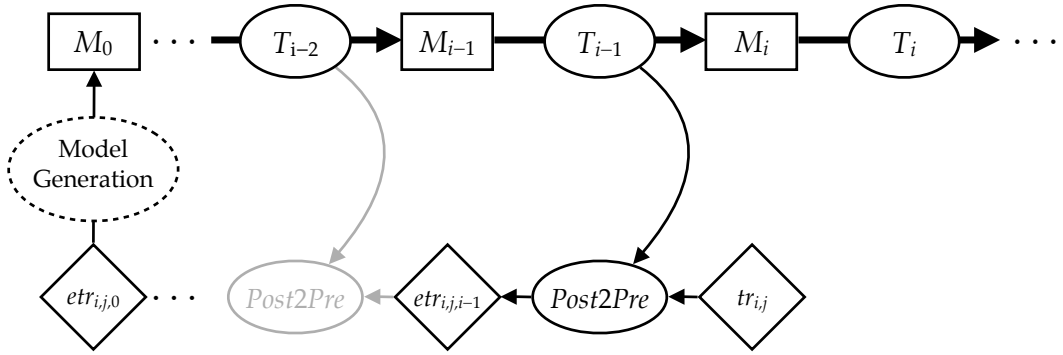



Figure 5.1: Step by step advancement of test requirements

solvers can be used to produce a test model $M_0$ that satisfies $etr_{i,j,0}$. Given that *Post*2*Pre* produces at each step a precondition ensuring the satisfaction of the post-condition, then executing the chain over $M_0$ yields an intermediate model $M_i$ that satisfies the test requirement $tr_{i,j}$. In the following paragraphs we detail this approach and demonstrate how it solves the problem.

Formally, *Post*2*Pre* should have the following property.

**Definition 5.1 (*Post*2*Pre*).** Given a model transformation $T$ from metamodel $MM_{in}$ to metamodel $MM_{out}$ and a constraint *Post* over $MM_{out}$, *Post*2*Pre* $(T, Post)$ is a constraint over $MM_{in}$ such that:

$$M_{in} \models Post2Pre(T, Post) \quad \Leftrightarrow \quad M_{out} = T(M_{in}) \models Post$$

□

Given a non-satisfied test requirement $tr_{i,j}$, applying *Post*2*Pre* iteratively as in Figure 5.1 yields the following results at each iteration:

$$\begin{aligned} etr_{i,j,i-1} &= Post2Pre\left(T_{i-1},\ tr_{i,j}\right) \\ etr_{i,j,i-2} &= Post2Pre\left(T_{i-2},\ etr_{i,j,i-1}\right) \\ &\vdots \\ etr_{i,j,1} &= Post2Pre\left(T_1,\ etr_{i,j,2}\right) \\ etr_{i,j,0} &= Post2Pre\left(T_0,\ etr_{i,j,1}\right) \end{aligned}$$

Once we reach the constraint $etr_{i,j,0}$ over the input language and apply a constraint solver, we produce a satisfying test model $M_0 \models etr_{i,j,0}$. When executing the chain over this model, the properties of *Post*2*Pre* give us the following:

$$
\begin{array}{rcll}
M_0 \models etr_{i,j,0} & \Leftrightarrow & M_1 = T_0\,(M_0) & \models etr_{i,j,1} \\
& \Leftrightarrow & M_2 = T_1\,(M_1) & \models etr_{i,j,2} \\
& & \vdots & \\
& \Leftrightarrow & M_i = T_{i-1}\,(M_{i-1}) & \models tr_{i,j}
\end{array}
$$

Executing the chain over the test model $M_0$ leads to the satisfaction of $tr_{i,j}$. We have thus demonstrated that with this approach we are theoretically able to produce integration test models to satisfy unit test requirements of intermediate steps of the chain. The problem is now to design a *Post*2*Pre* construction that satisfies Definition 5.1.

### 5.2.2 Weakest Precondition of Algebraic Graph Transformations

In our search for a satisfactory *Post*2*Pre* construction, we have turned to the notions of postconditions and preconditions. These notions were originally introduced in [Hoare, 1969] as a tool to prove the correctness of programs. Postconditions and preconditions are *predicates* or *constraints* on the state space of a program. Later [Dijkstra, 1975] introduced *predicate transformers*, in particular the *weakest precondition* transformer denoted as $wp(P, d)$ where $P$ is a program and $d$ is a postcondition. $wp$ transforms the postcondition $d$ into a necessary and sufficient precondition that must be satisfied by the initial state to guarantee the termination of $P$ and the satisfaction of the postcondition $d$ in the final state. This precondition is called the *weakest* because it is the least constraining, meaning that it encompasses *all* inputs that lead to the satisfaction of the postcondition. In these seminal works, programs consisted of assignment statements, alternative choice statements, and repetition statements manipulating numeric variables.

More recently, similar work has been done for graph manipulation programs in the theory of *Algebraic Graph Transformation* (AGT) in [Habel *et al.*, 2006a] and [Poskitt, 2013]. Instead of numeric variables, such programs manipulate typed graphs. According to the definitions in [Habel *et al.*, 2006a] given a non-deterministic graph program $P$ and a postcondition $d$ on resulting graphs the properties of the $wp$ construction are such that for all input graphs $G$ such that $G \models wp(P, d)$, we have:

1. For all possible result graphs $H$, $H \models d$.
2. There exists some result graph $H$.
3. $P$ terminates when executed with $G$ as input.

Assuming $P$ is deterministic, the above properties can roughly be simplified as follows:

$$\forall G, \quad G \models wp(P,d) \quad \Leftrightarrow \quad \exists H, \quad H = P(G) \models d$$

The above properties of $wp$ are similar to the ones we are looking for in *Post*2*Pre*. Consequently we propose to formalise the model transformations of our chain as algebraic graph transformations, and use constructions similar to the ones in the *weakest precondition* construction as the basis for the *Post*2*Pre* construction in our approach to the generation of integration tests. This proposal is depicted in Figure 5.2.

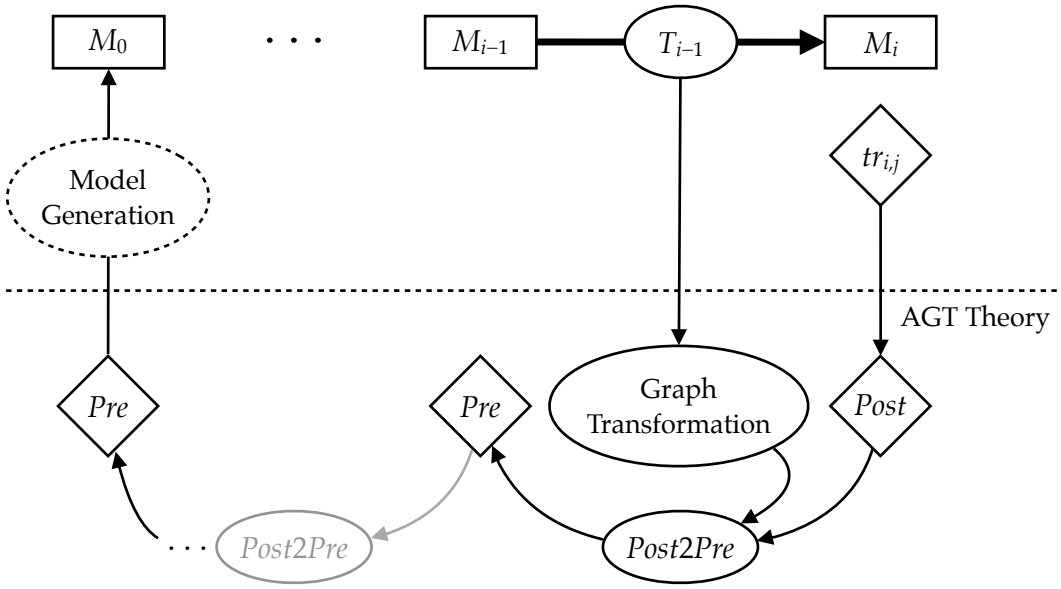



Figure 5.2: Backwards advancement of test requirements

Before we explain the details of this proposal, it is important to highlight some assumptions and choices that we make in our work.

### 5.2.3 Assumptions and Scope of the Contributions

So far we have presented our ideas with little assumptions on the nature of the transformations and their underlying formalisms. At this point it is important to formulate some assumptions and define the scope that will be addressed within this thesis.

The key to the iterative step-by-step solution that we proposed is the *Post*2*Pre* construction which deals with one step of the model transformation chain. Therefore performing this construction for one transformation unlocks the solution for the complete chain. Within this thesis and from this point forward, we focus on

defining and validating *Post*2*Pre* for *one* model transformation and not the complete chain.

In Section 2.6 we explained that the scope of our research encompasses model transformation chains involving endogenous (refinement within the same metamodel) and exogenous (translation from a metamodel to another) model transformations. At this point it is important to specify that our work on the one-step backwards translation focuses on *exogenous* model transformations. We make this choice because the formal AGT framework that we base our work upon is fundamentally endogenous making this kind of transformations easier to address. As a result we tackle the less straightforward exogenous kind of transformations.

As for the choice of languages and technologies, we select ATL [Jouault and Kurtev, 2006] for the specification of the model transformations that we consider because it is a popular language both in academia and in the industry. ATL's most common use is for exogenous transformations, but since it also supports an endogenous refinement mode, this would allow to extend our work to endogenous transformations in the future.

We select OCL [OMG, 2014] as a constraints language for test requirements because its expressiveness allows considering various sources of test requirements. OCL test requirements can be specified manually by the tester, or generated automatically based on test adequacy criteria using existing approaches detailed in Section 3.4.

Finally, we select Henshin [Biermann *et al.*, 2012] as an implementation of the theoretical AGT framework because it applies the formal AGT semantics to EMF models which are also the basis of the ATL tooling. This allows for an easy implementation and experimentation of our proposals.

Having clarified the assumptions and choices defining the scope of our work, we present our main contributions in the next section.

### 5.2.4 Contributions

We propose a translation of ATL transformations and OCL test requirements to equivalent concepts in the AGT theory where we use existing theoretical results to define *Post*2*Pre*. As depicted in Figure 5.3, ATL transformations and OCL test requirements are translated respectively to *Graph Transformations* and *Nested Graph Conditions* (NGC) which are constraints in the AGT framework that will be defined and detailed in upcoming chapters. Then *Post*2*Pre* is applied on the graph transformation and the postcondition to produce an equivalent precondition based on

the constructions of weakest preconditions in the AGT theory. We published an overview of this approach at the AMT'14 workshop in [Richa *et al.*, 2014]. Within this general approach we put forward several contributions that are detailed next.

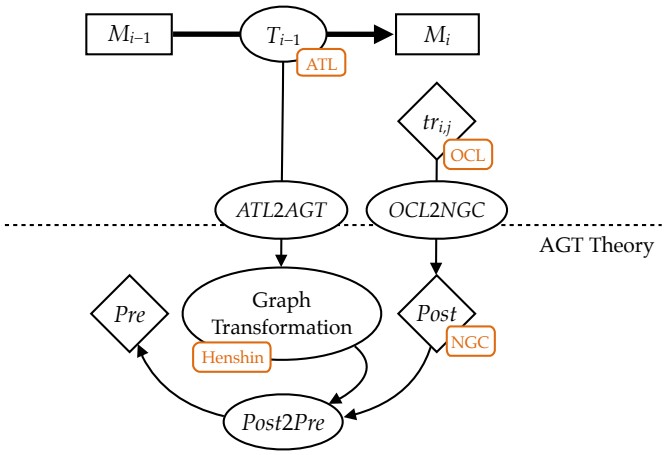



Figure 5.3: Translation to AGT and advancement of test requirements with *Post*2*Pre*

**Translation of ATL and OCL to AGT – *ATL2AGT* and *OCL2NGC***

We propose a translation of ATL transformations to AGT called *ATL2AGT*. This is an original contribution as no translations of ATL to AGT exist in the literature to the best of our knowledge.

The challenge in this work is mapping the rich semantics of ATL, namely the *resolve mechanisms*, to the simpler rewriting semantics of AGT where no such mechanisms exist. Our translation addresses this challenge by organising the AGT transformation in 2 phases similarly to the ATL execution semantics, and using explicit trace nodes to emulate the resolving mechanisms. Since ATL embeds OCL as a sub-language for constraints and queries, *ATL2AGT* relies on the translation of OCL.

*OCL2NGC* is based on an existing translation of general OCL constraints to NGC [Radke *et al.*, 2015a]. However since this and other existing translations are limited to the support of non-ordered sets, we propose an extension allowing the support of ordered sets in the context of *ATL2AGT*.

The challenge in this extension is making sure that output elements are placed in output sets in the same order as their corresponding input elements, as per the ATL semantics. Existing translations of OCL to NGC do not ensure this property because the graph matching semantics of AGT does not consider order. We address this challenge by extending the existing translations with additional NGCs dedicated to ensuring the orderly matching of elements.

In this thesis, *OCL2NGC* is presented in the context of *ATL2AGT* because our contributions on the support of ordered sets are specific to that context.

Our contributions of *ATL2AGT* and *OCL2NGC* were the subject of a peer-reviewed publication [Richa *et al.*, 2015] which received the *Best Paper Award* at the ICMT'15 conference. Both these contributions are the focus of Chapter 6.

**Translation of Postconditions to Preconditions –** *Post2Pre*

We defined *Post2Pre* based on the AGT theoretical definitions related to the weakest precondition construction [Habel and Pennemann, 2005; Habel *et al.*, 2006a; Poskitt, 2013]. Specifically, we identified the *weakest liberal precondition* (*wlp*) as the construction that is suitable for our needs. This part of our work is developed within the scope of purely structural transformations and does not yet support the manipulation of object attributes.

The first challenge in this work is that *wlp* is theoretically infinite for the kind of transformations that we analyse. To address this problem, we propose to analyse a bounded version of the transformations by introducing a new construct, *bounded iteration*, and defining its *wlp* construction which is finite. Since this result is only valid for bounded transformations, we propose a new construction called *scopedWlp* which extends the previous one and makes the result applicable to the original unbounded transformations. Depending on the desired properties of the precondition, *Post2Pre* can then consist of either *wlp* or *scopedWlp*.

For all the above new concepts, we provide formal definitions and proofs of correctness that are not limited to ATL, making them a contribution to the theory of AGT with potential applications beyond this thesis. This part of our work is the focus of Chapter 7.

**Simplification Strategies**

The *wlp* construction has a high computational complexity that prevents it from scaling for large transformations and postconditions. It is necessary to address this challenge to make the *Post2Pre* analysis feasible for ATL transformations.

We propose several simplification strategies allowing to eliminate irrelevant portions of computed precondition in order to reduce their size and avoid the combinatorial explosion of the construction. Some of these strategies are specific to ATL while others apply to arbitrary AGT transformations. We also propose a modified construction of *wlp* which is equivalent to the original one but allows an early application of the simplification strategies. This helps avoiding unnecessary computations early on. With the combination of these proposals, we are able to success-

fully perform analyses that were previously infeasible due to the high complexity. These contributions are the focus of Chapter 8.

**Implementation and Validation**

Our contributions are all implemented as components of our Java and EMF-based tool called *ATLAnalyser*[1]. The validation of our conceptual contributions and their implementation is detailed in Chapter 10.

*ATL2AGT* and *OCL2NGC* are validated by translating several ATL transformations to AGT transformations and executing both ATL and AGT versions over the same input models. We verify that the models resulting from ATL and AGT are the same, including the order of elements in ordered sets, to validate the correctness of our translation.

The correctness of *Post2Pre* is established by formal proof in, but its implementation is validated by considering AGT transformations obtained automatically with *ATL2AGT*, and postconditions written manually. The resulting preconditions are interpreted manually to confirm that they exhibit the theoretically expected properties.

And finally, the simplification strategies are first validated *functionally* by verifying that their application does not compromise the correctness of the result. Then scalability is assessed by considering larger transformations and postconditions, and evaluating the impact of our strategies on the performance of the analysis. As will be discussed in Chapter 10, despite remaining scalability issues our simplification strategies prove to be highly efficient and allow performing previously infeasible analyses.

This concludes the presentation of our approach to solving the first problem tackled in this thesis: the backward translation of unit test requirements through a model transformation chain. We proposed a step-by-step iterative analysis and focused specifically on one step of the analysis. The iteration of this analysis allows to translate the unit test requirement into an equivalent constraint over the input language of the chain, where a satisfactory integration test can finally be generated. Determining the verdict of such integration tests with suitable test oracles is the focus of second problem tackled in this thesis. We develop the solution that we propose in the next section.

[1] *ATLAnalyser*, `https://github.com/elericha/atlanalyser`

## 5.3 Syntactic Specification of Model-to-Code Transformation

The second problem tackled by this thesis is devising a test oracles approach for the integration tests of an ACG in the context of its qualification. We determined that for the needs of qualification, the test oracles need to be based on a specification of the ACG focusing on the *syntax* of the generated code. Readability was an important factor to consider because that specification formalises Tool Operational Requirements (TORs) which are part of the qualification evidence of the tool. Thus we determined that we need a way to specify the generated code in terms of its textual concrete syntax which is its most common and widely known form.

In the following we give an overview of our proposed approach. As previously indicated, the solution is largely specific to the QGen Simulink to C code generator, as requested by AdaCore, the industrial partner of this thesis. However the core principles that we propose can be generalised to arbitrary model-to-text transformations. Even though this generalisation is not performed in this thesis, we provide hints about it where relevant.

### 5.3.1 Core Principle: Specification Templates

We propose to consider code generation as a general model-to-text transformation and propose a specification approach based on the notion of *specification templates* to define the generated textual artifacts. A specification template is composed of the following elements:

1. Input elements: objects of the input metamodel of the transformation
2. A guard: a constraint over the input elements defining when the template is applicable
3. A pattern of text that should be generated when the guard is satisfied: the pattern of text is an arbitrary concatenation of the following four kinds of content:
   a. verbatim text
   b. queries to the input model (enclosed in brackets `[` and `/]`)
   c. regular expressions (enclosed in `%<` and `>%`)
   d. repetition statements (`[for ...]` statements)

Concretely, a specification template is represented as follows:

Listing 5.1: General structure of a specification template

```
1 [template templateName (inputElement : inputType) ? (guard)]
2 verbatim text, interleaved with [inputModelQueries/] between brackets,
3 %<regularExpressions>% between percent delimiters
4 and loop statements expressing repeating patterns:
5 [for ( iterator | collection )]
6   This text repeats for all elements of the collection.
7   We can use [queriesInvolvingIterator] here.
8 [/for]
9 [/template]
```

The semantics of a specification template is that for each element of the input model that satisfies the guard of the template, a portion of text corresponding to the specified pattern should be generated at some location in the generated artifacts. A specification template is therefore a simple positive constraint requiring the existence of a pattern of text in the generated artifacts. This constraint may be simply expressed as follows:

$$\textit{application condition} \quad \Rightarrow \quad \exists\,(\textit{textual pattern})$$

where the application condition is defined by the input elements of the specification template and its guard, while the textual pattern is described by the content of the specification template.

In our specific application to Simulink to C code generation, the location in the generated artifacts where the pattern of a specification template should be generated is hard coded with a simple naming convention. This is because the C code generated for a Simulink model has a fixed structure with well defined separate code sections. The name of a specification template indicates to which code section it applies.

This aspect of the approach is evidently specific to our code generator however we believe it is possible to generalise the proposal to arbitrary model-to-text transformations by introducing other specification constructs that specify the structure of generated artifacts (*i.e.* folders and files) and specify where each specification template should match in that structure. Further constructs could also be introduced for example to define *negative* specification templates that *prohibit* patterns of text instead of requiring their existence. We believe that such a generalisation would allow the specification of arbitrary model-to-text transformations, however this is not done in this thesis.

Next we explain how the proposed specification templates can be used as automatic test oracles.

### 5.3.2 Automatic Test Oracles

Specification templates are executable. Figure 5.4 illustrates how the execution of specification templates constitutes an automatic specification-based test oracle. Given an input test model, each specification template is executed only for input elements that satisfy its guard. The template is executed by evaluating the queries it contains and replacing them with the evaluation results in the textual pattern. This yields a so-called *expected pattern* composed of verbatim text (where queries have been replaced by query results) and regular expressions. Essentially the expected pattern is a large regular expression that should be matched in the actual output of the tested transformation. An expected pattern and the actual output are given as input to a *Matcher* which looks for a match of the expected pattern in the actual output. The delimiters `%<` `%>` of regular expressions allow the *Matcher* to *escape* verbatim text to prevent special characters such as `*` and `?` from being interpreted as regular expression commands outside of the delimiters.

The execution of a complete specification over an input test model yields multiple expected patterns, one per executed specification template. Each expected pattern should have a match in the actual output of the tested transformation. If all expected patterns have a match, the test passes, otherwise the test fails.

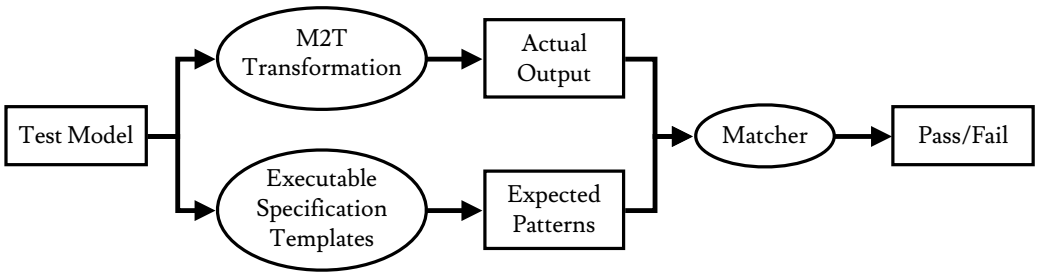



Figure 5.4: Specification-based test oracles for model-to-text transformations

As mentioned earlier, in the context of our Simulink to C code generator, the structure of the generated code is fixed and the location where a specification template should match is predefined based on a naming convention. In a generalisation of this approach the structure of the generated artifacts (*i.e.* folders, files, file names *etc.*) and the location of specification templates in that structure would be part of the specification and the matching operation would be more complex.

Having presented the aspects of specification and automatic oracles of our approach, we now discuss the aspect of readability of the specification and its role as qualification evidence.

### 5.3.3 Readability and Generation of Qualification Documents

From qualification viewpoint, we propose to consider each specification template as a Tool Operational Requirement (TOR). This gives requirements a uniform structure:

1. Scope of the requirement (input of the template and guard)
2. Required output (the textual source code pattern)

Additionally, we've already emphasized that exhibiting the code generation patterns in the well known concrete textual syntax of the source code language makes the specification understandable by a wider audience. As for queries to the input model, they can be expressed in an Object-Oriented query language (such as OCL) which is also an intuitive and widely known notation. Regular expressions can be encapsulated in helper functions of the query language to avoid exposing them in requirements. In fact the regular expressions used in code generation requirements often consist of similar forms of expressions which makes their factorisation into reusable helper functions convenient and even necessary.

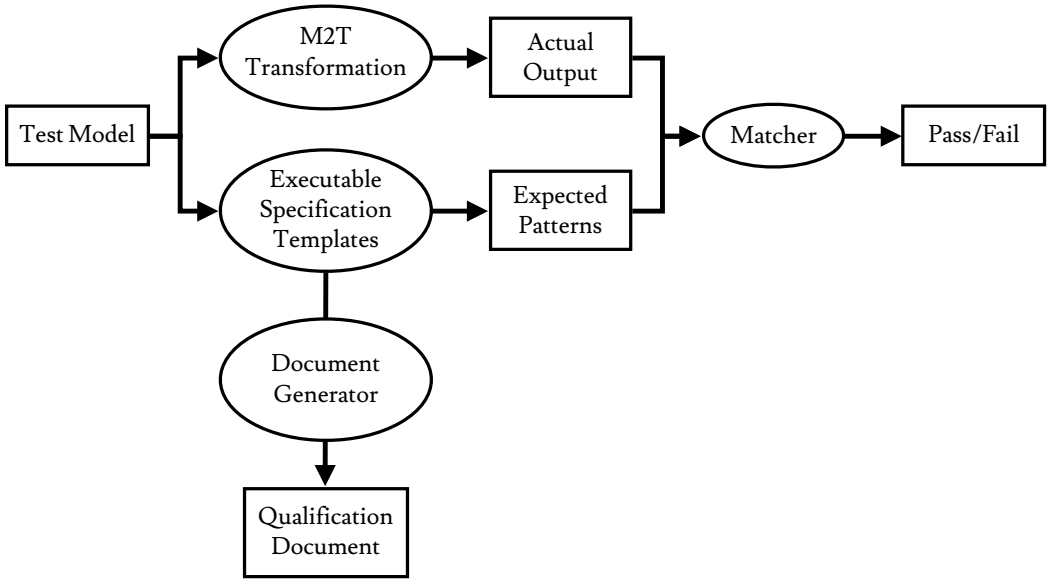



Figure 5.5: Generation of qualification documents

Additionally, thanks to the well defined structure of specification templates, we propose to translate specification templates into a document format such as

Sphinx[2] or LaTeX[3] or any other document management system used for the qualification evidence, as shown in Figure 5.5. This enhances the readability of TORs during review and eases their integration with the rest of the qualification documents.

### 5.3.4 Implementation Technology: Acceleo

We have selected Acceleo[4] as a template language and execution framework for specification templates. Acceleo is a model-to-text transformation language based on templates and in fact the representation we proposed in Listing 5.1 already uses the Acceleo syntax. The guards of Acceleo templates and the queries to the input models (between brackets `[, /]`) are expressed in OCL. Additionally, Acceleo allows the definition of reusable *queries* which can be invoked in the OCL expressions constituting guards and model queries which allows to encapsulate common regular expressions or specification patterns and enhance the readability of the requirements.

From a tooling perspective, Acceleo provides a template editor in the Eclipse environment, with syntax highlighting and auto-completion features that help support the requirements writing process.

The *Matcher* of expected patterns with actual test output was implemented in Python using a standard regular expressions library. As mentioned earlier, the Matcher also implements hard coded logic to determine the location where an expected pattern should be matched based on a convention specific to our Simulink to C code generator.

### 5.3.5 Contributions

Our contributions in the scope of model-to-text specification and test oracles are summarised as follows. They will be discussed in more detail in Chapter 9.

#### Conceptual Contributions

1. A model-to-text specification approach based on the notion of *specification templates* which specify the generated text in terms of verbatim text, queries to the input model, regular expressions and repetition statements.

[2] Sphinx, `http://sphinx-doc.org/`
[3] `https://www.latex-project.org/`
[4] Acceleo, `https://www.eclipse.org/acceleo`

2. Specification-based automatic test oracles relying on the execution of specification templates over an input test model, and the matching of resulting *expected patterns* in the actual output of the tested transformation.

**Implementation and Validation**

The implementation of our approach is specific to the qualification of QGen: the Simulink to C code generator developed at AdaCore. Acceleo was used as a host language and execution framework for our specification templates, and a minimal *Matcher* of expected patterns was implemented. The document generation component was implemented as a higher-order transformation written in Acceleo, and taking as input Acceleo specification templates.

The above components of our implementation were integrated in a tool called *TOR Toolkit* that was deployed within AdaCore's development team for assessment of the approach. The outcome of this assessment will be detailed in Chapter 10.

## 5.4 Conclusion

In this chapter we have given an overview of the contributions of this thesis. The first set of contributions addresses the first problem tackled by this thesis which is the backward translation of non-satisfied test requirements up to equivalent constraints over the input language of a model transformation chain. We propose to perform this translation step-by-step with an iterative analysis called *Post*2*Pre* that considers the non-satisfied test requirement as a postcondition of the preceding transformation and translates it into an equivalent precondition. The analysis can thus be iterated for all preceding steps of the chain until a constraint over the input language is reached.

Focusing on one iteration of the backward translation, we propose to define *Post*2*Pre* in the theoretical framework of *Algebraic Graph Transformation* (AGT) where existing work proposes translations of postconditions to preconditions. Assuming transformations are specified in ATL and test requirements in OCL, we propose translations $ATL2AGT$ and $OCL2NGC$ to transpose the problem into the AGT theory. Then the *Post*2*Pre* backward translation is defined using the existing constructions of *weakest liberal preconditions* ($wlp$) in AGT. Given the computational complexity of $wlp$, simplification strategies are provided to reduce the complexity and avoid unnecessary computations.

After this first set of contributions that work towards producing integration tests, we tackle the second problem of this thesis which is determining the verdict

of these integration tests in the case of an ACG. Given the context of qualification, we need test oracles based on a syntactic specification of the ACG in terms of the textual concrete syntax of the generated code.

We propose an approach based on the notion of specification templates where the generated code is specified in terms of textual patterns composed of verbatim text, queries to the input model, regular expressions and repetition statements. Executing specification templates over test models provides expected patterns which should be matched in the test output, thus constituting an automatic test oracle. As for qualification needs, specification templates are amenable to document generation which enhances their readability and eases their integration with other qualification documents.

The next chapters detail our proposals and the challenges that they tackle. First, *ATL2AGT* and *OCL2NGC* are the focus of Chapter 6. Then, the theoretical aspects of *Post2Pre* are detailed in Chapter 7 while the simplification strategies addressing its scalability are presented in Chapter 8. The syntactic specification and test oracles approach is detailed in Chapter 9. Finally, the experimental validation and assessment of all our contributions is presented in Chapter 10.

# Chapter 6

# Translating ATL Transformations to Algebraic Graph Transformations

## Contents

## 6.1 Introduction

This chapter details the first contribution of this thesis, the translation of ATL transformations to Algebraic Graph Transformations with equivalent semantics. As will be explained, we address two challenges in this work: the translation of the ATL resolve mechanisms which do not have an equivalent in the AGT semantics, and the translation of OCL constraints and queries of ATL rules into Nested Graph Conditions in AGT. To solve the first challenge, we propose to use a 2-phase AGT transformation with explicit trace nodes. To solve the second challenge we reuse and extend existing translations of OCL to NGC. Since these approaches only support non-ordered sets, we supplement them with support for ordered sets in the context of the ATL translation.

The chapter is organised as follows. First we give an overview of the semantics of ATL and AGT in sections 6.2 and 6.3, and then discuss the challenges of translating the former into the latter in Section 6.4. In Section 6.5 we tackle the first part of our translation which deals essentially with the ATL resolve mechanisms. In Section 6.6 we tackle the second aspect, the translation of OCL to NGC, recalling first the general principles of existing translations and then presenting our contribution regarding the support of ordered sets. Section 6.7 discusses the implementation of our proposals and finally, Section 6.8 recalls relevant related work.

## 6.2 Semantics of ATL

ATL [Jouault and Kurtev, 2006] is a model-to-model transformation language combining declarative and imperative approaches in a hybrid semantics. ATL transformations are primarily *out-place*, *i.e.* they produce an output model different from the input model (though both may be in the same language), and a so-called *refining mode* allows for *in-place* model refinement transformations. In the scope of our work, we focus only on the declarative features of ATL in the standard *out-place* mode.

An ATL transformation consists of a set of declarative *matched rules*, each specifying a *source pattern* (the `from` section) and a *target pattern* (the `to` section). The source pattern is a set of objects of the input metamodel and an optional OCL [OMG, 2014] constraint acting as a *guard*. The target pattern is a set of objects of the output metamodel and a set of *bindings* that assign values to the attributes and references of the output objects. For example in Figure 6.1, `R1` has one source pattern element `s` and two target pattern elements: `t1` with 3 bindings and `t2` with 1 binding.

```
1  rule R1 {
2  from s  : IN!A
3   (s.refB->exists(b | b.name = 'Hodor'))
4  to t1 : OUT!D
5     (name <- s.name + '1',
6      refD <- t2,
7      refE <- s.refB),
8   t2 : OUT!D
9     (name <- s.name + '2') }
10
11
12
13 rule R2 {
14 from s : IN!B
15 to t : OUT!E
16    (refD <- thisModule.resolveTemp(s.refA, 't2') ) }
```

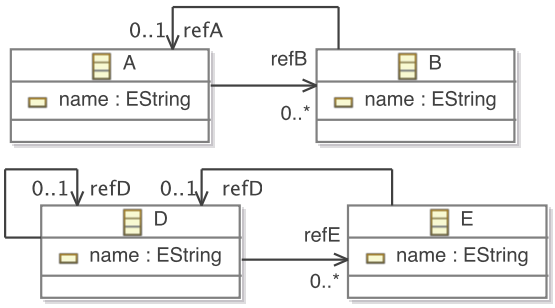



Figure 6.1: Example of ATL transformation

An ATL transformation is executed in two phases. First, the matching phase searches in the input model for objects matching the source patterns of rules (*i.e.* satisfying their filtering guards). A tuple of source objects is only allowed to match with one rule. For this reason ATL rules always have either source patterns with different types of elements, or non-overlapping guards to ensure that input elements only match one rule. For each match of a rule's source pattern, the objects specified in the target pattern are instantiated. Then in a second stage, the target elements' initialization phase executes the bindings for each triggered rule.

A *binding* defines a *target property* which is an attribute or a reference on the left side of the `<-` symbol, and an OCL query on the right side of the symbol. A binding maps a scalar value to a target attribute (line 5), target objects (instantiated by the same rule) to a target reference (line 6), or source objects to a target reference (line 7). In the latter case, a *resolve* operation is automatically performed to find the rule that matched the source objects, and the *first* output pattern object created by that rule is used for the assignment to the target reference. This is referred to as the *default resolve mechanism*. For example in Figure 6.1, the binding at line 7 resolves the objects in **`s.refB`** into the output objects of type **`E`** created by **`R2`**, and assigns them to **`t1.refE`**.

Another *non-default resolve mechanism* allows resolving a (set of) source object(s) to an arbitrary target pattern object instead of the first one as in the default mechanism. It is invoked via the following ATL standard operations:

```
thisModule.resolveTemp(obj, tgtPatternName)
thisModule.resolveTemp(Sequence{obj1, ...}, tgtPatternName)
```

The former is used to resolve with rules having one source pattern element while the latter is used to resolve with rules having multiple source pattern elements. For

example, the execution of the binding on line 16 in rule `R2` will retrieve the target object `t2` (instead of `t1` as with the default resolve) that was created by `R1` when it matched `s.refA`.

## 6.3 Semantics of AGT

*Algebraic Graph Transformation* (AGT) [Ehrig *et al.*, 2006] is a formal framework that provides mathematical definitions to model graph transformations. We will be using the *Henshin* [Henshin, accessed 2015] graph transformation framework which applies the theoretical semantics to standard EMF models in the Eclipse platform. The details of the formal foundations of Henshin can be found in [Biermann *et al.*, 2012] and are only briefly recalled here. A graph transformation is composed of two main elements: a set of *transformation rules*, and a *high-level program* defining the sequencing of rules.

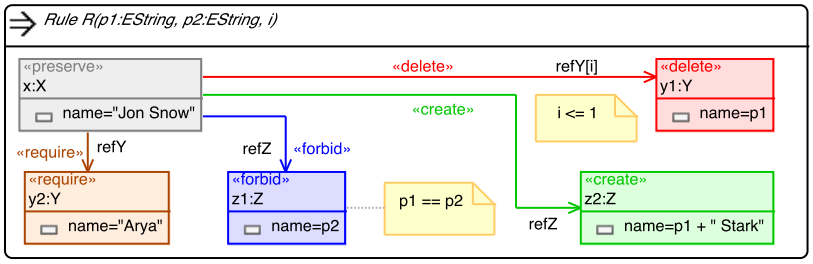



Figure 6.2: Henshin graphical representation of an AGT rule

An AGT rule consists of a Left-Hand Side ($LHS$) graph and a Right-Hand Side ($RHS$) graph both depicted on the same diagram as in Figure 6.2. $LHS$ elements are annotated with «*preserve*» or «*delete*» while $RHS$ elements are annotated with «*preserve*» or «*create*». Roughly, a rule is executed by finding a match of $LHS$ in the transformed graph, deleting the elements of $LHS - RHS$ («*delete*»), and creating the elements of $RHS - LHS$ («*create*»). Elements of $LHS \cap RHS$ are preserved («*preserve*»). A rule transforms elements matched by the $LHS$ into the $RHS$, therefore an AGT is an *in-place* rewriting of the input model. For example, rule $R$ in Figure 6.2 matches nodes $x$ of type $X$ and $y1$ of type $Y$ and edge $refY$ in the transformed graph, deletes the node matched by $y1$ and the edge matched by $refY$, and creates node $z2$ of type $Z$ and the edge $refZ$.

Matches of a rule may be restricted with additional constraints by assigning *attribute values* to nodes. For example the rule in Figure 6.2 can only match an object $x$ when $x.name$ = "Jon Snow". Moreover, attribute values may be stored in *rule parameters* such as in $y1.name = p1$ where the *name* attribute of the object matched by $y1$ is stored in the rule parameter $p1$. Finally, a rule may assign new

values to attributes such as in $z2$ where $z2.name$ is initialized to $p1$ concatenated to the string " Stark".

In Henshin, edges typed by a *multi-valued ordered reference* (*i.e.* with upper bound higher than 1) can be labeled with an *index*. This feature will play an important role in the handling of the ATL resolve mechanisms and the support of ordered sets in Section 6.6. A literal integer index such as $ref[2]$ represents a matching constraint: only the object at index 2 may be matched by the rule. A rule parameter index such as $ref[i]$ allows to read an object's index in the ordered reference and store it in the parameter. For example in Figure 6.2, $refY[i]$ indicates that the index of $y1$ is stored in $i$. Edge indexes are zero-based.

An AGT rule can have an *application condition* (AC) which constrains its possible matches. An AC is a *Nested Graph Condition* (NGC) over the $LHS$. Formally, a NGC over a graph $P$ is of the form *true* or $\exists(a \mid \gamma, c)$ where $a : P \hookrightarrow C$ is an injective morphism, $\gamma$ is a boolean expression over rule parameters and $c$ is a NGC over $C$. A match $p : P \hookrightarrow G$ of $P$ in a graph $G$ satisfies an AC $\exists(a \mid \gamma, c)$ if there exists a match $q : C \hookrightarrow G$ of $C$ in $G$ such that $p = q \circ a$ and $\gamma$ evaluates to *true* under the parameter assignment defined by $p$ and $q$ satisfies $c$. Boolean formulas can be constructed such as the negation $\neg c$ , the conjunction $\bigwedge_i c_i$ and the disjunction $\bigvee_i c_i$ of NGCs $c_i$ over $P$. We use short notations $\forall(a, c)$ and $c_1 \implies c_2$ for $\neg\exists(a, \neg c)$ and $\neg c_1 \vee c_2$ respectively. For example the AC in Figure 6.3 defined for rule $R$ requires the existence of a node $y2$ whose *name* attribute is *"Arya"* and forbids the existence of a node $z1$ with the same *name* as $y1$. The boolean expression $i <= 1$ constrains the rule to match only for the first two objects in the ordered reference $x.refY$. Note that $P$ is omitted from the notation when it can be inferred from the context, and so are $\gamma$ and $c$ when they are *true*. The AC is graphically represented in Figure 6.2 using the annotations «*require*» and «*forbid*», however this is only possible for one level of nesting in the AC. For complete NGCs the full notation of Figure 6.3 is necessary. In Section 6.6 we will translate OCL guards and bindings into suitable ACs of AGT rules.

$$\exists\left(\boxed{x : X} \xrightarrow{refY} \boxed{\begin{array}{c} y2 : Y \\ \hline name = \text{"Arya"} \end{array}} \mid i <= 1\right) \wedge \neg\exists\left(\boxed{x : X} \xrightarrow{refZ} \boxed{\begin{array}{c} z1 : Z \\ \hline name = p2 \end{array}} \mid p1 = p2\right)$$

Figure 6.3: Example of a Nested Graph Condition

Finally an AGT transformation is defined by a so-called *high-level program* which specifies in which order AGT rules are applied. A program can be:

1. elementary, consisting of a rule $r$,
2. the sequencing of two programs $P$ and $Q$ denoted by $(P; Q)$, or

3. the iteration of a program $P$ as long as possible, denoted by $P\downarrow$, which is equivalent to a sequencing $(P;(P;(P\cdots)$ until the program $P$ can no longer be applied.

## 6.4 Challenges of the Translation

Having presented ATL and AGT, we now tackle the translation of the former into the latter. To avoid confusion, we will use the notation "$\text{rule}_{\text{ATL}}$" to denote ATL rules, and "$\text{rule}_{\text{AGT}}$" to denote AGT rules. There are several challenges to the translation of ATL to AGT:

1. The first challenge is that the ATL transformations we consider are *out-place* whereas AGT transformations are *in-place*. Therefore an ATL transformation will have to be translated as a rewriting of the input model (or rather a copy of it in order to preserve the input model intact).

2. The second challenge is dealing with the ATL resolve mechanisms. In AGT no such mechanisms exist, and any objects that a $\text{rule}_{\text{AGT}}$ needs to use must already exist in the transformed graph and must be matched by the $\text{rule}_{\text{AGT}}$'s $LHS$. If a $\text{rule}_{\text{AGT}}$ $R1$ needs to use an object created by $\text{rule}_{\text{AGT}}$ $R2$, then $R2$ must be executed before $R1$. But as demonstrated by the example ATL transformation in Figure 6.1, $\text{rules}_{\text{ATL}}$ can mutually use objects created by each other. This case cannot be solved with simple $\text{rule}_{\text{AGT}}$ sequencing and therefore a more complex scheme is required. Moreover, the non-default resolve mechanism of ATL requires to relate output objects to output pattern identifiers so that we can retrieve the object corresponding to a specific output pattern identifier given as argument to the `resolveTemp` operation.

3. The third challenge is translating OCL expression in $\text{rule}_{\text{ATL}}$ guards and bindings into graphs and NGCs in AGT. Several works [Arendt *et al.*, 2014; Radke *et al.*, 2015b; Bergmann, 2014] have tackled this challenge and we have based our translation on these existing works. However in all existing works ordered sets are not supported, because the AGT frameworks used in these works do not provide the semantic concepts necessary to deal with ordering. This lead us to tackle this particular aspect using the feature of edge indexing of the AGT framework that we are using.

In the following we introduce our translation of ATL to AGT. The next section focuses on the first two challenges while Section 6.6 addresses the third challenge separately.

## 6.5 Translating ATL to AGT

This section details our proposed translation of ATL transformations to AGT transformations. We will use the example ATL transformation of Figure 6.1 to illustrate the translation as it is developed.

### 6.5.1 General Translation Scheme

Given the *out-place* nature of the ATL transformations we consider and *in-place* nature of AGT we propose to model the ATL transformation in AGT as a refinement of the input model which only adds the elements of the output model without modifying the input elements. The AGT transformation is organised similarly to the ATL execution semantics, as two main sequential phases: an *instantiation* phase followed by a *resolving* phase. Moreover, we introduce *trace nodes* that maintain the relationship between input and output elements. Finally, we add an optional *cleanup* phase which deletes all input elements and trace nodes leaving only the output model.

$$T_{AGT} = Instantiation\ ;\ Resolving\ ;\ Cleanup$$

The first phase applies a sequence of *instantiation rules*$_{AGT}$ that create output objects without initializing their attributes and references, and relate them to input objects through *trace nodes*. Each rule$_{\text{ATL}}$, *e.g.* `R1` from Figure 6.1, yields one instantiation rule$_{\text{AGT}}$ $R1_{Inst}$ that matches the same objects as `R1`. $R1_{Inst}$ is iterated as long as possible so that all matches in the input model are processed. The order of application of instantiation rules$_{\text{AGT}}$ is irrelevant as they do not interfere with each other since objects are allowed to match for only one rule$_{\text{ATL}}$, as per the ATL semantics.

The second phase of the transformation applies a set of *resolving rules*$_{AGT}$ which initialize references and attributes of output objects. Each binding in a rule$_{\text{ATL}}$ is translated to one or more resolving rules$_{\text{AGT}}$ as will be discussed shortly. For example, `R1` yields 4 resolving rules$_{\text{AGT}}$ $R1^{t1,name}_{Res}$, $R1^{t1,refD}_{Res}$, $R1^{t1,refE}_{Res}$ and $R1^{t2,name}_{Res}$. Resolving rules$_{\text{AGT}}$ navigate the input model and rely on the trace nodes created in the instantiation phase to perform the resolving and retrieve the corresponding output objects if needed. Like instantiation rules$_{\text{AGT}}$, resolving rules$_{\text{AGT}}$ are also iterated as long as possible so that bindings are applied to all output objects.

The third phase of the transformation applies two *cleanup rules*. The first rule *DeleteInObjs* deletes all objects of the input model, and the second rule *DeleteTra-*

*ceNodes* deletes all trace nodes. Iterating both rules yields a final model containing only the output elements.

The resulting AGT transformation is the following:

$$T_{AGT} = R1_{Inst}\downarrow;\ R2_{Inst}\downarrow;\ R1^{t1,name}_{Res}\downarrow;\ R1^{t1,refD}_{Res}\downarrow;\ R1^{t1,refE}_{Res}\downarrow;\ R1^{t2,name}_{Res}\downarrow;\ R2^{t,refD}_{Res}\downarrow;\ DeleteInObjs\downarrow;\ DeleteTraceNodes\downarrow$$

This scheme addresses the highlighted challenges (1) and (2) regarding the translation of the out-place ATL semantics to the in-place semantics of AGT, and the handling of ATL resolve mechanisms: separating the creation of output objects from their use allows resolving $\text{rules}_{\text{AGT}}$ to use any output object even in the case of mutual resolve dependencies. Moreover, the trace nodes maintain the information required to perform the resolving as explained next.

### 6.5.2 Translating the ATL Resolve Mechanisms

#### Trace Nodes

Our translation supports ATL resolve mechanisms using explicit trace nodes that conform to a *trace metamodel* generated specifically for each ATL transformation. We assume that both the input and output metamodels define a root abstract metaclass from which all other metaclasses inherit directly or transitively[1] and refer to them respectively as *RootIn* and *RootOut*. The trace metaclasses are produced as follows. First, an *abstract* metaclass *Trace* is defined with a *from* reference to *RootIn* and a *to* reference to *RootOut* (Figure 6.4). For each $\text{rule}_{\text{ATL}}$, *e.g.* `R1`, a so-called *typed trace* metaclass named *R1_Trace* inheriting the abstract *Trace* metaclass is created. For each input and output pattern element of the $\text{rule}_{\text{ATL}}$, a reference with the same name is created from the typed trace to the type of the pattern element. For `R1` this yields references *s*, *t1* and *t2* in Figure 6.4.

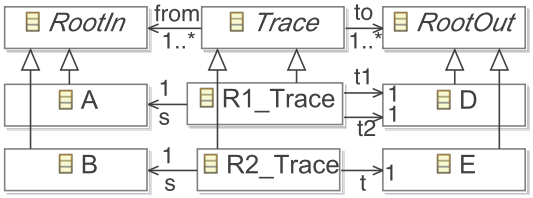



Figure 6.4: Trace metamodel

[1] if it is not the case, such a root abstract metaclass can be added automatically

### Instantiation Rules$_{AGT}$

Each rule$_{ATL}$, `R1` for example, yields one instantiation rule$_{AGT}$, $R1_{Inst}$, which matches the same objects as `R1` and creates the output objects as well as a typed trace node. As can be seen in Figure 6.5, the instantiation rule$_{AGT}$ is constructed by creating a *«preserve»* node for each input pattern element (node $s : A$). Then the OCL rule$_{ATL}$ guard is translated to an AC as per Section 6.6. This yields the *«require»* navigation to node $b : B$ with $name = p1$ and $p1 = "Hodor"$. Then, a *«create»* node is created for each output pattern element (nodes $t1 : D$, $t2 : D$) as well as a typed trace node ($tr : R1\_Trace$). The trace node is connected to input nodes with generic *from* references and typed references ($s$) and to output node with generic *to* references and typed references ($t1$ and $t2$). The order of input and output pattern elements is preserved in *from* and *to* references by indexing the created edges accordingly ($from[0]$, $to[0]$ and $to[1]$). This will allow resolve rules$_{AGT}$ to retrieve the first output object ($to[0]$) for the default resolve mechanism or any arbitrary output object ($t1$ or $t2$) for the non-default resolve mechanism.

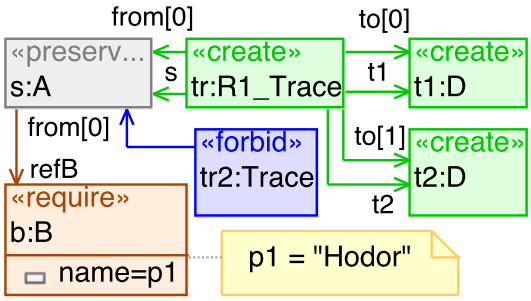
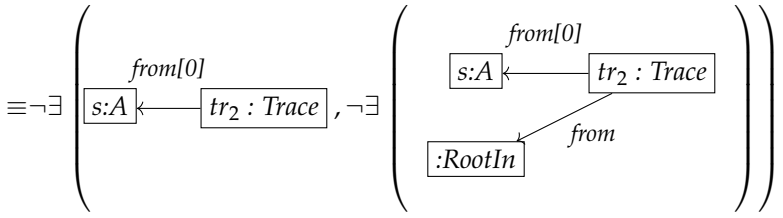



Figure 6.5: Instantiation rule$_{AGT}$ $R1_{Inst}$

Finally, since a rule$_{ATL}$ only applies once per match, we add a negative AC $applyOnce_{R1}$ preventing the application of the rule$_{AGT}$ if *another* trace node $tr_2$ with the exact same *from* elements already exists. That AC is as follows:

$$applyOnce_{R1} = \neg\exists \left( \overset{from[0]}{s{:}A \longleftarrow tr_2 : Trace}, \neg\exists \left( \overbrace{\overset{from}{:RootIn \longleftarrow tr_2 : Trace}}^{exactFrom(tr_2)} \right) \right) \tag{6.1}$$

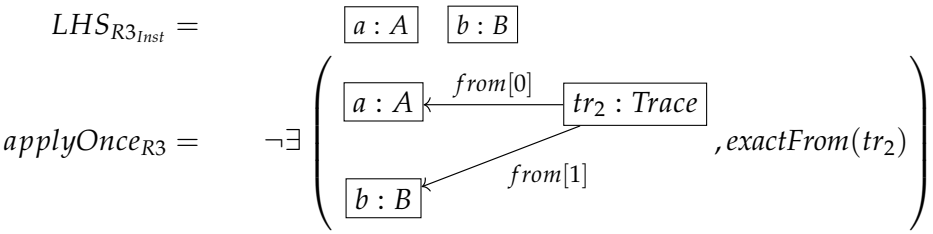



(6.2)

To understand the meaning of *exactFrom* (not visible on Figure 6.5) we have shown in (6.2) a less succinct form of the AC with equivalent semantics. $exactFrom(tr_2)$ means that $s : A$ should be the only node in $tr_2$. This is needed to express the fact that $s : A$ is allowed to participate in another $\text{rule}_{\text{ATL}}$ if there are other objects in the source pattern, *i.e.* the complete set of *from* elements is not exactly the same. To illustrate this situation let us add to the transformation the following third $\text{rule}_{\text{ATL}}$:

```
1 rule R3 {
2 from a : IN!A,
3      b : IN!B
4 to  e : OUT!E }
```

Objects of type $A$ can now match both in **R1** and in **R3**. However this does not violate the fact that ATL should be independent because the object matches in **R1** on its own and in **R3** with another object of type **B**. The role of *exactFrom* in $applyOnce_{R1}$ is to allow this behavior. Likewise, *exactFrom* appears also in $R3_{Inst}$ as follows:




*exactFrom* is reused for resolving $\text{rules}_{\text{AGT}}$ in the following sections.

### Resolving $\text{Rules}_{\text{AGT}}$ with Default Resolving

Each binding in a $\text{rule}_{\text{ATL}}$ is translated to at least one resolving $\text{rule}_{\text{AGT}}$. A resolving $\text{rule}_{\text{AGT}}$ matches the same elements as the OCL query of the binding, performs resolving if needed, and initializes the target attribute/reference of the binding. Let us consider a binding of the following general shape:

*tgtObj* : *tgtType* ( *tgtProp* <- *oclQuery* )

A binding involving default resolving or no resolving at all is translated to a resolving $\text{rule}_{\text{AGT}}$ $R_{Res}^{tgtObj,tgtProp}$ according to the algorithm presented in Table 6.1.

The translation depends on the type of the target property *tgtProp* hence the tabular presentation. Note that multi-valued target attributes are not supported at the current stage.

<table>
<tr><td></td><td colspan="3">Binding tgtObj : tgtType ( tgtProp <- oclQuery )</td></tr>
<tr><td></td><td>Single-valued Attribute<br>tgtAtt ≡ tgtProp</td><td>Single-valued Reference<br>tgtRef ≡ tgtProp</td><td>Multi-valued Reference<br>tgtRef ≡ tgtProp</td></tr>
<tr><td>Step 1</td><td colspan="3">Initialize LHS with $\boxed{\text{: <ruleName>\_Trace}} \xrightarrow{tgtObj} \boxed{tgtObj : tgtType}$</td></tr>
<tr><td rowspan="2">Step 2</td><td colspan="3">Translate oclQuery as per Section 6.6. This will complement the LHS with the required navigations and ACs and return a result.</td></tr>
<tr><td>Result is an expression expr over $\text{rule}_{\text{AGT}}$ parameters</td><td>Result is a node $\boxed{qNode}$ representing the query result</td><td>Result is a node $\boxed{qNode}$ representing one element of the result set</td></tr>
<tr><td rowspan="2">Step 3</td><td rowspan="2">Not Applicable.<br>Step 3 is specific to reference target properties.</td><td colspan="2">If the node is a source model element, perform a default resolve by matching a trace node with the exact from object using the following in the LHS:<br>$\boxed{qNode} \xleftarrow{from[0]} \boxed{tr:Trace} \xrightarrow{to[0]} \boxed{rNode:type(tgtRef)}$<br>and the AC $exactFrom(tr)$</td></tr>
<tr><td colspan="2">If not, let $\boxed{rNode} \equiv \boxed{qNode}$</td></tr>
<tr><td>Step 4</td><td>Create the following attribute in the RHS<br>$\boxed{\begin{array}{c} tgtObj \\ \hline tgtAtt = expr \end{array}}$</td><td colspan="2">Create $\boxed{tgtObj} \xrightarrow{tgtRef} \boxed{rNode}$ in the RHS</td></tr>
<tr><td rowspan="2">Step 5</td><td colspan="3">Add a negative AC to force the application of the rule once per match</td></tr>
<tr><td>$\neg\exists \left( \begin{array}{c} \boxed{\begin{array}{c} tgtObj \\ \hline tgtAtt = p1 \end{array}} \\ p1 = expr \end{array} \right)$</td><td colspan="2">$\neg\exists \left( \boxed{tgtObj} \xrightarrow{tgtRef} \boxed{rNode} \right)$</td></tr>
</table>

Table 6.1: Translation of an ATL binding with default resolving

Figure 6.6 shows the steps of the translation of binding `t1:D (refE <- s.refB)` in `R1` (Figure 6.1) to $\text{rule}_{\text{AGT}}$ $R1^{t1,refE}_{Res}$. Note that $to[0]$ in *Step 3* allows to retrieve the first target pattern element as per the default resolve semantics. Moreover, for multivalued target references such as `t1.refE`, the translation is a sort of a *flattening* whereby the result elements of the OCL query `s.refB` are not handled all at once but one by one. Each application of $R1^{t1,refE}_{Res}$ matches one element in `s.refB` and

appends the corresponding output object to the target reference `t1.refE`. However, since there are no guarantees in AGT on the order in which elements are matched, $R1^{t1,refE}_{Res}$ as presented in Figure 6.6 is only correct if `refB` is a non-ordered reference. This will be detailed and addressed in Section 6.6.

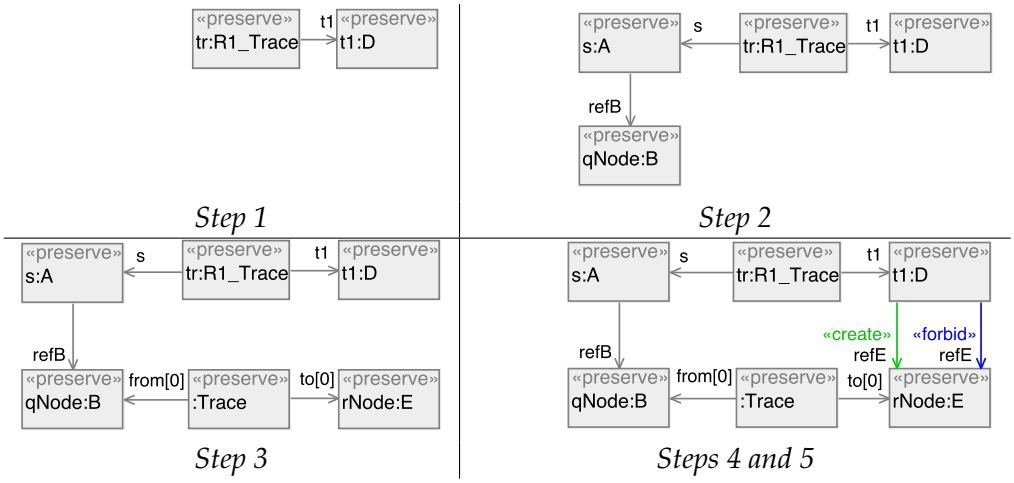



Figure 6.6: Construction of resolving rule$_{\text{AGT}}$ $R1^{t1,refE}_{Res}$

We have thus presented the translation of bindings involving default resolving or no resolving at all. We now present the translation of the last kind of bindings, those involving non-default resolving.

### Resolving Rules$_{\text{AGT}}$ with Non-Default Resolving

Bindings with non-default resolving have the following shape[2]:

*tgtObj* `:` *tgtType* `(` *tgtRef* `<-`
`thisModule.resolveTemp(Sequence{`$navExp_1$`, ..., `$navExp_N$`}, `*tgtPat*`))`

The construction of the resolving rule$_{\text{AGT}}$ $R^{tgtObj,tgtRef}_{Res}$ operates in the same steps as Table 6.1 except for steps 2 and 3 which are presented in Table 6.2. The difference here is that the trace node that is added in step 3.a to perform the resolving now has multiple *from* elements to account for rules$_{\text{ATL}}$ that have multiple source pattern elements. Additionally, its outgoing *to* edge does not have an index, meaning that any output object can be retrieved, not necessarily the first one as was the case with default resolving. It is in step 3.c that we add the edge *tgtPat* that determines which one of the output objects is the result of the resolve operation.

[2]The case where the first parameter of `resolveTemp` is an object is treated in the same way as a `Sequence` containing only that object.

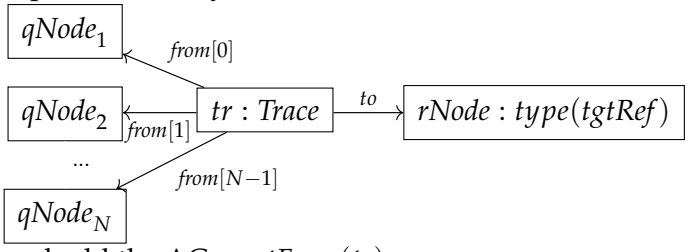
| | **resolveTemp(Sequence{**$navExp_1$**, ..., **$navExp_N$**}, **$tgtPat$**)** |
|---|---|
| **Step 2** | Translate each $navExp_i$ as per Section 6.6. This will complement the $LHS$ with the required navigations and ACs and return as a result a set of nodes $qNode_i$ representing the navigated objects |
| **Step 3.a** | Perform a *non-default resolve* by matching a trace node with the exact *from* tuple. Differently than for the default resolve, *to* is not indexed.   and add the AC $exactFrom(tr)$ |
| **Step 3.b** | Compute *CRules* as the set of all candidate $rules_{ATL}$ that have $N$ source pattern elements and *tgtPat* as one of their target pattern elements. |
| **Step 3.c** | Add to the $rule_{AGT}$'s AC the following disjunction: $\bigvee_{cRule \in CRules} \exists \left( tr : <cRule>\_Trace \xrightarrow{tgtPat} rNode \right)$ |

Table 6.2: Translation of an ATL binding with non-default resolving

### 6.5.3 Final Cleanup Phase

At this stage, we have explained how instantiation and resolving $rules_{AGT}$ create the elements of the output model. The final phase of the transformation is the cleanup phase which deletes the trace nodes used for resolving, and the input model to leave only the output model. This is done by iterating the cleanup rules *DeleteInObjs* which deletes input objects and *DeleteTraceNodes* which deletes trace nodes. These $rules_{AGT}$ are depicted in Figure 6.7. Note that these $rules_{AGT}$ only delete objects and not references between them. This is because Henshin automatically deletes edges incoming to a deleted node. In other AGT frameworks which do not have this automatic behavior, it could be necessary to add cleanup rules that delete each kind of source and trace edge explicitely.


*DeleteInObjs* *DeleteTraceNodes*

Figure 6.7: Cleanup rules

## 6.6 Translating OCL Guards and Binding Expressions

So far we have presented the general translation scheme producing instantiation, resolving, and cleanup $\text{rules}_{\text{AGT}}$. As explained previously, one of the steps in this translation consisted of translating OCL $\text{rule}_{\text{ATL}}$ guards and binding expressions to ACs of respectively instantiation and resolving $\text{rules}_{\text{AGT}}$. Despite the considerable difference between NGC and OCL, NGC has been shown to be expressively equivalent to first order logic [Poskitt, 2013] which is the core of OCL. Translations of subsets of OCL to NGC have been proposed in [Arendt *et al.*, 2014; Radke *et al.*, 2015b] with a highly theoretical approach and in [Bergmann, 2014] with a wider supported OCL subset and an experimental approach. We have based our translation on the one in [Radke *et al.*, 2015b], however since both existing works do not support ordered sets, we have extended the translation with support of ordered sets in the context of ATL transformations. The next section recalls the basic principles of the existing OCL to NGC translation and illustrates it on concrete examples in the context of the ATL to AGT translation. For the general translation schemes we refer the interested reader to [Radke *et al.*, 2015b] (and the long version of the publication [Radke *et al.*, 2015a]). Section 6.6.2 will then detail our own contribution addressing ordered sets.

### 6.6.1 General Principles of the Existing Translation

The subset of OCL supported by our translation is the one defined in [Radke *et al.*, 2015b] which includes basic navigation of references of all multiplicities and single multiplicity attributes, first order logic constructs and `Set` as the only collection type. Basic set operations such as `exists()`, `forAll()`, `select()`, `collect()`, `union()`, `intersection()` are supported. In the following we illustrate the translation of some of these constructs on concrete examples.

#### OCL $\text{Rule}_{\text{ATL}}$ Guards

The main idea is to translate OCL guards into NGC application conditions that are satisfied under the same conditions and OCL queries into graphs that match the objects in the query's result set. Starting with OCL guards, let us consider the guard of rule `R1` from Figure 6.1 recalled on the first row of Table 6.3.

Table 6.3 shows the step by step translation of this guard into an application condition of the instantiation $\text{rule}_{\text{AGT}}$ $R1_{Inst}$. The translation proceeds along the abstract syntax tree of the OCL expression. Accessing the variable `s` yields the

| Guard: `from s:IN!A ( s.refB->exists(b \| b.name = 'Hodor'))` | |
|---|---|
| `s` | $\exists\left(\boxed{s:A}\right)$ |
| `s.refB` | $\exists\left(\boxed{s:A}\xrightarrow{refB}\boxed{r:B}\right)$ |
| `s.refB->exists(b \| ...)` | $\exists\left(\boxed{s:A}\xrightarrow{refB}\boxed{r:B}, \exists\left(\boxed{r:B}\right)\right)$ |
| `s.refB->exists(b \| b.name ...)` | $\exists\left(\boxed{s:A}\xrightarrow{refB}\boxed{r:B}, \exists\left(\boxed{\begin{array}{c} r:B \\ \hline name = p \end{array}}\right)\right)$ |
| `s.refB->exists(b \| b.name = 'Hodor')` | $\exists\left(\boxed{s:A}\xrightarrow{refB}\boxed{r:B}, \exists\left(\boxed{\begin{array}{c} r:B \\ \hline name = p \end{array}}, p = \text{"Hodor"}\right)\right)$ |

Table 6.3: Step-by-step translation of an OCL guard to a NGC application condition

creation of a node $\boxed{s:A}$ mapped to its counterpart in the LHS of the rule. The navigation of a reference `s.refB` yields the creation of a reference $\boxed{s}\xrightarrow{refB}\boxed{r:B}$ where $r$ represents one object in `s.refB`. Then `exists()` yields a new nesting level in the NGC where the iterator $b$ is mapped to node $r$ in the upper nesting level. Finally, the attribute navigation `b.name = 'Hodor'` is translated by creating a rule parameter $p$ assigned to the *name* attribute in node $b$ and a condition $p =$ "Hodor". The resulting NGC is the application condition associated with the instantiation $\text{rule}_{\text{AGT}}$ $R1_{Inst}$ as depicted in Figure 6.5.

### OCL Object Queries

OCL queries in $\text{rule}_{\text{ATL}}$ bindings are handled similarly to OCL guards except that along with the resulting application condition, a node or an expression is returned as a result. For object queries, a node representing objects in the result set of the query is returned by the translation. This result node is then used by the algorithms in Table 6.1 and Table 6.2 to construct resolving $\text{rules}_{\text{AGT}}$. Since bindings in the example transformation of Figure 6.1 are all simple, we illustrate this translation over the following query where `s` is the source pattern object of type `A` from rule `R1`:

```
s.refB->select(b | b.name = 'Sansa')
```

Table 6.4 shows the step-by-step translation of this query into an application condition. Note that the result of the OCL query is a set of elements while the result of the translation is a single node $\boxed{r:B}$. This node represents one element of the result set, and it is the iteration of the resolving $\text{rule}_{\text{AGT}}$ that allows to process all elements of the query, one after the other.

| Query: `s.refB->select(b \| b.name = 'Sansa')` | |
|---|---|
| `s.refB` | $\exists \left( \boxed{s:A} \xrightarrow{refB} \boxed{r:B} \right)$ |
| `s.refB->select(b \| ...)` | $\exists \left( \boxed{s:A} \xrightarrow{refB} \boxed{r:B}, \exists \left( \boxed{r:B} \right) \right)$ |
| `s.refB->select(b \| b.name = 'Sansa')` | $\exists \left( \boxed{s:A} \xrightarrow{refB} \boxed{r:B}, \exists \left( \boxed{\begin{array}{c} r:B \\ \hline name = p \end{array}}, p = "Sansa" \right) \right)$ <br> Result node is $\boxed{r:B}$ |

Table 6.4: Step-by-step translation of an OCL object query

### OCL Attribute Queries

Finally for attribute queries, an expression involving rule parameters and constants is returned and used to initialize the target attribute in the resolving $\text{rule}_{\text{AGT}}$ constructed by the algorithm in Table 6.1. This is shown in Table 6.5 for the query `s.name + '1'`. The resulting expression $p + "1"$ is then used by the translation algorithm of Table 6.1 to initialize a target attribute of an output object.

| Query: `s.name + '1'` | | |
|---|---|---|
| `s.name` | $\exists \left( \boxed{\begin{array}{c} s:A \\ \hline name = p \end{array}} \right)$ | |
| `s.name + '1'` | $\exists \left( \boxed{\begin{array}{c} s:A \\ \hline name = p \end{array}} \right)$ | Result is the expression: $p + "1"$ |

Table 6.5: Step-by-step translation of an OCL attribute query

## 6.6.2 Supporting Ordered Sets

The translation presented in the previous section and on which we have based our work does not support ordered sets. As a result we cannot ensure that the semantics of ATL regarding ordered references is preserved in the AGT transformation. This is concretely observed by a difference in the ordering of elements in models resulting from the ATL and AGT versions of the same transformation. Since ordering can be very important in transformations involved in code generation (*e.g.* the order of code statements), we have extended the existing translation of OCL to NGC with support for ordered sets in the context of our translation of ATL trans-

formations. The ordering problem occurs in several situations that we detail in the following.

#### Problem 1: Navigation of Ordered References

The first problem is dealing with the navigation of ordered references. We illustrate this problem with the following binding from `R1` in Figure 6.1:

```
t1 :  OUT!D ( refE <- s.refB )
```

`refB` is a multivalued reference (*i.e.* upper bound larger than 1). We have previously shown the translation of this binding in Figure 6.6 under the assumption that `refB` is a *non-ordered* reference. As explained previously, the navigation `s.refB` is *flattened*, meaning that the elements of the collection are not handled all at once, but rather one by one thanks to the iteration of $R1^{t1,refE}_{Res}$. According to AGT graph matching, objects in a multivalued reference may be matched in any order[3], so $R1^{t1,refE}_{Res}$ may be applied to objects in `s.refB` in any order. Therefore objects in `t1.refE` may end up in a different order than their counterparts in `s.refB`. If `refB` and `refE` are ordered references, then this constitutes a divergence from the ATL semantics which honours the order of objects in collections. Therefore we need a way to force the matching of objects in `s.refB` in an orderly fashion.

#### Solution to Problem 1: The Ordering Application Condition

We propose to complement the regular translation of navigation expressions [Radke *et al.*, 2015b] with an additional NGC forcing objects to be matched in the correct order. Intuitively, this NGC should express the fact that an object in `s.refB` should be matched only if all preceding objects in `s.refB` have already been handled by the resolving $\text{rule}_{\text{AGT}}$. This corresponds to the following NGC:

$$orderingAC = \exists \Big( \boxed{s:A} \xrightarrow{refB[i]} \boxed{qNode:B}, \\ \forall \Big( \boxed{s:A} \xrightarrow{refB[j]} \boxed{qNode_1:B} \mid j < i,\ wasResolved^{t1,refE}_{R1}(qNode_1) \Big) \Big)$$

Where:

- $i$ : index of the object *qNode* currently being handled.
- $j$ : index of the object $qNode_1$ which iterates over objects preceding *qNode*.
- $wasResolved^{t1,refE}_{R1}(n)$ : A NGC which evaluates to *true* if node $n$ has already been handled by the resolving $\text{rule}_{\text{AGT}}$.

[3] the multivalued reference is in fact represented as several references in the graph, and any of them may be matched with no particular ordering priority

Now we need to define $wasResolved_{R1}^{t1,refE}(n)$. We can determine that a node $n$ has been already handled by checking if the node to which it resolves exists in the target reference $t1.refE$. Therefore the following definition is suitable: $wasResolved_{R1}^{t1,refE}(n) =$

$$\exists \left( \boxed{n} \xleftarrow{from[0]} \boxed{tr:Trace} \xrightarrow{to[0]} \boxed{:E} \xleftarrow{refE} \boxed{t1:D},\ exactFrom(tr) \right)$$

With the above definitions, adding *orderingAC* as an application condition of $R1_{Res}^{t1,refE}$ ensures that objects in `s.refB` are processed in the correct order, thus honoring the ATL semantics.

Let us now generalize this reasoning to the case where the navigation is filtered with a `select` operation:

```
t1 :  OUT!D ( refE <- s.refB->select(e | body(e) )
```

Now an object in `s.refB` should be matched only if it satisfies the `select` condition, and if all preceding objects in `s.refB` which also satisfy the `select` condition have been handled by the resolving $\text{rule}_{\text{AGT}}$. Therefore the AC that would ensure the orderly processing of objects is the following:

$$\begin{aligned} orderingAC = \exists \Big( & \boxed{s:A} \xrightarrow{refB[i]} \boxed{qNode:B},\ tr_{body}(qNode) \bigwedge \\ & \forall \left( \boxed{s:A} \xrightarrow{refB[j]} \boxed{qNode_1:B} \mid j < i, \right. \\ & \left. tr_{body}(qNode_1) \implies wasResolved_{R1}^{t1,refE}(qNode_1) \right) \Big) \end{aligned}$$

Where $tr_{body}(n)$ is the NGC resulting from the translation of the OCL constraint $body(\texttt{e})$, applied to a node $n$.

This solution can be further generalized to more complex expressions such as chained navigation (`s.refB->collect(b | b.refA)->collect(a | a.refB)`). The generalisation is not yet formalised at this stage, however it has been implemented to a considerable extent in the prototype discussed in the next section which allowed an experimental validation of the core ideas of our solution.

**Problem 2: Aggregating Queries**

A second challenge concerning ordering is the handling of bindings that aggregate results of several queries. This is the case of the following binding shapes where in (1) resolved objects in *tgtRef* should be in the same order as the source objects in the `OrderedSet`, and in (2) $oclQuery_1$ should be resolved before $oclQuery_2$.

$$tgtRef \texttt{ <- OrderedSet\{}oclQuery_1\texttt{, }oclQuery_2 \ldots oclQuery_N\texttt{\}} \quad (1)$$

$$tgtRef \texttt{ <- }oclQuery_1\texttt{->union(}oclQuery_2\texttt{)} \quad (2)$$

**Solution to Problem 2: Sequential Rules**

To preserve the ordering of elements, we propose to translate such bindings as separate successive bindings: $tgtRef$ `<-` $oclQuery_1$, $tgtRef$ `<-` $oclQuery_2$, ... Each such binding results in a separate resolving $\text{rule}_{\text{AGT}}$ and the $\text{rules}_{\text{AGT}}$ are sequenced in the same order as the queries in the original binding. Consequently objects are appended to $tgtRef$ in the right order at run-time. Therefore (1) is translated to $N$ sequential resolving $\text{rules}_{\text{AGT}}$ and (2) is translated to 2 sequential resolving $\text{rules}_{\text{AGT}}$.

This concludes the conceptual presentation of our translation of ATL to AGT. We have emulated the ATL default and non-default resolve mechanisms by organising the AGT translation in 2 phases, instantiation and resolving, and by using trace nodes. We have translated OCL $\text{rule}_{\text{ATL}}$ guards and binding queries to $\text{rule}_{\text{AGT}}$ application conditions with support for ordered sets by extending existing works. We now discuss the concrete implementation of this translation.

## 6.7 Implementation

We have implemented our translation of ATL to AGT in *ATLAnalyser* as two Java components, *ATL2AGT* and *OCL2NGC* addressing respectively the aspects of resolving and OCL expressions with support for ordered sets. Both rely on the EMF API based on the abstract syntax metamodel of ATL which embeds the abstract syntax metamodel of OCL, and the metamodel of Henshin graph transformations. The translation operates in the following general steps:

1. Parse the input ATL transformation using the ATL compiler API.
2. For each ATL $\text{rule}_{\text{ATL}}$ create an instantiation $\text{rule}_{\text{AGT}}$. Use *OCL2NGC* to translate OCL guards into an NGC application conditions.
3. For each ATL binding create a resolve $\text{rule}_{\text{AGT}}$. Use *OCL2NGC* to translate OCL queries into NGC application conditions.
4. Create the final cleanup rules.
5. Create the resulting graph transformation consisting of the iteration and sequencing of *Instantiation*, *Resolve* and *Cleanup* rules.

## 6.8 Related Work

Though translations of OCL to NGC have been conducted [Radke *et al.*, 2015b; Bergmann, 2014], no previous work has proposed a translation of ATL to AGT to the best of our knowledge. In [Poskitt *et al.*, 2014] the authors propose to translate model transformations from the Epsilon language family (arguably similar to ATL and OCL) to AGT to show through formal proof that a given pair of unidirectional transformations forms a bidirectional transformation. However this work is still at an early stage and an automatic translation is not yet proposed.

In the broader context of the analysis of model transformations several existing works have translated ATL to other formalisms. In [Büttner *et al.*, 2012b] ATL transformations are translated to a *transformation model*, a representation in first-order predicate logic used for the formal verification of model transformations[4] as discussed in Section 3.2. In [Troya and Vallecillo, 2011] ATL transformations are translated to a Maude specification with a rewriting logic arguably similar to our graph rewriting transformation. It is interesting to note that both these translations rely on trace nodes similar to the trace nodes in our approach. On the contrary, the translation to first-order predicate logic in [Büttner *et al.*, 2012a] does not use trace nodes because it targets SMT-solvers which were found to perform badly when the encoding involves trace nodes. Instead, resolving mechanisms are encoded directly as logic formulae over input and output elements without intermediary trace elements. A similar trace-less approach could be considered for the translation to AGT if it shows benefits in that context. This is yet to be investigated in future work.

In terms of the scope of the existing translations, we find that most approaches handle a subset of ATL similar to ours (*i.e.* the declarative out-place subset) except for the translation to MAUDE which covers a wider subset including imperative constructs (*i.e.* `do` blocks), lazy and called rules, and the refining *in-place* mode. Supporting these features is part of the future evolutions of our translation. Even though it is certainly not straightforward, we do not foresee major obstacles to supporting these advanced features in AGT.

It is also important to note that our translation of ATL to AGT opens the door to applying analyses of AGT to ATL transformations. While some of these analyses such as the (interactive or automatic) formal proof of Hoare-style correctness can already be performed with existing formalisations of ATL, others such as the construction of preconditions are not possible with existing translations of ATL. The

[4] transformation models are also used for the verification of other transformation languages, not just ATL

latter will of course be demonstrated shortly in Chapter 7. Other analyses of the AGT theory can be useful for ATL translations and are made possible by our novel translation. This will be the subject of future work as discussed in Chapter 11.

## 6.9 Conclusion

In this chapter we have detailed the second contribution of our work: the translation of ATL transformation to AGT transformations. The two main challenges in this contribution are the handling of ATL resolve mechanisms which do not have equivalents in the AGT semantics, and the translation of OCL guards and queries into application conditions of AGT rules. For the first challenge we proposed to build the resulting AGT transformation as 2 phases: an instantiation phase which creates output elements and trace nodes, and a resolving phase which creates links between output elements and initializes their attributes. Resolving mechanisms were then implemented in the resolving phase relying on trace nodes created in the instantiation phase. For the second challenge we have relied on an existing translation of OCL to NGC [Radke *et al.*, 2015b] and extended it with support for ordered sets in the context of our translation. The validation of these proposals will be detailed in Chapter 10.

Having presented the translation of ATL transformations to AGT, we now move to the second contribution of this thesis: translating postconditions to preconditions using the AGT framework.

# Chapter 7

# Transforming Postconditions to Preconditions

## Contents

## 7.1 Introduction

In the previous chapter we have translated ATL transformations into equivalent AGT transformations. In this chapter, we define analyses that reason on AGT transformations and transform a postcondition into a precondition that guarantees the postcondition. The transformation of postconditions to preconditions is a know subject in the literature of AGT. In particular, the construction $wlp$ of the *weakest liberal precondition* is proposed in existing work in the context of the formal proof of correctness of graph transformations. The main challenge of using this construction is that when the graph transformation involves iterations, $wlp$ can theoretically be an infinite construction.

In this chapter, we propose an approach to guarantee that $wlp$ is always finite. To do so we introduce a new *bounded iteration* construct to replace unbounded iteration in AGT transformations. We propose a finite $wlp$ construction for this new construct and prove formally the correctness of the proposed construction. The computed precondition is only valid within a scope defined by the bounds chosen for bounded iterations. For this reason we propose an alternate $scopedWlp$ construction which embeds the validity scope into the precondition itself. The resulting precondition becomes applicable to the original unbounded transformation. We also prove this result formally. This set of formal contributions is not specific to ATL and is applicable to arbitrary structural AGT transformations.

Given the complexity of the theoretical concepts involved in the $wlp$ construction, we have chosen to consider solely structural aspects of transformations and graph constraints (*i.e.* scalar object attributes are not supported). Extending our proposals beyond this scope remains for future work.

The chapter is organised as follows. First Section 7.2 recalls the background of precondition construction in AGT and defines the scope of our work. Then Section 7.3 recalls the formal foundations of AGT that will be the basis for our definitions and proofs. In Section 7.4 we present the properties of different kinds of preconditions and explain why we choose to rely on the $wlp$ construction, then Section 7.5 presents the details of this construction. Section 7.6 introduces our theoretical contributions regarding the finiteness of $wlp$ and the $scopedWlp$ construction. Finally Section 7.7 discusses related work, in particular an alternative approach to construct preconditions which was proposed for OCL.

## 7.2 Background and Scope

In general, postcondition to precondition transformation in the AGT theory has been proposed under several forms and variants in existing works of this field. Earlier works in [Ehrig *et al.*, 2006; Habel *et al.*, 2006a; Habel and Pennemann, 2009; Ehrig *et al.*, 2012a] proposed the construction for structural graph conditions (*i.e.* no scalar attributes of nodes or edges). More recent works [Poskitt and Plump, 2013; Poskitt, 2013; Deckwerth and Varró, 2014] have extended the construction to conditions involving node and/or edge attribution and are therefore better suited to our needs. However as stated previously, we will focus on structural aspects, and will therefore use the earlier works on this topic.

In comparison with the AGT framework used in Chapter 6, restricting our work to structural aspects roughly means that transformation rules do not have rule parameters and cannot match and manipulate object scalar attributes (*e.g.* Strings, Integers). Similarly nested graph conditions will not include constraints over scalar rule parameters and will only consist of the purely structural morphism part. Indexing edges with literal values will be supported because it is needed by the ATL resolving mechanisms. However indexing edges with variables (*i.e.* rule parameters) will not be supported which means that ordered sets cannot be supported. The next section will define precisely the AGT concepts that we will use in their purely structural form.

## 7.3 Fundamentals of AGT

In the following we recall the main concepts required to define the $wlp$ construction. The definitions are based on, and often identical to, the ones in [Ehrig *et al.*, 2012a] and [Radke *et al.*, 2015a] however we will sometimes take shortcuts for less relevant aspects. Part of this presentation is somewhat redundant with the informal introduction of AGT previously made in Chapter 6 however it is necessary to redefine the concepts formally as they will be necessary for the definition of our theoretical contributions. We will first define *Graphs* and *Graph Morphisms*, followed by *Nested Graph Conditions* and *Graph Constraints*, and finally *Graph Transformations*.

**Definition 7.1 (Graph (informal definition)).** As previously introduced and extensively used in this thesis, a graph $G$ is composed of *nodes* and directed *edges* connecting the nodes. A graph is typed by a *metamodel* (or *type graph* in the AGT theory). A metamodel is itself a graph where nodes are *metaclasses* and directed edges are either *references* between metaclasses or *inheritance* relationships between meta-

classes. Each node in a graph is typed by a metaclass of the metamodel, and each edge is typed by a reference. A metaclass $X$ is a child of another metaclass $Y$ if it inherits directly or transitively from $Y$.

Edges of a graph which have an *ordered reference* as a type can have an *index* which is an integer value. Edges of the same type, outgoing from the same node, are thus ordered according to their indexes.

Contrarily to the graphs used in Chapter 6, nodes here do not have scalar attributes such as strings and integers.

□

**Example 7.1 (Graph).** In all examples of this chapter we use the metamodel resulting from the combination of the input, output and trace metamodels that were used in Chapter 6 where an example ATL transformation was translated to AGT. We recall this metamodel in a purely structural form in Figure 7.1.

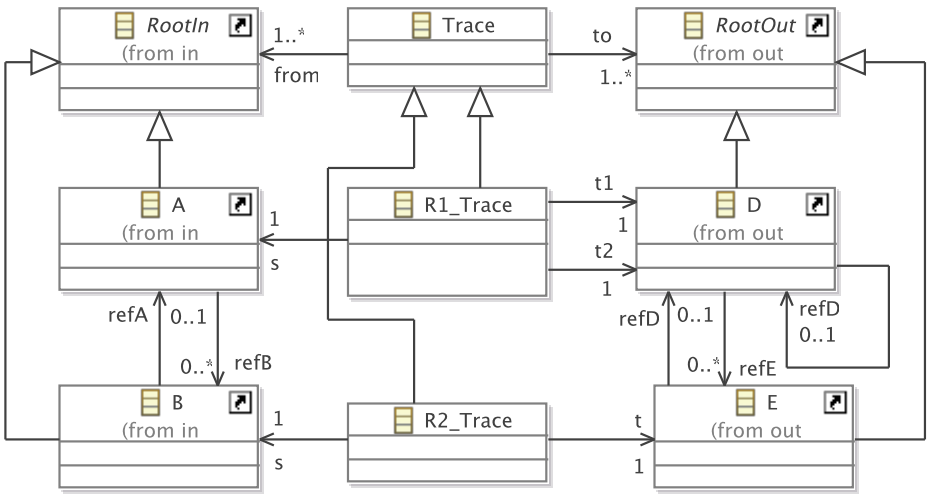
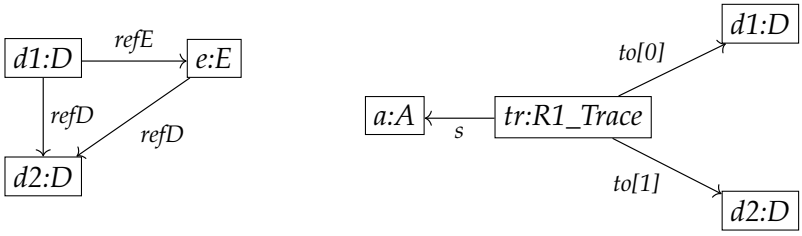



Figure 7.1: Example metamodel

The following two graphs are instances of this metamodel. The integers appearing between brackets in the graph on the right are indexes that define an order relationship between edges of the same type outgoing from the same node.




□

Next we define the notion of a graph morphism which is a mapping from a source graph to a target graph. Graph morphisms will then be the basis to define graph conditions and constraints, and transformation rules.

**Definition 7.2 (Graph Morphism (informal definition)).** A graph morphism $m : O \longrightarrow I$ is a mapping from an *origin* graph $O$ to an *image* graph $I$ that preserves sources and targets of edges. This means that if an edge in $O$ is mapped to an image edge in $I$, then its source and target nodes are respectively mapped to the source and target nodes of the image edge. A morphism also honors typing: an origin node can only be mapped to an image node having the same metaclass, or a child metaclass. And finally, a morphism honors edge indexing meaning that if an edge in $O$ has an index, then its image edge must have the same index in $I$.

A morphism can be *partial*, meaning that not all elements of $O$ have an image in $I$. We denote by $dom(m) \subseteq O$ the domain of $m$ *i.e.* the set of elements (nodes and edges) in $O$ mapped by $m$. We denote by $codom(m) \subseteq I$ the codomain of $m$ *i.e.* the set of elements in $I$ that have origin elements in $O$ via $m$. A morphism is *total* if $dom(m) = O$, meaning that all elements (nodes and edges) of $O$ are mapped by the morphism to images in $I$.

A morphism is *injective* if it preserves distinctness. That is if $x_1$ and $x_2$ are two nodes or two edges in $O$, then $x_1 \neq x_2 \Rightarrow m(x_1) \neq m(x_2)$. In that case the morphism is denoted as $m : O \hookrightarrow I$.

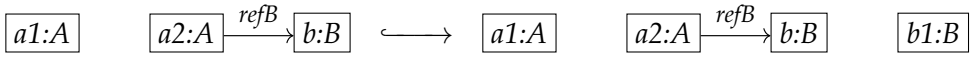
A morphism is *surjective* if all elements in the image graph have a corresponding origin element in the origin graph. That is if $y$ represents a node or an edge in $I$, then $\forall y \in I, \quad \exists x \in O,\ m(x) = y$.

A morphism that is both total and injective is called a *match* of $O$ in $I$ because it means that there is a subgraph in $I$ that has the same structure as $O$. A morphism that is both injective and surjective is called an *isomorphism*.

The mapping of nodes is graphically represented by assigning mapped nodes the same identifiers in the origin and target graphs.

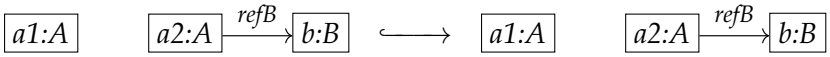
□

**Example 7.2 (Graph Morphisms).** The following morphism is injective and non-surjective.




The following morphism is non-injective and surjective. Note that it honors edge indexing because *refB* does not have an index in the origin graph.


The following morphism is injective and surjective (an isomorphism).




□

Now we use the notion of graph morphism to define Nested Graph Conditions and Graph Constraints which express constraints on graphs. We will later use graph constraints to express post- and pre-conditions.

**Definition 7.3 (Nested Graph Condition).** A Nested Graph Condition (NGC) over a graph $P$ is one of the following:

- *true*.
- a *basic condition* over $P$ : $\exists(a, c)$
  where $a : P \hookrightarrow C$ is an injective morphism and $c$ is a NGC over $C$. $P$ is called the *host* of the condition, and $C$ is its *conclusion*.
- a boolean formula of NGCs over $P$ : the negation $\neg c$, the conjunction $\bigwedge_i c_i$ and the disjunction $\bigvee_i c_i$ where $c$ and $c_i$ are NGCs over $P$.

The following notational shortcuts are used:

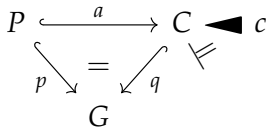
$$
\begin{aligned}
false &\equiv \neg\, true \\
\exists a &\equiv \exists(a, true) \\
\forall(a, c) &\equiv \neg\, \exists(a, \neg\, c) \\
c_1 \Rightarrow c_2 &\equiv \neg\, c_1 \vee c_2 \\
\exists(C, c) &\equiv \exists(P \hookrightarrow C, c) \text{ when } P \text{ is the empty graph } \phi \\
&\quad \text{or is known from the context } \textit{i.e.} \text{ it is the conclusion} \\
&\quad \text{of the containing condition}
\end{aligned}
$$

□

Now we explain the meaning of NGCs and under which circumstances they are satisfied.

**Definition 7.4 (Semantics of NGCs).** Given graphs $P$ and $G$, and a match $p : P \hookrightarrow G$ of $P$ in $G$, $p$ satisfies a NGC $d$ over $P$ (denoted as $p \models c$) under the following semantics:

- if $d = true$ then $p$ always satisfies $c$, that is $\forall p, p \models true$.
- if $d = \exists(a, c)$ where $a : P \hookrightarrow C$ and $c$ is a NGC over $C$, then $p \models d$ if there exists a match $q : C \hookrightarrow G$ such that $q \circ a = p$ and $q \models c$, as depicted in the following diagram:




$$p \models \exists(a, c) \qquad \equiv \qquad \exists\, q : C \hookrightarrow G, \quad q \circ a = p \ \wedge \ q \models c$$

  A rough explanation of this semantics is that $P$ expresses a structure of nodes and edges that have been matched via morphism $p$ in $G$. $C$ contains the same structure as $P$ (mapped via $a$) and adds further elements to be matched. Therefore $p$ satisfies the condition if it can be complemented with matches of the additional elements in $C$ to form $q$, such that $q$ also satisfies the nested condition $c$.

- Negation, conjunction and disjunction have the usual semantics.

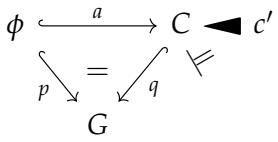
□

We have seen that NGCs define constraints over matches of one graph in another. We can also define a special case of NGCs called *graph constraints* which express constraints over all graphs and not over particular matches. Graph constraints are used to represent postconditions and preconditions.

**Definition 7.5 (Graph Constraint and Post-/Pre-condition).** NGCs over the empty graph $\phi$ are called *graph constraints*. A graph $G$ satisfies a graph constraint $c$ if the morphism $\phi \hookrightarrow G$ satisfies $c$.

$$G \models c \quad \equiv \quad \phi \hookrightarrow G \models c$$

As illustrated by the diagram below this definition means that for basic conditions, $G$ satisfies $\exists(\phi \hookrightarrow C, c')$ if there exists a match of $C$ in $G$ that satisfies $c'$, as illustrated by the diagram below.




As their name indicates, graph constraints are a way to express constraints over graphs and not over particular matches. As such, they are used to express postconditions and preconditions of transformations which are constraints over the transformed graph respectively *after* and *before* the application of a transformation.

□

**Example 7.3 (Graph Constraint).** The following graph constraint requires the existence of 2 objects of types $D$ and $E$ connected with a reference *refE*.

$$\begin{aligned} Post_1 &= \exists \left( \boxed{d{:}D} \xrightarrow{refE} \boxed{e{:}E} \right) \\ &\equiv \exists \left( \phi \hookrightarrow \boxed{d{:}D} \xrightarrow{refE} \boxed{e{:}E} \right) \end{aligned}$$

The following graph constraint requires the existence of 2 objects of types $D$ and $E$ connected with mutual references *refE* and *refD*.

$$Post_2 = \exists \left( \begin{array}{c} \overset{refE}{\curvearrowright} \\ \boxed{d{:}D} \quad \boxed{e{:}E} \\ \underset{refD}{\curvearrowleft} \end{array} \right)$$

The following slightly more complex constraint requires that every object of type $D$ should have a *refE* reference to an object of type $E$.

$$Post_3 = \forall \left( \boxed{d{:}D} \ , \quad \exists \left( \boxed{d{:}D} \xrightarrow{refE} \boxed{e{:}E} \right) \right)$$

□

Using the definitions of morphisms and NGCs, we can now define transformation rules which manipulate graphs.

**Definition 7.6 (Rule).** A *rule* $\rho = \langle p, ac_L \rangle$ is composed of a *plain rule* $p = \left\langle LHS \overset{r}{\hookrightarrow} RHS \right\rangle$ where $r$ is an injective morphism, and an application condition $ac_L$ which is a NGC over $LHS$.

$$ac_L \blacktriangleright LHS \overset{r}{\hookrightarrow} RHS$$

□

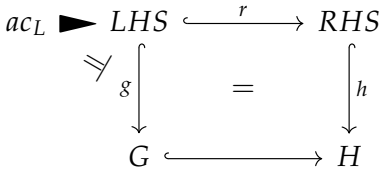
**Definition 7.7 (Applicability and Application of a Rule).** The first step of applying a rule $\rho$ on a graph $G$ is finding a match of its $LHS$ in $G$. For every match $g : LHS \hookrightarrow G$ the rule is *applicable* if the match satisfies the application condition $ac_L$ and satisfies a so-called *dangling condition*. The dangling condition specifies that the application of a rule cannot leave dangling edges in the event of node deletion. In our context, rules translated from ATL never delete nodes/edges. Therefore we simplify the applicability of a rule to the simple satisfaction of its application condition $ac_L$.

When a rule $\rho$ is applicable with a match $g$, its application results in a graph $H$. This is denoted as $G \Rightarrow_{\rho,g} H$ or $G \Rightarrow_{\rho} H$.

$$g \models ac_L \qquad \Leftrightarrow \qquad \exists H,\ G \Rightarrow_{\rho,g} H$$

Applying the rule consists of constructing the so-called *pushout* of $r$ and $g$ as illustrated by the following diagram. $H$ is the resulting *pushout object*. A pushout is

a formal construction defined in category theory that provides a formal definition to the manipulation of graphs. We will not detail the formal definition but will explain it practically.




Roughly, $H$ is constructed by performing a kind of a *union* of graphs $G$ and $RHS$, where the elements in $g(LHS)$ and in $r(LHS)$ are unified in $H$. This means that:

– elements of $G$ that are not matched by $g$ are kept untouched in $H$. That is, $G \setminus g(LHS)$ is preserved in $H$[1].

– matched elements in $g(LHS)$ that are also mapped by $r$ to nodes in the $RHS$ are preserved. That is, $g(LHS \cap dom(r))$ is preserved in $H$.

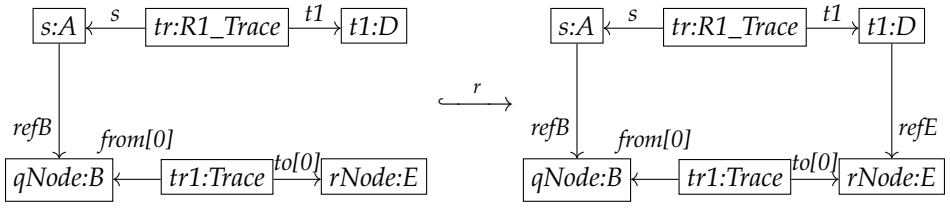
– matched elements in $g(LHS)$ that are *not* mapped by $r$ to nodes in the $RHS$ are deleted[2]. That is, $g(LHS \setminus dom(r))$ is deleted and does not exist in $H$.

– elements in $RHS$ that are not mapped by $r$ are new elements that are created in $H$. That is, $h(RHS \setminus codom(r))$ are new elements created in $H$.

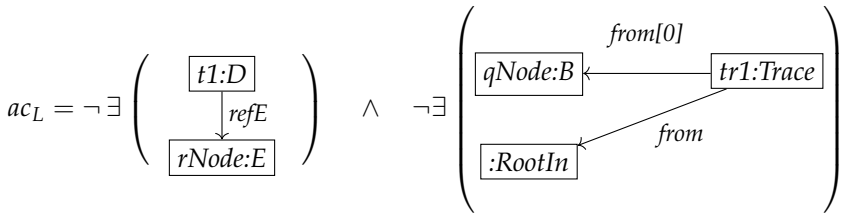
$g$ and $h$ are respectively referred to as the *match* and *comatch* of the rule application.

□

**Example 7.4 (Rule).**
We recall rule $R1^{t1,refE}_{Res}$ from Figure 6.6 where $R1^{t1,refE}_{Res} = \left\langle \left\langle LHS \overset{r}{\hookrightarrow} RHS \right\rangle, ac_L \right\rangle$ where $r$ and $ac_L$ are as follows.




[1]the symbol "\" expresses set difference applied to sets of graph elements
[2]though this is never the case in rules translated from ATL




This rule is applicable when the pattern of its $LHS$ is matched and no *refE* reference already exists between *t1* and *rNode*, and *tr1* does not contain other objects than *qNode* in its *from* reference. The application of the rule preserves all matched elements (all the $LHS$ is mapped to the $RHS$) and creates *refE* between *t1* and *rNode*.

□

We have defined the core of graph manipulation with rules and rule applications. But to build graph transformations we need to sequence the application of rules and apply rules multiple times. This is done with the notion of a high-level program.

**Definition 7.8 (Graph Transformation (High-level Program) [Habel *et al.*, 2006a]).** A *high-level program* specifies in which order AGT rules are applied. A program can be one of the following:

- $Skip$ which is a program that does nothing.
- a rule $\rho$.
- the sequencing of two programs $P$ and $Q$ denoted by $(P; Q)$.
- the iteration of a program $P$ as long as possible, denoted by $P\downarrow$, which is equivalent to a sequencing $(P; (P; (P \cdots)$ until the program $P$ can no longer be applied.

□

The previous definition specifies the semantics of high-level programs in an informal way. However in subsequent contributions, we will need to perform formal proofs which require the following formal definition of the semantics of high-level programs.

**Definition 7.9 (Semantics of High-level Programs [Habel *et al.*, 2006a]).** The *semantics* of a program $P$ is a binary relation $[\![P]\!] \subseteq \mathcal{G} \times \mathcal{G}$ where $\mathcal{G}$ is the (infinite)

set of all graphs. $[\![P]\!]$ contains all pairs of graphs $\langle G, H\rangle$ such that executing $P$ on $G$ yields $H$. $[\![P]\!]$ is defined as follows:

- $[\![Skip]\!] = \{\langle G, G\rangle \mid G \in \mathcal{G}\}$
- for a rule $\rho$,
  $[\![\rho]\!] = \{\langle G, H\rangle \mid G \Rightarrow_\rho H\}$
- for a sequence $(P; Q)$,
  $[\![(P; Q)]\!] = [\![Q]\!] \circ [\![P]\!]$
- for an as-long-as-possible iteration $P\downarrow$,
  $[\![P\downarrow]\!] = \{\langle G, H\rangle \in [\![P]\!]^* \mid \neg\exists M.\ \langle H, M\rangle \in [\![P]\!]\}$
  where $[\![P]\!]^*$ is the *reflexive transitive closure* of $[\![P]\!]$.
  Roughly, the semantics includes all pairs $\langle G, H\rangle$ where $H$ results from the repetitive application of $P$, such that $P$ can no longer be applied over $H$.

When a program $P$ is able to completely execute over a graph $G$ resulting in a graph $H$, we denote this with $G \Rightarrow_P H \Leftrightarrow \langle G, H\rangle \in [\![P]\!]$.

□

Having laid down the foundations of graph transformation, we can now start defining the transformation of postconditions to preconditions.

## 7.4 Properties of Preconditions

We propose to rely on existing work on precondition construction for graph transformation programs and graph constraints in [Habel *et al.*, 2006a; Habel *et al.*, 2006b; Radke *et al.*, 2015a; Ehrig *et al.*, 2012a]. Based on the definitions in [Habel *et al.*, 2006a], given a program $P$ and a condition $d$, a condition $c$ is said to be a *precondition* for $P$ relative to $d$ if for all graphs $G \models c$, the following 3 properties hold:

(1) All resulting graphs $H$ satisfy $d$
$\forall H.\quad \langle G, H\rangle \in [\![P]\!] \ \Rightarrow\ H \models d$

(2) There exists at least one resulting graph, or in other terms, $P$ can be executed on $G$
$\exists H.\quad \langle G, H\rangle \in [\![P]\!]$

(3) The execution of $P$ over $G$ terminates.

A condition $c$ is a *liberal precondition* if for all graphs $G \models c$, at least (1) is satisfied. $c$ is a *termination precondition* if for all graphs $G \models c$, at least (1) and (3) are satisfied. A precondition $c$ is the *weakest precondition* if it is implied by all other preconditions.

The *weakest liberal precondition* and the *weakest termination precondition* are defined similarly.

In our context, we will focus on the *weakest liberal precondition* and assume that the programs that we analyse are always executable and terminating. This assumption is valid for the ATL transformations that we analyse because:

1. The high-level program representing an ATL transformation always relies on as-long-as-possible iteration of rules. This means that even if there are no matches, the execution where no rules are actually executed is still a valid execution which results in an empty output model. As a result, high-level programs representing ATL transformations are always executable.
2. In a high-level program representing an ATL transformation, AGT rules are all equipped with negative application conditions forcing them to apply only once per match of elements in the input model. Since the input model is finite, then there is always a point where rules are no longer applicable and as-long-as-possible iteration terminates. As a result, high-level programs representing ATL transformations are always terminating.

For the above reasons, our work will be based on *liberal preconditions* and the *weakest liberal precondition*. However, despite these assumptions, since the aspects of existence of results and termination are defined in [Habel *et al.*, 2006a] in terms of the weakest liberal precondition, it is easy to extend our work to encompass these aspects as well.

Before going further, we give the formal definitions of liberal preconditions and of the weakest liberal precondition according to [Habel *et al.*, 2006a].

**Definition 7.10 (Liberal Precondition [Habel *et al.*, 2006a]).** Given a program $P$ and a graph constraint $d$, a graph constraint $c$ is a *liberal precondition* of $P$ relative to $d$ if for all graphs $G$:

$$G \models c \quad \Rightarrow \quad \forall H.\ \langle G, H \rangle \in [\![P]\!] \ \Rightarrow\ H \models d$$

Given $P$ and $d$, more than one liberal precondition of $P$ relative to $d$ can exist.

□

**Definition 7.11 (Weakest Liberal Precondition [Habel *et al.*, 2006a]).** Given a program $P$ and a graph constraint $d$, there is a graph constraint $wlp\,(P, d)$ called the *weakest liberal precondition* of $P$ relative to $d$ such that for all other liberal preconditions $c$, we have $c \Rightarrow wlp\,(P, d)$.

This definition is equivalent to saying that for all graphs $G$:

$$G \models wlp\,(P, d) \quad \Leftrightarrow \quad \forall H.\ \langle G, H\rangle \in [\![P]\!] \ \Rightarrow\ H \models d$$

Note that the difference between this definition and the former definition of liberal preconditions is that in this definition the relation is an equivalence $\Leftrightarrow$ while in the former one it is an implication $\Rightarrow$.

□

In the following, Section 7.5 will present the $wlp$ construction and it will become evident that this construction can theoretically be infinite for high-level programs involving as-long-as-possible iteration. For this reason in Section 7.6 we will propose an alternate *bounded* high-level program which partially represents the unbounded program and yields a finite construction.

## 7.5 Weakest Liberal Precondition Construction

The weakest liberal precondition construction $wlp$ relies on a number of basic transformations of NGCs. We will first present the basic transformations defined in [Radke *et al.*, 2015a] and [Ehrig *et al.*, 2012a] and then present the $wlp$ construction as defined in [Habel *et al.*, 2006a]. In our presentation we only detail the constructions and refer the readers to the original publications for the formal proofs that the constructions exhibit the necessary properties.

The core of the $wlp$ construction is handling rules. Given a rule $\rho$ and a postcondition $d$ the construction is roughly done in 3 steps:

1. Transform the postcondition into an equivalent right application condition over the $RHS$ of the rule. This will be transformation $A$ and will be defined in terms of another transformation called *Shift*. The right application condition is:
   $A(\rho, d)$
2. Transform the right applicable condition into an equivalent left application condition over the $LHS$. This will be transformation $L$ and the left application condition is:
   $L(\rho, A(\rho, d))$
3. Construct a precondition that ensures that all valid matches of the rule also satisfy the above left application condition, and by virtue of the previous

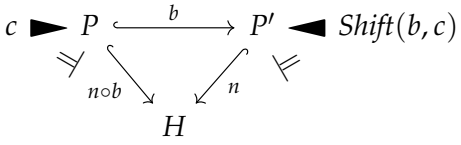
transformations ensure the satisfaction of the postcondition. This is transformation $C_\forall$, and the precondition is:
$C_\forall(\rho, ac_L \Rightarrow L(\rho, A(\rho, d)))$

Next we define the basic transformations *Shift*, $A$, $L$ and $C_\forall$.

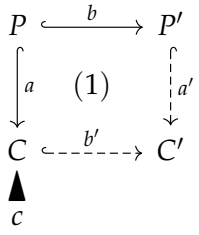
### 7.5.1 Basic NGC Transformations

**Definition 7.12 (*Shift* of NGCs over injective morphisms).** Given a NGC $c$ over $P$ and an injective morphism $b : P \hookrightarrow P'$, there is a *Shift*[3] construction which transforms $c$ via $b$ into a NGC $Shift(b, c)$ over $P'$ such that for each match $n : P' \hookrightarrow H$ of $P'$ in a graph $H$:

$$n \circ b \models c \;\Leftrightarrow\; n \models Shift(b, c)$$




**Construction.** The *Shift* construction is defined as follows:




- $Shift(b, true) = true$
- $Shift(b, \exists(a, c)) = \bigvee_{(a',b') \in \mathcal{F}} \exists(a', Shift(b', c))$
  where $\mathcal{F} =$
  $\{(a', b') \mid (a', b')$ jointly surjective, $a'$ and $b'$ injective, (1) commutes$\}$
  if $\mathcal{F} \neq \emptyset$, and $false$ otherwise.
- $Shift(b, \neg c) = \neg\, Shift(b, c)$
  $Shift(b, \bigwedge_i c_i) = \bigwedge_i Shift(b, c_i)$
  $Shift(b, \bigvee_i c_i) = \bigvee_i Shift(b, c_i)$

*Shift* performs a so-called *overlapping*, or *gluing* of graphs $P'$ and $C$, and $C'$ is referred to as the *overlap graph*. $(a', b')$ jointly surjective means that each element $e \in C'$ is mapped either to an element $e_C \in C$ via $b'$, or to an element $e_{P'} \in P'$ via $a'$, or both to $e_C$ and to $e_{P'}$. In the latter case, we say that $e_C$ and $e_{P'}$ have been *overlapped* or *glued* or *identified*. The set $\mathcal{F}$ is composed of all possible overlaps $C'$ differing by the overlapped elements.

We refer to the pair $(a, b)$ as the *anchor* of the overlapping and to $P$ as the *anchor graph*. This is because if an element $e \in P$ is mapped simultaneously by $a$ and $b$,

[3] This corresponds to the *Shift'* construction over injective morphisms in [Radke *et al.*, 2015a]

then $a(e)$ and $b(e)$ are necessarily overlapped in $C'$. This is because (1) commutes, *i.e.* $a' \circ b = b' \circ a \;\Rightarrow\; a'(b(e)) = b'(a(e))$. Therefore $\mathcal{F}$ contains all overlaps where the elements anchored by $(a, b)$ are always glued and the rest of the elements may be glued or not. When $P = \phi$, no nodes are forced to be overlapped and $\mathcal{F}$ contains all possible overlaps of $P'$ and $C$.

To summarise, the *Shift* construction allows to enumerate all overlaps of two graphs where certain elements of the two graphs all forced to always overlap. An example of this will be shown shortly, after we present the use of the construction in the context of $wlp$.

□

The first step of $wlp$ is to transform the postcondition *graph constraint* into a *right application condition* over the RHS of the rule. This is denoted as the construction $A$ which is based on *Shift*.

**Definition 7.13 ($A$: Transforming a postcondition to a right application condition).** Given a rule $\rho = \left\langle \left\langle LHS \overset{r}{\hookrightarrow} RHS \right\rangle, ac_L \right\rangle$, a graph constraint $d = \exists(\phi \hookrightarrow C, c)$ is transformed to an equivalent right application condition over the $RHS$ via the construction $A(\rho, d)$. $A$ is such that any rule application $G \Rightarrow_{\rho,g} H$ where the comatch $h$ satisfies $A(\rho, d)$ yields a result graph $H$ that satisfies the constraint $d$.

$$h \models A(\rho, d) \quad \Leftrightarrow \quad H \models d$$

$$\begin{array}{ccccc} ac_L & \blacktriangleright & LHS & \overset{r}{\hookrightarrow} & RHS \;\blacktriangleleft\; ac_R = A(\rho, d) \\ & \models & \downarrow g & = & \downarrow h \quad \not\models \\ & & G & \hookrightarrow & H \end{array} \quad \Leftrightarrow \quad H \models d$$

**Construction.** $d$ is a graph constraint of the form $d = \exists(\phi \hookrightarrow C, c)$. $A(\rho, d)$ is constructed by shifting the constraint $d$ over the morphism $\phi \hookrightarrow RHS$. The above properties of $A$ are provided by virtue of the properties of *Shift*.

$$ac_R = A(\rho, d) = \mathit{Shift}\,(\phi \hookrightarrow RHS,\ \phi \hookrightarrow C)$$

Essentially, $A$ consists in enumerating all the possible ways the rule may contribute to the satisfaction or non-satisfaction of the postcondition. This is technically done by enumerating all overlaps of the $RHS$ with the postcondition thanks to the *Shift* construction. Each overlap represents one way in which the elements involved in the rule may be involved in the satisfaction of the postcondition.

□

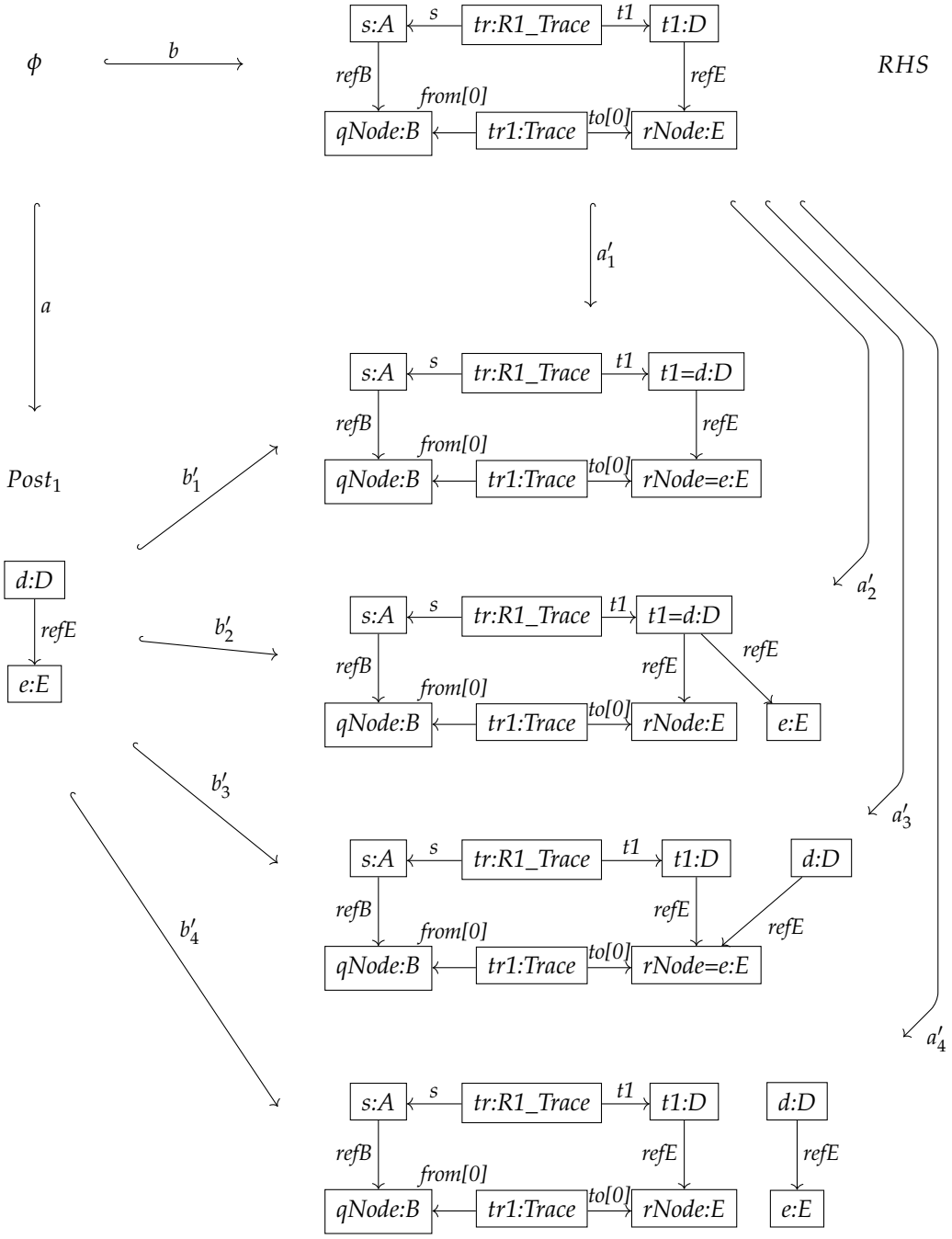



Figure 7.2: Enumerating overlaps of $Post_1$ and the $RHS$ of $R1^{t1,refE}_{Res}$

**Example 7.5 ($A$: Transforming a postcondition to a right application condition).** Considering, $R1^{t1,refE}_{Res}$ from Example 7.4 and $Post_1$ from Example 7.3, the graphical construction of $A\left(R1^{t1,refE}_{Res}, Post_1\right)$ is depicted in Figure 7.2 with all pairs of overlap morphisms.

The construction yields 4 overlaps characterized by the overlap pairs $(a'_1, b'_1)$, $(a'_2, b'_2)$, $(a'_3, b'_3)$ and $(a'_4, b'_4)$. Therefore, the right application condition is:

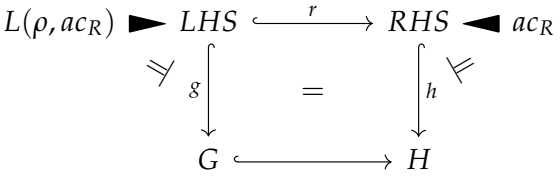
$$ac_R = A\left(R1^{t1,refE}_{Res}, Post_1\right) = \exists\, a'_1 \vee \exists\, a'_2 \vee \exists\, a'_3 \vee \exists\, a'_4$$

□

We have thus far transformed the postcondition into an equivalent right application condition of the rule $\rho$. The next step is to translate the right application condition into a left application condition over the $LHS$. This is the construction $L$.

**Definition 7.14 ($L$: Transforming a right to a left application condition).** Given a rule $\rho$ and a right application condition $ac_R$ over $RHS$, $L(\rho, ac_R)$ is an equivalent condition over $LHS$ such that for any rule application $G \Rightarrow_{\rho,g} H$

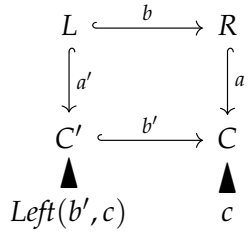
$$g \models L(\rho, ac_R) \qquad \Leftrightarrow \qquad h \models ac_R$$




**Construction.** $L$ is defined using another construction *Left* as follows:

$$L(\rho, ac_R) = Left(LHS \hookrightarrow RHS, ac_R)$$

Given $b : L \hookrightarrow R$ and $c'$ a NGC over $R$, $Left(b, c')$ is an equivalent condition over $L$ defined as follows:




- $Left(b, true) = true$
- $Left(b, \exists(a, c)) = \exists(a', Left(b', c))$ if the pushout complement $(a', b')$ of $(b, a)$ exists and $false$ otherwise.
- $Left(b, \neg\, c) = \neg\, Left(b, c)$
  $Left(b, \bigwedge_i c_i) = \bigwedge_i Left(b, c_i)$
  $Left(b, \bigvee_i c_i) = \bigvee_i Left(b, c_i)$

The pushout complement is a theoretical categorical construction that in our context corresponds to reversing the effect of a hypothetical rule $\langle L \hookrightarrow R \rangle$ on the graph $C$ yielding $C'$. Therefore $L(\rho, ac_R)$ roughly corresponds to reversing the effect

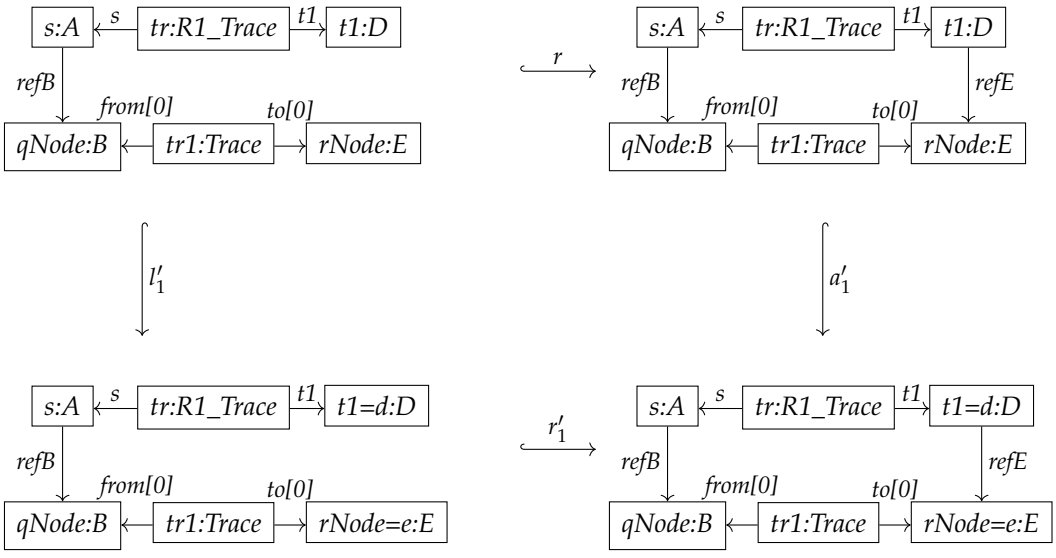
of the rule $\rho$ from $ac_R$. When the reversal cannot be performed for part of the NGC (*i.e.* the pushout complement does not exist), the result of $L$ for that part is $false$. This indicates that the rule could never contribute to the satisfaction of that part of the right application condition.

□

**Example 7.6 ($L$: Transforming a right to a left application condition).** In Example 7.5 we had constructed a right application condition $ac_R = \exists\, a'_1 \vee \exists\, a'_2 \vee \exists\, a'_3 \vee \exists\, a'_4$. Now we apply $L$:

$$L(\rho, ac_R) = Left(r, a'_1) \vee Left(r, a'_2) \vee Left(r, a'_3) \vee Left(r, a'_4)$$

We will only show the construction of $Left(r, a'_1)$ which is the following:




As a result, $Left(r, a'_1) = \exists\, l'_1$ which is obtained by reversing the effect of the rule, *i.e.* removing the edge *refE* between nodes *t1* and *rNode*. The same is done for the other conditions:

$$Left(r, a'_2) = \exists\, l'_2$$
$$Left(r, a'_3) = \exists\, l'_3$$
$$Left(r, a'_4) = \exists\, l'_4$$

The left application condition is therefore:

$$L(\rho, ac_R) = \exists\, l'_1 \vee \exists\, l'_2 \vee \exists\, l'_3 \vee \exists\, l'_4$$

□

The final step is transforming the left application condition into a precondition graph constraint with the construction $C_\forall$.

**Definition 7.15 ($C_\forall$: Transforming a left application condition to a graph constraint).** A left application condition $ac_L$ of a rule $\rho$ can be transformed to a graph constraint $C_\forall(\rho, ac_L)$ expressing that all matches of the $LHS$ satisfy $ac_L$. For any graph $G$:

$$G \models C_\forall(\rho, ac_L) \quad \Leftrightarrow \quad \forall\, g : LHS \hookrightarrow G,\ g \models ac_L$$

**Construction.** The construction consists in simply nesting $ac_L$ into a $\forall$ condition based on the $LHS$ as follows:

$$C_\forall(\rho, ac_L) \quad = \quad \forall(LHS,\ ac_L)$$

Note that the original definition of $C_\forall$ in [Habel *et al.*, 2006a] is different because it takes into account the dangling condition (see Definition 7.6) which is not needed when dealing with ATL transformations. Nonetheless, the remainder of our work is still valid for arbitrary transformations as long as $C_\forall$ is replaced with its original definition if the dangling condition is necessary.

□

### 7.5.2 The $wlp$ Construction

Having defined all the necessary basic transformations, we can now define the $wlp$ construction as per [Habel *et al.*, 2006a].

**Definition 7.16 ($wlp$: Construction of the Weakest Liberal Precondition).** For any rule $\rho$, programs $P$ and $Q$, and graph constraint $d$, $wlp$ is defined inductively as follows:

$$
\begin{array}{lcl}
wlp(Skip, d) & = & d \\
wlp(\rho, d) & = & C_\forall(\rho,\ ac_L \Rightarrow L(\rho, A(\rho, d))) \\
wlp(\ (P;Q)\ , d) & = & wlp(P,\ wlp(Q, d)) \\
wlp(P\downarrow, d) & = & \bigwedge_{i=0}^{\infty} wlp(P^i,\ wlp(P, false) \Rightarrow d)
\end{array}
$$

Where $P^0 = Skip$ and $P^{i+1} = (P^i; P)$ for $i \geq 0$.

The proof that this construction provides the necessary properties of $wlp$ (Definition 7.11) is done by induction over the structure of programs in Appendix D of [Habel *et al.*, 2006b].

□

**Example 7.7 ($wlp$ construction).** In Example 7.6 we had transformed postcondition $Post_1$ into an equivalent left application condition $L(\rho, A(\rho, Post_1))$. We can now construct the weakest liberal precondition:

$$wlp(\rho, Post_1) = C_\forall(\rho,\ ac_L \Rightarrow L(\rho, A(\rho, Post_1)))$$

$$\forall \left( LHS,\ \overbrace{\neg \exists \left( \begin{array}{c} t1{:}D \\ \downarrow refE \\ rNode{:}E \end{array} \right) \ \wedge\ \neg \exists \left( \begin{array}{c} qNode{:}B \xleftarrow{from[0]} tr1{:}Trace \\ :RootIn \xleftarrow{from} tr1{:}Trace \end{array} \right)}^{ac_L} \Rightarrow \underbrace{(\exists\, l_1' \vee \exists\, l_2' \vee \exists\, l_3' \vee \exists\, l_4')}_{L(\rho, A(\rho, Post_1))} \right)$$

□

Having defined the $wlp$ construction for arbitrary AGT high-level programs, we now notice that it can be infinite for programs involving as-long-as-possible iteration. This is the focus of our first contribution detailed in the next section: an approach to ensure the finiteness of the $wlp$ construction for a bounded version of the high-level program.

## 7.6 Bounded Programs and Finite Liberal Preconditions

When we have as-long-as-possible iteration $P\downarrow$ in a transformation $T$, then $wlp(T, d)$ may theoretically be infinite as per Definition 7.16 of the $wlp$ construction. If we are to automate this construction we need a way to make the construction finite. This is a well known problem in the context of weakest preconditions for general purpose programming languages. It is a complex problem that is not yet completely solved in the literature. Some solutions exist typically involving so-called *loop invariants*, manually provided by the programmer or automatically computed for particular kinds of programs. Other solutions perform an *unrolling* of loops into equivalent loop-less programs.

With the exception of manual loop invariants that have been used in [Habel *et al.*, 2006a], solutions to the problem of loops have not yet been explored in the context of AGT programs because the study of weakest preconditions in AGT is relatively recent. Because the study of loop invariants is a complex matter, and given the timing constraints of this thesis, we have pursued a solution similar to loop unrolling for general purpose programs. However due to fundamental differences between AGT programs and general purpose programs, it was not possible to reuse existing work.

The solution we propose is to replace as-long-as-possible iteration with a new form of bounded iteration that has a finite $wlp$ construction. In the following sections we first define the semantics of the proposed bounded iteration construct, and propose its corresponding finite $wlp$ construction that we prove to be indeed the weakest liberal precondition. Then we discuss the scope of validity of the constructed weakest liberal precondition: it is valid only for the bounded transformation but not always for the original unbounded transformation. Finally we propose a variant of $wlp$ called $scopedWlp$ that embeds the scope of validity into the precondition. We prove that the result is a (non-weakest) liberal precondition that is valid for the original unbounded transformation. Even though we illustrate our proposals on ATL transformations, the proposed constructions and the corresponding formal proofs are valid for arbitrary AGT programs.

### 7.6.1 Bounded Iteration with Finite $wlp$

We propose to replace as-long-as-possible iteration $P\downarrow$ with a bounded iteration $P\downarrow_N$. Instead of executing *P as long as possible*, $P\downarrow_N$ executes *P as long as possible and up to N times*.

**Definition 7.17 (Bounded Iteration of Graph Programs).** For any high-level program $P$ and integer $N > 0$, we define $P\downarrow_N$ as the iteration of program $P$ as long as possible, up to $N$ times. After $N$ iterations of $P$ the execution stops even if $P$ is still applicable.

The semantics of $P\downarrow_N$ is:

$$[\![P\downarrow_N]\!] = \left\{ \langle G, H\rangle \in \bigcup_{i=0}^{N-1} [\![P^i]\!] \mid \neg\exists M.\, \langle H, M\rangle \in [\![P]\!] \right\} \cup [\![P^N]\!]$$

Where $P^0 = Skip$ and $P^{i+1} = (P^i; P)$ for $i \geq 0$.

The semantics is similar to that of $P\downarrow$ up to $N-1$ because it includes pairs $\langle G, H\rangle$ where $H$ results from the repetitive execution of $P$, such that $P$ can no longer be applied again on $H$. However, after the $N$th iteration the program stops even though $P$ may still be applicable, hence including all of $[\![P^N]\!]$.

□

Having defined this new form of programs, we now propose a $wlp$ construction for it and prove that this construction yields a weakest liberal precondition.

**Theorem 7.1 ($wlp$ for Bounded Iteration).** For any program $P$, condition $d$, and integer $N > 0$, the following construction $wlp(P\downarrow_N, d)$ is the weakest liberal precondition of program $P\downarrow_N$ relative to $d$.

$$wlp(P\downarrow_N, d) = \bigwedge_{i=0}^{N-1} wlp\left(P^i,\ wlp(P, false) \Rightarrow d\right) \ \wedge\ wlp\left(P^N, d\right)$$

The construction that we propose is finite, and the proof that it yields weakest liberal preconditions is the following.

**Proof.** For any graph $G$, program $P$, integer $N > 0$ and condition $d$:

$$\begin{aligned}
& G \models wlp(P\downarrow_N, d) \\
\Leftrightarrow\ & \forall H.\, (\langle G, H\rangle \in [\![P\downarrow_N]\!] \Rightarrow H \models d) && \left(wlp \text{ Def. 7.11}\right) \\
\Leftrightarrow\ & \forall H.\, \Bigg(\Bigg(\Bigg(\langle G, H\rangle \in \bigcup_{i=0}^{N-1} [\![P^i]\!] \wedge \neg\exists M.\, \langle H, M\rangle \in [\![P]\!]\Bigg) && (\text{Def. } [\![P\downarrow_N]\!]) \\
& \qquad \vee \left(\langle G, H\rangle \in [\![P^N]\!]\right)\Bigg) \Rightarrow H \models d\Bigg)
\end{aligned}$$

$$\Leftrightarrow \quad \forall H.\left(\left(\langle G,H\rangle \in \bigcup_{i=0}^{N-1} [\![P^i]\!] \wedge \neg\exists M.\, \langle H,M\rangle \in [\![P]\!]\right) \Rightarrow H \models d \right. \qquad \begin{pmatrix} a \vee b \Rightarrow c \\ \equiv a \Rightarrow c \wedge b \Rightarrow c \end{pmatrix}$$
$$\left. \wedge \quad \left(\langle G,H\rangle \in [\![P^N]\!]\right) \Rightarrow H \models d \right)$$

$$\Leftrightarrow \quad \forall H.\left(\left(\langle G,H\rangle \in \bigcup_{i=0}^{N-1} [\![P^i]\!] \wedge \neg\exists M.\, \langle H,M\rangle \in [\![P]\!]\right) \Rightarrow H \models d\right) \qquad \begin{pmatrix} \forall x.A \wedge B \\ \equiv \forall x.A \wedge \forall x.B \end{pmatrix}$$
$$\wedge \quad \forall H.\left(\langle G,H\rangle \in [\![P^N]\!] \Rightarrow H \models d\right)$$

$$\Leftrightarrow \quad \forall H.\left(\left(\langle G,H\rangle \in \bigcup_{i=0}^{N-1} [\![P^i]\!] \wedge \neg\exists M.\, \langle H,M\rangle \in [\![P]\!]\right) \Rightarrow H \models d\right) \qquad \left(\text{Def. } wlp\right)$$
$$\wedge \quad G \models wlp\left(P^N, d\right)$$

$$\Leftrightarrow \quad \forall H.\left(\left(\langle G,H\rangle \in \bigcup_{i=0}^{N-1} [\![P^i]\!] \wedge \forall M.\neg\left(\langle H,M\rangle \in [\![P]\!]\right)\right) \Rightarrow H \models d\right) \qquad \begin{pmatrix} \neg\exists x.A \\ \equiv \forall x.\neg A \end{pmatrix}$$
$$\wedge \quad G \models wlp\left(P^N, d\right)$$

$$\Leftrightarrow \quad \forall H.\left(\left(\langle G,H\rangle \in \bigcup_{i=0}^{N-1} [\![P^i]\!] \wedge \forall M.\left(\langle H,M\rangle \in [\![P]\!] \Rightarrow false\right)\right)\right. \qquad \begin{pmatrix} \neg A \\ \equiv \neg A \vee false \\ \equiv A \Rightarrow false \end{pmatrix}$$
$$\left. \Rightarrow H \models d\right)$$
$$\wedge \quad G \models wlp\left(P^N, d\right)$$

$$\Leftrightarrow \quad \forall H.\left(\left(\langle G,H\rangle \in \bigcup_{i=0}^{N-1} [\![P^i]\!] \wedge H \models wlp(P, false)\right) \Rightarrow H \models d\right) \qquad \left(\text{Def. } wlp\right)$$
$$\wedge \quad G \models wlp\left(P^N, d\right)$$

$$\Leftrightarrow \quad \forall H.\left(\langle G,H\rangle \in \bigcup_{i=0}^{N-1} [\![P^i]\!] \Rightarrow (H \models wlp(P, false) \Rightarrow H \models d)\right) \qquad \begin{pmatrix} A \wedge B \Rightarrow C \\ \equiv A \Rightarrow (B \Rightarrow C) \end{pmatrix}$$
$$\wedge \quad G \models wlp\left(P^N, d\right)$$

$$\Leftrightarrow \quad \forall H.\left(\langle G,H\rangle \in \bigcup_{i=0}^{N-1} [\![P^i]\!] \Rightarrow H \models (wlp(P, false) \Rightarrow d)\right) \qquad \begin{pmatrix} (H \models A) \Rightarrow (H \models B) \\ \equiv H \models (A \Rightarrow B) \end{pmatrix}$$
$$\wedge \quad G \models wlp\left(P^N, d\right)$$

$$\Leftrightarrow \quad \forall H.\left(\left(\bigvee_{i=0}^{N-1} \langle G,H\rangle \in [\![P^i]\!]\right) \Rightarrow H \models (wlp(P, false) \Rightarrow d)\right) \qquad \left(\text{Def. } \cup\right)$$
$$\wedge \quad G \models wlp\left(P^N, d\right)$$

$$\Leftrightarrow \quad \forall H.\bigwedge_{i=0}^{N-1}\left(\langle G,H\rangle \in [\![P^i]\!] \Rightarrow H \models (wlp(P, false) \Rightarrow d)\right) \qquad \begin{pmatrix} a \vee b \Rightarrow c \\ \equiv a \Rightarrow c \wedge b \Rightarrow c \end{pmatrix}$$
$$\wedge \quad G \models wlp\left(P^N, d\right)$$

$$\Leftrightarrow \quad \bigwedge_{i=0}^{N-1} \forall H.\left(\langle G,H\rangle \in [\![P^i]\!] \Rightarrow H \models (wlp(P, false) \Rightarrow d)\right) \qquad \begin{pmatrix} \forall x.A \wedge B \\ \equiv \forall x.A \wedge \forall x.B \end{pmatrix}$$
$$\wedge \quad G \models wlp\left(P^N, d\right)$$

$$\Leftrightarrow \quad \bigwedge_{i=0}^{N-1} G \models wlp\left(P^i,\ wlp(P, false) \Rightarrow d\right) \quad \wedge \quad G \models wlp\left(P^N, d\right) \qquad \left(\text{Def. } wlp\right)$$

$$\Leftrightarrow \quad G \models \left(\bigwedge_{i=0}^{N-1} wlp\left(P^i,\ wlp(P, false) \Rightarrow d\right) \quad \wedge \quad wlp\left(P^N, d\right)\right) \qquad \begin{pmatrix} G \models A \wedge G \models B \\ \equiv G \models (A \wedge B) \end{pmatrix}$$

### 7.6.2 $wlp$ of Bounded Programs

Having defined the finite $wlp$ construction for bounded iteration, we now apply it to AGT programs. To illustrate our proposal we will consider ATL transformations, even though the approach is applicable to arbitrary AGT programs.

When considering ATL transformations translated to AGT, we can ignore the final *Cleanup* phase of these transformations since it only affects elements of the input model and trace nodes and therefore does not affect the postcondition which only concerns the output model. As a result, the transformations considered for precondition construction have the following general form:

$$T = \quad R1_{Inst}\!\downarrow ;\ R2_{Inst}\!\downarrow \quad \ldots \quad R1^{\ldots}_{Res}\!\downarrow ;\ R2^{\ldots}_{Res}\!\downarrow \quad \ldots$$

Since in the above we have as-long-as-possible iteration of rules, the $wlp$ construction can theoretically be infinite. As proposed, we avoid this by analysing a bounded version of the transformation obtained by replacing all unbounded iterations with iterations bounded to an arbitrary number $N > 0$ as follows:

$$T_{\leq N} = \quad R1_{Inst}\!\downarrow_N ;\ R2_{Inst}\!\downarrow_N \quad \ldots \quad R1^{\ldots}_{Res}\!\downarrow_N ;\ R2^{\ldots}_{Res}\!\downarrow_N \quad \ldots$$

Then $wlp$ becomes a finite construction and we can compute $wlp\,(T_{\leq N}, d)$. We notice in the above that we only have iterations of rules, so before moving to examples, let us expand the construction $wlp\,(r\!\downarrow_N, d)$ in the specific case of the iteration of a rule $\rho$ as this will ease subsequent explanations.

**Definition 7.18 ($wlp$ for bounded rule iteration).** For any rule $\rho = \langle LHS \hookrightarrow RHS,\ ac_L \rangle$, integer $N > 0$ and graph constraint $d$:

$$\begin{aligned}
wlp\,(\rho, false) &= C_\forall(\rho,\ ac_L \Rightarrow L(\rho, A(\rho, false))) && (wlp \text{ Def. 7.16}) \\
&= \forall(LHS,\ ac_L \Rightarrow false) && (A, L, C_\forall \text{ Def. 7.13, 7.14, 7.15}) \\
&= \forall(LHS,\ \neg ac_L) && (A \Rightarrow false \equiv \neg A \vee false \equiv \neg A) \\
&= \neg\exists(LHS,\ ac_L) && (\forall \text{ Def. 7.3})
\end{aligned}$$

And therefore:

$$
\begin{aligned}
wlp\,(\rho{\downarrow_N}, d) &= \bigwedge_{i=0}^{N-1} wlp\left(\rho^i,\ wlp(\rho, false) \Rightarrow d\right) \ \wedge\ wlp\left(\rho^N, d\right) \\
&= \bigwedge_{i=0}^{N-1} wlp\left(\rho^i,\ \neg\exists(LHS,\ ac_L) \Rightarrow d\right) \ \wedge\ wlp\left(\rho^N, d\right) \\
&= \bigwedge_{i=0}^{N-1} wlp\left(\rho^i,\ \exists(LHS,\ ac_L) \vee d\right) \ \wedge\ wlp\left(\rho^N, d\right)
\end{aligned}
$$

□

After applying $Pre = wlp\,(T_{\leq N}, d)$, the weakest liberal precondition *Pre* should be a constraint over the input metamodel. However as seen in the construction above, the condition will contain the *LHS* graphs of rules. These graphs may contain trace nodes and elements of the output metamodel. But input models of the transformation can only contain elements of the input metamodel and could never contain trace nodes or elements of the output metamodel. Therefore conditions nested in *Pre* that contain trace nodes or elements of the output metamodel can never be satisfied : they may be replaced with $false$ and *Pre* can be simplified accordingly. This simplification does not alter the semantics of *Pre* and is not needed in theory, however without it *Pre* would be unnecessarily complex, both for human interpretation as well as for machine processing. Next we show some examples of $wlp\,(T_{\leq N}, d)$.

**Example 7.8 ($wlp$ of a bounded ATL transformation).** We consider a modified version of the example from Figure 6.1 where we remove attribute-related aspects and only keep structural aspects. The considered ATL transformation is therefore the following:

```
1  rule R1 {
2  from s  : IN!A
3     (s.refB->exists(b | true))
4  to t1 : OUT!D
5     (refD <- t2,
6      refE <- s.refB),
7   t2 : OUT!D }
```

```
8  rule R2 {
9  from s : IN!B
10 to t : OUT!E
11    (refD <- thisModule.resolveTemp
12                  (s.refA, 't2') ) }
```

Then we translate this purely structural ATL transformation to the following AGT transformation:

$$T = R1_{Inst}\downarrow;\ R2_{Inst}\downarrow;\ R1_{Res}^{t1,refD}\downarrow;\ R1_{Res}^{t1,refE}\downarrow;\ R2_{Res}^{t,refD}\downarrow$$

By choosing $N = 1$, we consider a bounded version of the transformation where there no longer is unbounded iteration:

$$T_{\leq 1} = R1_{Inst}\downarrow_1;\ R2_{Inst}\downarrow_1;\ R1_{Res}^{t1,refD}\downarrow_1;\ R1_{Res}^{t1,refE}\downarrow_1;\ R2_{Res}^{t,refD}\downarrow_1$$

Next, we consider a postcondition and construct its corresponding weakest liberal precondition.

$$Post_0 = \exists\left(\begin{array}{c}\boxed{d1{:}D}\\ \downarrow{\scriptstyle refD}\\ \boxed{d2{:}D}\end{array}\right)$$

$$Pre_0 = wlp\,(T_{\leq 1},\ Post_0) = \exists\left(\underbrace{\boxed{s{:}A}}_{a}\ ,\quad \underbrace{\exists\left(\begin{array}{c}\boxed{s{:}A}\\ \downarrow{\scriptstyle refB}\\ \boxed{b{:}B}\end{array}\right)}_{b}\right)$$

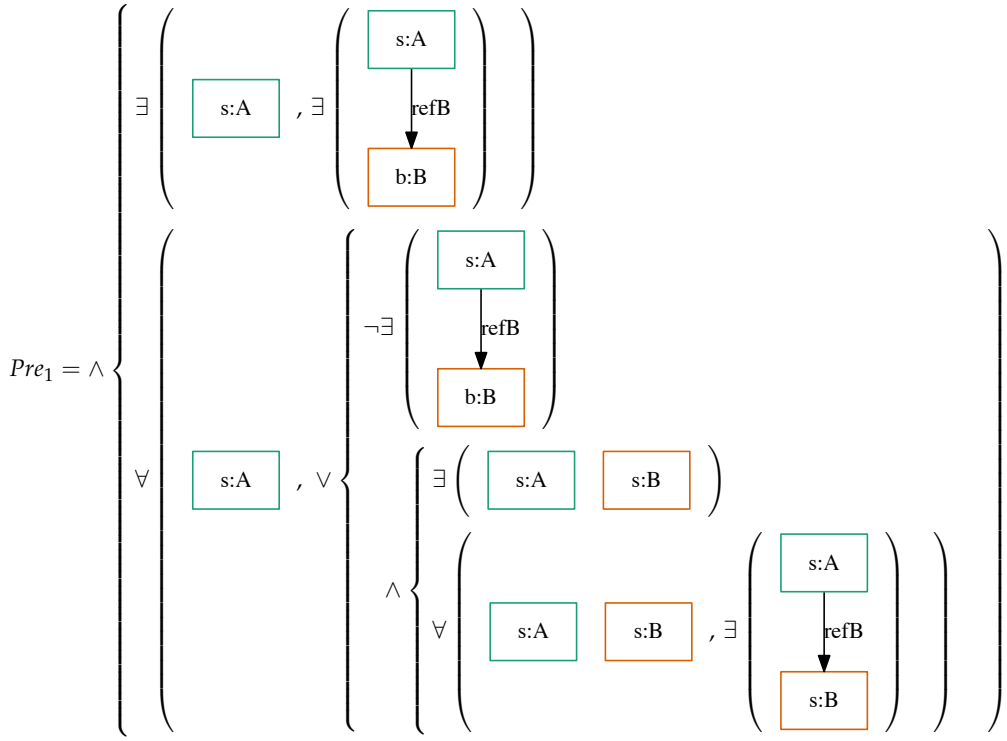
Note that this precondition and all the ones shown in subsequent examples were obtained with our implementation of $wlp$ that will be discussed later in this chapter. To explain the precondition intuitively, we notice that the ATL rule **R1** produces 2 instances of $D$ that satisfy the postcondition. Therefore to ensure the postcondition we must ensure that **R1** is triggerred at least once. Indeed, $Pre_0$ requires the existance of **R1**'s source pattern (part $a$) such that the guard is satisfied (part $b$). This ensures that **R1** is triggerred and that the postcondition is satisfied.

□

**Example 7.9 ($wlp$ with ATL resolve mechanism).** With the same transformation of Example 7.8, we now consider the postcondition $Post_1$ and apply $wlp$:

$$Post_1 = \exists\left(\boxed{d{:}D}\xrightarrow{refE}\boxed{e{:}E}\right)$$

$Pre_1 = wlp\,(T_{\leq 1},\ Post_1)$




In the representation of $Pre_1$, the coloring of nodes is used to represent the morphisms between nesting levels of the condition. Two nodes in two different nesting levels are represented with the same color when they are mapped by the morphism from the upper nesting level to the lower nesting level.

To be able to interpret this precondition and assess its validity, we need to notice a particular recurring pattern in its structure. In several places, notably at the root of the condition, we observe the following form:

$$\begin{aligned} &\exists\,(C,\quad c) \\ \wedge\ &\forall\,(C, \neg c \vee d) \end{aligned}$$

Since both the $\exists$ and $\forall$ of the conjunction have the exact same conclusion graph $C$, then we can conjunct the nested condition of $\forall$ into the $\exists$ as follows:

$$\begin{aligned} &\exists\,(C, c \wedge (\neg c \vee d)) \\ \wedge\ &\forall\,(C, \quad \neg c \vee d) \end{aligned} \quad \equiv \quad \begin{aligned} &\exists\,(C,\quad c \wedge d) \\ \wedge\ &\forall\,(C, \neg c \vee d) \end{aligned}$$

Rewriting $Pre_1$ in the above form gives the precondition in Figure 7.3 which is easier to understand. In the upper part of the condition, we notice that part *a* requires the existence of the source pattern of **R1** such that its guard is verified thus ensuring that **R1** is triggered. Similarly, part *b* ensures that **R2** is triggered and that a resolve between **R1** and **R2** is possible based on *refB*. These conditions combined ensure that:

1. **R1** will produce an object *d:D*.
2. **R2** will produce an object *e:E*.
3. The resolve mechanism will create *refE* from *d* to *e*.

As a result, $Pre_1$ guarantees that $Post_1$ will be satisfied in the output model.

□

**Example 7.10 ($wlp$ of unsatisfiable postcondition).** With the same transformation of Example 7.8 ,we now consider another postcondition, $Post_2$:

$$Post_2 = \exists \left( \begin{array}{c} \boxed{d:D} \\ refD \;\; refE \\ \boxed{e:E} \end{array} \right)$$

Applying the analysis we obtain $wlp(T_{\leq 1}, Post_2) = false$ indicating that no input model could ever lead to the satisfaction of the postcondition. In the ATL transformation, **R1** creates objects *t1:D* and *t2:D*, and creates *refE* outgoing from *t1*. **R2** performs a non-default resolve and uses **t2** instead of **t1** to initialize *refD*, therefore the cycle in $Post_2$ could never be produced by this transformation, which justifies the precondition $false$.

□

### 7.6.3 Scope of the Bounded Weakest Liberal Precondition

$wlp\,(T_{\leq N}, d)$ is the weakest liberal precondition of $T_{\leq N}$ relative to $d$, but *not* necessarily that of $T$. While exceptions may exist, in general $wlp\,(T_{\leq N}, d) \neq wlp\,(T, d)$. However $wlp\,(T_{\leq N}, d)$ can still be useful for reasoning about $T$ within a certain scope. In $T_{\leq N}$, $N$ is the maximum number of times each rule is iterated. In $T$, iterations are not bounded, but for all executions of $T$ where each rule is executed less than or exactly $N$ times, $T$ is equivalent to $T_{\leq N}$, *i.e.* $T \sim T_{\leq N}$. Only in that case is $wlp\,(T_{\leq N}, d)$ valid for $T$. For this reason the interpretation and usage of $wlp\,(T_{\leq N}, d)$ for $T$ is not straightforward. We distinguish two situations:

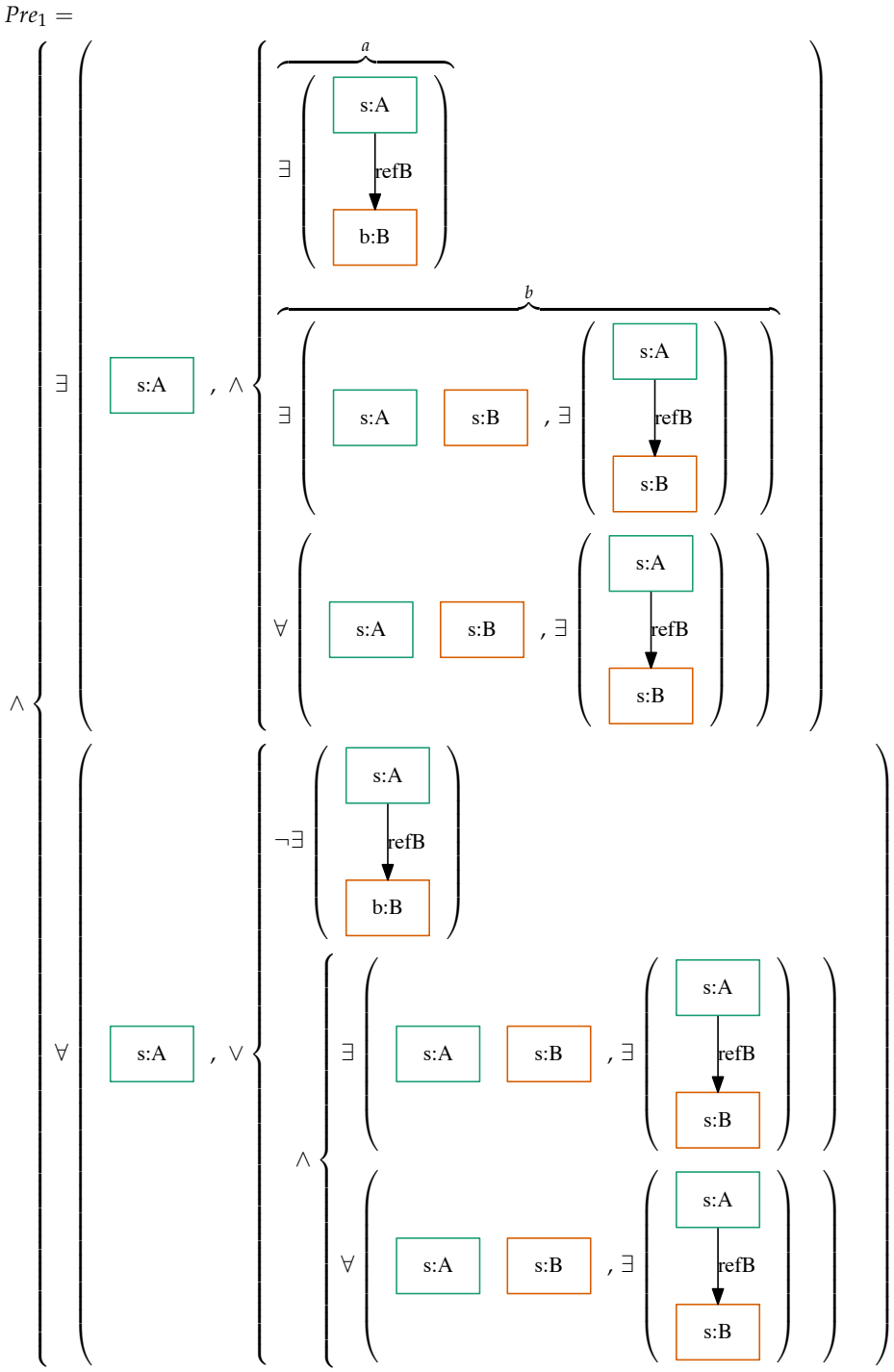



Figure 7.3: $Pre_1 = wlp\,(T_{\leq 1}, Post_1)$

1. $wlp\,(T_{\leq N}, d) = false$

2. $wlp\,(T_{\leq N}, d) \neq false$

**When the Precondition is $false$**

$wlp\,(T_{\leq N}, d) = false$ means that it is impossible to satisfy the postcondition with $T_{\leq N}$. However the postcondition may in fact be satisfiable with a higher bound $M > N$. To illustrate this, let us consider the ATL transformation from Example 7.8 and the postcondition $Post_3 = \exists \left( \boxed{:D} \boxed{:D} \boxed{:D} \right)$ which requires the existence of 3 objects of type $D$ in the output model. In the transformation, objects of type $D$ are only created by **R1**: each application of **R1** produces 2 instances of $D$. Setting the bound $N = 1$ forces $R1_{Inst}$ (and by consequence **R1**) to be triggered only once, therefore $T_{\leq 1}$ can only produce up to 2 instances of $D$. Under these constraints the postcondition can never be satisfied hence $wlp\,(T_{\leq 1}, Post_3) = false$. Setting $N = 2$ would give a different precondition which ensures that **R1** is triggered twice and ensures by consequence the satisfaction of $Post_3$. However in other cases such as Example 7.10 a postcondition that is unsatisfiable with $T_{\leq N}$ may also be unsatisfiable with $T$.

To work around this problem, when $wlp\,(T_{\leq N}, d) = false$ we can try to increase the bound $N$ to see if we can construct a different precondition. If not, then we must manually analyse the postcondition and the transformation to determine if the postcondition is indeed unsatisfiable with $T$.

**When the Precondition is Non-$false$**

Let us consider $wlp\,(T_{\leq N}, d) \neq false$ and assume that an input model $G$ was produced such that $G \models wlp\,(T_{\leq N}, d)$. If we execute $T$ (not $T_{\leq N}$) to obtain a result graph $H$, there are 2 possible execution scenarios:

1. Each rule is triggered less than (or exactly) $N$ times. In that case $T \sim T_{\leq N}$ and we are sure that $H \models d$.

2. Some rules are triggered $M$ times where $M > N$. In that case $T \nsim T_{\leq N}$ and we cannot claim for sure that $H$ satisfies $d$. Therefore it is necessary to evaluate $d$ over $H$ to check that it is satisfied. We note that it is likely that $H \models d$ because $wlp\,(T_{\leq N}, d)$ is likely to be effective with $T$ as well.
   If however $H \not\models d$, then we can compute $wlp\,(T_{\leq M}, d)$ and produce an input model $G' \models wlp\,(T_{\leq M}, d)$. The same reasoning is then iterated with $G'$ to make sure that the result satisfies $d$.

We illustrate this situation with the following example.

**Example 7.11 (Scope-sensitive Precondition).** A simple way to illustrate the scoping problem is with the following postcondition stating that there should not exist

3 or more objects of type $E$ *i.e.* there should exist at most 2 objects of type $E$ in the output model.

$$Post_4 = \neg\exists \left( \boxed{\text{e1:E}} \quad \boxed{\text{e2:E}} \quad \boxed{\text{e3:E}} \right)$$

Computing $wlp$ we find that $wlp\,(T_{\leq 1}, Post_4) = wlp\,(T_{\leq 2}, Post_4) = true$ meaning that any input model given to $T_{\leq 1}$ and $T_{\leq 2}$ leads to the satisfaction of the postcondition. This is because **R2** can never be triggered beyond 2 times in these transformations, so there can never be more than 2 objects of type $E$ in the output model regardless of the input model. However this is surely not the case for $T$.

If we consider the transformation $T$ with the input graph $G = \boxed{b1:B}\ \boxed{b2:B}\ \boxed{b3:B}$, **R2** will be triggered 3 times and the output model will contain 3 objects of type $E$. Therefore the postcondition will be violated even though the precondition was satisfied. This is because this execution is beyond the scope of validity of $wlp\,(T_{\leq 1}, Post_4)$ and $wlp\,(T_{\leq 2}, Post_4)$.

Given that **R2** was triggered 3 times in that execution, we can iterate the analysis with $N = 3$ and compute $wlp\,(T_{\leq 3}, Post_4) \neq true$. In that case we get $G \not\models wlp\,(T_{\leq 3}, Post_4)$ which means that the postcondition cannot be satisfied with $G$.

□

### 7.6.4 From Bounded $wlp$ to Non-Bounded Liberal Preconditions

In the previous section, we have ensured the validity of $wlp\,(T_{\leq N}, d)$ for $T$ by checking *a posteriori* of execution that $H \models d$. Now we propose to ensure the applicability of $wlp\,(T_{\leq N}, d)$ to $T$ *a priori* by complementing $wlp\,(T_{\leq N}, d)$ with a constraint that limits the number of possible triggers of each rule to $N$. In this way we ensure that for any model that satisfies the precondition, we have $T \sim T_{\leq N}$ and therefore the precondition is valid for $T$. We propose to call such a construction $scopedWlp\,(T_{\leq N}, d)$ because it encodes the scope of validity in the precondition itself. We will then show that $scopedWlp\,(T_{\leq N}, d)$ is a (non-weakest) liberal precondition of $T$ which is definite formal proof that the scoped precondition is valid for $T$.

Naturally, $scopedWlp$ is largely the same as $wlp$ except for the handling of bounded program iteration $P\downarrow_N$. We complement $wlp\,(P\downarrow_N, d)$ with a condition that prevents the program from executing more than $N$ times. From [Habel *et al.*, 2006a]

we know that for any program $P$, the condition $\neg wlp\,(P, false)$ ensures the existence of a result *i.e.* the applicability of the program $P$. In formal terms, for any graph $G$:

$$\begin{array}{lcll} G \models \neg wlp\,(P, false) & \Leftrightarrow & P \text{ is executable over } G & \Leftrightarrow \quad \exists H.\, \langle G, H\rangle \in [\![P]\!] \\ G \models \quad wlp\,(P, false) & \Leftrightarrow & P \text{ is not executable over } G & \Leftrightarrow \quad \neg\exists H.\, \langle G, H\rangle \in [\![P]\!] \end{array}$$

To illustrate $wlp\,(P, false)$ we expand the construction for rules in the following example.

**Example 7.12 (Existence of a result for rules).** The condition ensuring the existence of a result for a rule $\rho = \langle LHS \hookrightarrow RHS, ac_L\rangle$ expands as follows:

$$\neg wlp\,(\rho, false) = \exists\,(LHS, ac_L) \qquad \text{(expansion details in Def. 7.18)}$$

We observe that a result exists (*i.e.* the rule is executable) when there exists a match of the $LHS$ that satisfies the application condition $ac_L$. Conversely, a result doesn't exist (*i.e.* the rule is not executable) when there doesn't exist such a match. This reasoning is specific to rules and is generalised to arbitrary programs $P$ with $wlp\,(P, false)$.

□

Therefore to ensure that a program $P$ cannot apply beyond $N$ times, we can use the condition $wlp\left(P^{N+1}, false\right)$. To convince ourselves of this, we can prove the following theorem.

**Theorem 7.2 (Executability beyond $N$).** For all graphs $G$, programs $P$, integers $N > 0$

$$\begin{array}{rc} & G \models wlp\left(P^N, false\right) \;\Rightarrow\; G \models wlp\left(P^{N+1}, false\right) \\ \equiv & P^N \text{ is not executable on } G \;\Rightarrow\; P^{N+1} \text{ is not executable on } G \end{array}$$

And by induction over $N$ we can state that:

$$\begin{array}{rc} & G \models wlp\left(P^N, false\right) \;\Rightarrow\; \forall M > N.\, G \models wlp\left(P^M, false\right) \\ \equiv & P^N \text{ is not executable on } G \;\Rightarrow\; \forall M > N.\, P^M \text{ is not executable on } G \end{array}$$

**Proof.**

$$
\begin{array}{lll}
 & G \models wlp\left(P^N, false\right) & \\
\Leftrightarrow & \neg\exists H.\ \langle G, H\rangle \in [\![P^N]\!] & \\
\Rightarrow & \forall H_1.\left(\neg\exists H.\left(\langle G, H\rangle \in [\![P^N]\!] \ \wedge\ \langle H, H_1\rangle \in [\![P]\!]\right)\right) & \\
\Rightarrow & \forall H_1.\left(\neg \langle G, H_1\rangle \in [\![P^N]\!] \circ [\![P]\!]\right) & \\
\Rightarrow & \forall H_1.\left(\neg \langle G, H_1\rangle \in [\![P^{N+1}]\!]\right) & \\
\Rightarrow & \neg\exists H_1.\left(\langle G, H_1\rangle \in [\![P^{N+1}]\!]\right) & \\
\Rightarrow & G \models wlp\left(P^{N+1}, false\right) &
\end{array}
$$

□

With the above theorem, we have shown that condition $wlp\left(P^{N+1}, false\right)$ ensures that $P$ cannot be applied beyond $N$ times. We thus define $scopedWlp$ as follows.

**Definition 7.19 (Liberal precondition with scoping).** For any graph constraint $d$, rule $\rho$, programs $P, Q$ and integer $N > 0$, $scopedWlp$ is defined inductively over the structure of programs as follows:

$$
\begin{array}{lcl}
scopedWlp\,(Skip, d) & = & wlp\,(Skip, d) \\
scopedWlp\,(\rho, d) & = & wlp\,(\rho, d) \\
scopedWlp\,(\ (P; Q)\ , d) & = & wlp\,(\ (P; Q)\ , d) \\
scopedWlp\,(P{\downarrow_N}, d) & = & wlp\,(P{\downarrow_N}, d)\ \wedge\ wlp\left(P^{N+1}, false\right)
\end{array}
$$

□

Now we prove that $scopedWlp\,(T_{\leq N}, d)$ is a (non-weakest) liberal precondition of $T$ relative to $d$ which means that it ensures the satisfaction of $d$ as a postcondition of the unbounded $T$.

**Theorem 7.3 ($scopedWlp$ is a liberal precondition of $T$).** For any program $T$, integer $N > 0$ and graph constraint $d$, $scopedWlp\,(T_{\leq N}, d)$ is a liberal precondition of $T$. In formal terms, for all graphs $G$:

$$G \models scopedWlp\,(T_{\leq N}, d)\ \Rightarrow\ G \models wlp\,(T, d)\ \Rightarrow\ \forall H.\ \langle G, H\rangle \in [\![T]\!] \Rightarrow H \models d$$

**Proof.** For all kinds of programs except iteration, the proof entails from the fact that $scopedWlp \equiv wlp$. For iteration we need to prove that the scoped precondition of bounded iteration is a (non-weakest) liberal precondition of unbounded iteration, *i.e.* for all graphs $G$:

$$G \models scopedWlp\,(P\downarrow_N, d) \quad \Rightarrow \quad G \models wlp\,(P\downarrow, d)$$

First we recall the definitions of $[\![P\downarrow]\!]$ and $[\![P\downarrow_N]\!]$:

$$\begin{aligned}
[\![P\downarrow]\!] &= \{\langle G,H\rangle \in [\![P]\!]^* \mid \neg\exists M.\ \langle H,M\rangle \in [\![P]\!]\} \\
&= \left\{\langle G,H\rangle \in \bigcup_{i=0}^{\infty} [\![P]\!]^i \mid \neg\exists M.\ \langle H,M\rangle \in [\![P]\!]\right\} \\
&= \left\{\langle G,H\rangle \in \bigcup_{i=0}^{\infty} [\![P^i]\!] \mid \neg\exists M.\ \langle H,M\rangle \in [\![P]\!]\right\} \\
&= \quad \left\{\langle G,H\rangle \in \bigcup_{i=0}^{N} [\![P^i]\!] \mid \neg\exists M.\ \langle H,M\rangle \in [\![P]\!]\right\} \\
&\quad \cup \left\{\langle G,H\rangle \in \bigcup_{i=N+1}^{\infty} [\![P^i]\!] \mid \neg\exists M.\ \langle H,M\rangle \in [\![P]\!]\right\} \\
[\![P\downarrow_N]\!] &= \left\{\langle G,H\rangle \in \bigcup_{i=0}^{N-1} [\![P^i]\!] \mid \neg\exists M.\, \langle H,M\rangle \in [\![P]\!]\right\} \cup [\![P^N]\!]
\end{aligned}$$

Next we deduce the consequences of $G \models wlp\left(P^{N+1}, false\right)$ on $[\![P\downarrow]\!]$ and $[\![P\downarrow_N]\!]$.

$$\begin{aligned}
& G \models wlp\left(P^{N+1}, false\right) \\
\Rightarrow\quad & G \models wlp\left(P^{N+1}, false\right) \ \wedge\ \forall M > N+1.\ G \models wlp\left(P^{M}, false\right) \quad \text{(Thm. 7.2)} \\
\Rightarrow\quad & \forall M \geq N+1.\ G \models wlp\left(P^{M}, false\right) \\
\Rightarrow\quad & \forall M \geq N+1.\ \neg\exists H.\ \langle G,H\rangle \in [\![P^M]\!] \\
\Rightarrow\quad & \forall M \geq N+1.\ \left\{\langle G,H\rangle \in [\![P^M]\!]\right\} = \phi \\
\Rightarrow\quad & \left\{\langle G,H\rangle \in \bigcup_{i=N+1}^{\infty} [\![P^i]\!]\right\} = \phi \\
\Rightarrow\quad & \{\langle G,H\rangle \in [\![P\downarrow]\!]\} = \left\{\langle G,H\rangle \in \bigcup_{i=0}^{N} [\![P^i]\!] \mid \neg\exists M.\ \langle H,M\rangle \in [\![P]\!]\right\}
\end{aligned}$$

And

$$G \models wlp\left(P^{N+1}, false\right)$$

$$
\begin{array}{lll}
\Leftrightarrow & \neg\exists M.\ \langle G, M\rangle \in [\![P^{N+1}]\!] & (wlp \text{ Def. 7.11}) \\
\Leftrightarrow & \neg\exists M.\ \langle G, M\rangle \in [\![P^{N}]\!] \circ [\![P]\!] & \\
\Leftrightarrow & \neg\exists M.\ \left(\exists H.\ \left(\langle G, H\rangle \in [\![P^{N}]\!] \wedge \langle H, M\rangle \in [\![P]\!]\right)\right) & \\
\Leftrightarrow & \forall M.\ \left(\neg\exists H.\ \left(\langle G, H\rangle \in [\![P^{N}]\!] \wedge \langle H, M\rangle \in [\![P]\!]\right)\right) & \\
\Leftrightarrow & \forall M.\ \left(\forall H.\ \left(\neg \langle G, H\rangle \in [\![P^{N}]\!] \vee \neg \langle H, M\rangle \in [\![P]\!]\right)\right) & \\
\Leftrightarrow & \forall H.\ \left(\neg \langle G, H\rangle \in [\![P^{N}]\!] \vee \forall M.\ (\neg \langle H, M\rangle \in [\![P]\!])\right) & \\
\Leftrightarrow & \forall H.\ \left(\neg \langle G, H\rangle \in [\![P^{N}]\!] \vee \neg\exists M.\ \langle H, M\rangle \in [\![P]\!]\right) & \\
\Leftrightarrow & \forall H.\ \left(\langle G, H\rangle \in [\![P^{N}]\!] \Rightarrow \neg\exists M.\ \langle H, M\rangle \in [\![P]\!]\right) & \\
\Rightarrow & \{\langle G, H\rangle \in [\![P{\downarrow_N}]\!]\} = & \\
& \left\{ \langle G, H\rangle \mid \begin{array}{l} \langle G, H\rangle \in \bigcup_{i=0}^{N-1} [\![P^i]\!] \wedge \neg\exists M.\ \langle H, M\rangle \in [\![P]\!] \\ \vee \langle G, H\rangle \in [\![P^{N}]\!] \end{array} \right\} & \\
\Rightarrow & \{\langle G, H\rangle \in [\![P{\downarrow_N}]\!]\} = & \\
& \left\{ \langle G, H\rangle \mid \begin{array}{l} \langle G, H\rangle \in \bigcup_{i=0}^{N-1} [\![P^i]\!] \wedge \neg\exists M.\ \langle H, M\rangle \in [\![P]\!] \\ \vee \langle G, H\rangle \in [\![P^{N}]\!] \quad \wedge \neg\exists M.\ \langle H, M\rangle \in [\![P]\!] \end{array} \right\} & \\
\Rightarrow & \{\langle G, H\rangle \in [\![P{\downarrow_N}]\!]\} = \left\{ \langle G, H\rangle \in \bigcup_{i=0}^{N} [\![P^i]\!] \mid \neg\exists M.\ \langle H, M\rangle \in [\![P]\!] \right\} &
\end{array}
$$

Therefore

$$
\begin{array}{l}
G \models wlp\left(P^{N+1}, false\right) \Rightarrow \\
\{\langle G, H\rangle \in [\![P{\downarrow}]\!]\} = \{\langle G, H\rangle \in [\![P{\downarrow_N}]\!]\} = \left\{ \langle G, H\rangle \in \bigcup_{i=0}^{N} [\![P^i]\!] \mid \neg\exists M.\ \langle H, M\rangle \in [\![P]\!] \right\}
\end{array}
$$

And finally, we can prove that *scopedWlp* is a liberal precondition as follows.

$$
\begin{array}{lll}
 & G \models scopedWlp\,(P{\downarrow_N}, d) & \\
\Leftrightarrow & G \models wlp\,(P{\downarrow_N}, d) \quad \wedge \quad G \models wlp\left(P^{N+1}, false\right) & \\
\Leftrightarrow & \forall H.\ \langle G, H\rangle \in [\![P{\downarrow_N}]\!] \Rightarrow H \models d \quad \wedge \quad G \models wlp\left(P^{N+1}, false\right) & \\
\Rightarrow & \forall H.\ \langle G, H\rangle \in [\![P{\downarrow}]\!] \Rightarrow H \models d &
\end{array}
$$

$$\Rightarrow \quad G \models wlp\,(P\downarrow, d)$$

□

Next we illustrate $scopedWlp$ over some of the previous examples.

**Example 7.13 (Scoped liberal precondition).** We consider the same postcondition from Example 7.11.

$$Post_4 = \neg\exists\left(\begin{array}{|c|}\hline \text{e1:E} \\ \hline\end{array}\quad\begin{array}{|c|}\hline \text{e2:E} \\ \hline\end{array}\quad\begin{array}{|c|}\hline \text{e3:E} \\ \hline\end{array}\right)$$

In Example 7.11 we had obtained $wlp\,(T_{\leq 1}, Post_4) = wlp\,(T_{\leq 2}, Post_4) = true$, but that precondition was not always valid for $T$. We now apply $scopedWlp$. The result $scopedWlp\,(T_{\leq 1}, Post_4)$ is the following.

$scopedWlp\,(T_{\leq 1}, Post_4) =$

$$\wedge\left\{\begin{array}{l}
\neg\exists\left(\begin{array}{|c|}\hline \text{s:A} \\ \hline\end{array}\;,\;\wedge\left\{\begin{array}{l}
\exists\left(\begin{array}{|c|}\hline \text{s:A} \\ \hline \text{refB} \\ \hline \text{b:B} \\ \hline\end{array}\right) \\
\vee\left\{\begin{array}{l}
\exists\left(\begin{array}{|c|}\hline \text{s:A} \\ \hline\end{array}\;\begin{array}{|c|}\hline \text{s:A} \\ \hline\end{array}\;,\;\exists\left(\begin{array}{|c|}\hline \text{s:A} \\ \hline\end{array}\;\begin{array}{|c|}\hline \text{s:A} \\ \hline \text{refB} \\ \hline \text{b:B} \\ \hline\end{array}\right)\right) \\
\exists\left(\begin{array}{|c|}\hline \text{s:A} \\ \hline\end{array}\;\begin{array}{|c|}\hline \text{s:B} \\ \hline\end{array}\;,\;\exists\left(\begin{array}{|c|}\hline \text{s:A} \\ \hline\end{array}\;\begin{array}{|c|}\hline \text{s:B} \\ \hline\end{array}\;\begin{array}{|c|}\hline \text{s:B} \\ \hline\end{array}\right)\right)
\end{array}\right.
\end{array}\right.\right) \\
\neg\exists\left(\begin{array}{|c|}\hline \text{s:B} \\ \hline\end{array}\;,\;\exists\left(\begin{array}{|c|}\hline \text{s:B} \\ \hline\end{array}\;\begin{array}{|c|}\hline \text{s:B} \\ \hline\end{array}\right)\right)
\end{array}\right.$$

Since the precondition obtained with $wlp$ was $true$, the above precondition obtained with $scopedWlp$ expresses solely the scoping condition: rules should not be triggered more than once. It expresses the fact that there should not be 2 objects of type $A$ that satisfy `R1`'s application condition and there should not be 2 objects of type $B$ that trigger $R2$. This is what ensures that each rule can never be triggered more than once.

□

As illustrated by the example, $scopedWlp\,(T_{\leq N}, d)$ is more constraining that $wlp\,(T_{\leq N}, d)$ because it adds the scoping condition. The scoping condition essentially limits the maximum size of input models to ensure that iterated programs cannot be triggerred beyond $N$ times. This ensures that the behavior of $T$ is identical to the behavior of $T_{\leq N}$ and therefore ensures the validity of the bounded precondition for the unbounded transformation $T$.

### 7.6.5 Discussion

In this section we have shown 2 finite precondition constructions:

(1) $wlp\,(T_{\leq N}, d)$ computes the weakest liberal precondition of $T_{\leq N}$ relative to $d$, however this precondition may not always apply to the unbounded transformation $T$ *i.e.* there may be input models that satisfy the precondition but do not ensure the postcondition after execution of $T$. For this reason it is necessary to verify the satisfaction of the postcondition *a posteriori* of execution.

(2) $scopedWlp\,(T_{\leq N}, d)$ is similar to the first construction but complements it with a *scoping* constraint that ensures that it produces a (non-weakest) liberal precondition of $T$. This means that this construction always guarantees the satisfaction of the postcondition for the unbounded transformation $T$. However the precondition it produces is more constraining than that of construction (1) because it imposes bounds on the size of input models.

For the usage in the context of test generation construction (1) may be a better choice because it guides test generation without constraining test models too much. However *a posteriori* verification of the result is necessary. In comparison, construction (2) requires no *a posteriori* verification because it guarantees the validity of resulting models, but the precondition it produces is more constrained than construction (1) and imposes size limits over input models. The consequence for test generation is that in certain situations no test model can be found if the precondition is too constraining. For timing constraints, it was not possible to assess both constructions thoroughly with respect to the targeted usage context of test generation. This comparison is left as future work.

Beyond the context of test generation, the constructions presented and the accompanying formal proofs constitute a solid basis for formal analyses of model transformations in future work. As will be detailed in Chapter 11, such analyses include manual and automated formal proof of correctness of model transformations.

Before moving to the next chapter which will discuss the implementation and scalability of the proposed constructions, we discuss related work in the next section.

## 7.7 Related Work

Many works in the field of formal proof of correctness of traditional imperative programs have proposed ways to transform postconditions into preconditions starting with their original introduction in [Hoare, 1969]. The considered programs manipulate numeric variables and even pointers to memory locations. While it is possible to encode model transformation in this manner and transform conditions with traditional methods, we have found the AGT framework to be more suitable for the problem at hand because it acknowledges the *graph*-like nature of models and defines condition transformations accordingly.

Within the field of AGT, an alternate construction to the classical theoretical one that we used has been proposed in [Cabot *et al.*, 2010a]. This work considers transformations consisting of AGT rules but considers application conditions, postconditions and preconditions in OCL instead of NGC as classically done. OCL postconditions of a rule are then transformed to preconditions in two steps. First the rule is analysed to extract so-called *atomic updates* that the rule performs such as (roughly) the deletion/creation of links and the deletion/creation of objects. Then, a set of textual replacement patterns are applied to the OCL precondition. For each atomic update performed by the rule, the textual replacement patterns rewrite the postcondition into a corresponding precondition.

Conceptually, this *OCL-based approach* bears several similarities with the *NGC-based approach* that we presented in this chapter. In essence, the OCL textual replacement patterns *undo* the effects of the atomic updates of the rule from the postcondition, which is similar to step $L$ in the NGC-based construction. Additionally, the textual replacement patterns consider *all* possible ways in which an atomic update may affect expressions in the postcondition, which is similar to enumerating the overlaps of the rule $RHS$ with the postcondition in step $A$ of the NGC-based construction. However, being tailored to OCL, this approach supports more expressive postconditions involving constraints on numerical attributes and cardinalities of collections. Support of some of these constraints has been introduced to the NGC-based translation in different ways such as in [Poskitt, 2013] and [Deckwerth and Varró, 2014] however the theoretical concepts involved are fairly complex and we did not tackle them in the scope of this thesis.

Nonetheless, the NGC-based translation provides other advantages. It is proven to be complete and correct while the OCL-based approach does not provide a proof of completeness and correctness yet. Moreover, the fact that the NGC-based approach represents conditions as graphs will allow us to easily introduce simplifications based on the ATL semantics in the following chapter. This will be detailed further in Section 8.5.

## 7.8 Conclusion

In this chapter we have detailed the transformation of a postcondition of an AGT transformation into a precondition ensuring the satisfaction of the postcondition. This transformation was based on the *weakest liberal precondition* constructions $wlp$ of the AGT framework. Since its application to programs involving iteration may yield an infinite construction, we have proposed a way to make the construction finite by bounding the number of iterations. To do so we have introduced a new *bounded iteration* construct and its corresponding finite $wlp$. The precondition computed in this manner is not always valid for the initial unbounded transformation. For this reason we have introduced an alternate $scopedWlp$ construction that yields a liberal precondition that is not the *weakest*, but that is always valid for the unbounded transformation and that always guarantees the satisfaction of the postcondition. For all these new concepts we have proposed formal definitions and have proven their properties. This set of formal contributions is not specific to ATL and is applicable to arbitrary structural AGT transformations.

In the next chapter, we will discuss the implementation of the proposed constructions and introduce strategies to address their computational complexity.

# Chapter 8

# Implementation and Simplification Strategies for $wlp$

## Contents

## 8.1 Introduction

In the previous chapter we presented the theoretical translation of postconditions to preconditions using the $wlp$ construction. This chapter focuses on its implementation and the complexity issues that arise. The core of the $wlp$ construction is the *Shift* operation which relies on enumerating all possible ways in which the transformation can contribute to the satisfaction of the postcondition. The combinatorial nature of this construction yields very large and complex preconditions that grow exponentially as the successive rules of the transformation are analysed.

To address this problem we propose several strategies that jointly help reduce combinatorial explosion. Some of these strategies are specific to ATL transformations while others can be applied for arbitrary AGT transformations. We also propose a modified construction of $wlp$ which is equivalent to the original one but allows an early application of the proposed simplification strategies. This helps avoiding unnecessary computations early on. Finally, we propose a parallel implementation and a memory management strategy allowing the algorithm to perform faster and avoid memory exhaustion. Except for the ATL-specific simplification strategies, all of these contributions are generally applicable to arbitrary structural AGT transformations.

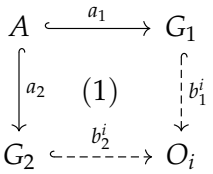
In the following sections we start by detailing the implementation of graph overlapping which is the basis of the *Shift* construction in Section 8.2 and we discuss the complexity of this operation in Section 8.3. Then we propose our simplification strategies in Section 8.4 and our alternate construction in Section 8.5. Finally we discuss parallelisation and memory management aspects in Section 8.6.

## 8.2 Implementing Graph Overlapping

The core of $wlp$ is the *Shift* construction which enumerates all possible overlaps of two graphs. Implementing this enumeration into an algorithm is not straightforward. We have taken inspiration from an existing overlapping algorithm in the *AGG* framework [AGG, accessed 2015] and adapted it to the *Henshin* framework taking particular care to the aspects of node type inheritance as will be explained.

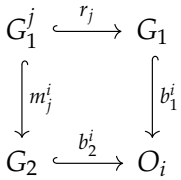
Let us consider the construction of $Shift(A \overset{a_1}{\hookrightarrow} G_1, A \overset{a_2}{\hookrightarrow} G_2)$.




As explained in the definition of *Shift*, the anchor $(a_1, a_2)$ specifies which nodes should necessarily be identified in all overlaps $O_i$. Let $G_1^{anchored} = \{a_1(e); e \in dom(a_1) \cap dom(a_2)\}$ be the set of anchored elements in $G_1$, *i.e.* the images through $a_1$ of the elements of $A$ that are mapped simultaneously by $a_1$ and $a_2$.

The idea is to enumerate all subgraphs of $G_1$ which contain the anchored elements $G_1^{anchored}$. For each such subgraph $G_1^j$, we construct the inclusion $r_j : G_1^j \hookrightarrow G_1$ and a temporary graph transformation rule $R_j =< r_j, true >$. Then we construct a partial morphism $m_j$ which maps each anchored element to its counterpart in $G_2$. Next, we use the Henshin execution engine to complete $m_j$ by matching the remaining elements of $G_1^j$ to $G_2$. If a complete morphism cannot be found, the subgraph $G_1^j$ is discarded and we move to the next subgraph. If a complete morphism $m_j^i$ is found, it constitutes a match of the rule $R_j$ in $G_2$. Executing $R_j$ with a match $m_j^i$ complements $G_2$ with the remaining elements $G_1 - G_1^j$ resulting in an overlap $O_i$. Iterating the process for all matches $m_j^i$ and all subgraphs $G_1^j$ produces all overlaps $O_i$.




This algorithm works when all nodes of $G_2$ have types that inherit the types of nodes in $G_1$. If it is not the case, *i.e.* if a node $m$ in $G_1$ has a type that is a child type of a node $n$ in $G_2$, then these nodes could never be overlapped by the above algorithm whereas they should be in theory. We have therefore modified the algorithm to deal with such cases. The solution was to upgrade the type of $m$ in $G_1^j$ to the type of $n$ which allows the nodes to be matched. Then once $O_i$ is constructed, we downgrade the type of $m$ (in $O_i$) back to its original type yielding the correct overlap.

## 8.3 Complexity of Graph Overlapping

The graph overlapping algorithm has a very high complexity and typically generates a large number of overlaps. Without making an exact calculation, we can estimate the complexity by noting that enumerating subgraphs, which is needed in the overlapping algorithm, is similar to enumerating subsets. For a set of size $N$ the number of subsets is $2^N$. For a graph of $N$ nodes, the number of subgraphs is greater than $2^N$ due to the existence of edges which may or may not be included in a subgraph. We therefore estimate that the number of overlaps grows very quickly with the size of the overlapped graphs. As a result, the precondition produced by $wlp$ can typically be very large.

Another complexity factor is the recursive nature of $wlp$. For program sequencing and looping the result of a $wlp$ application is used as input for the next one: $wlp\left((P;Q),d\right) = wlp\left(P, wlp\left(Q,d\right)\right)$. Therefore the large number of overlaps produced by the first application is then overlapped with other graphs in the next application of $wlp$ yielding an even larger precondition. Consequently, as the construction progresses through the rules of the transformation one by one, the precondition grows very large and does so very quickly.

We propose in the next section several strategies which jointly reduce this growth by simplifying conditions as they are constructed and eliminating useless computations.

## 8.4 Simplification Strategies for Taming Combinatorial Explosion

### 8.4.1 NGC and Standard Logic Properties

The first and most straightforward strategy is determining statically the NGCs that always evaluate to $true$ or $false$, and simplify boolean formulas according to standard properties of boolean connectives. The simplification rules are the following:

1. If $a$ is an isomorphism, then $\exists(P \overset{a}{\hookrightarrow} C, c) \equiv c'$
   Where $c'$ is the translation of condition $c$ into the exact same condition over $P$ through the isomorphism $a^{-1}$.

   Intuitively, since $a$ is an isomorphism, then all elements of $C$ have origin elements in $P$ and there are no other non-mapped elements in $C$. Therefore $C$ does not add further constraints to the condition, and the nested condition

$c$ can directly be applied to $P$. This simplification is particularly interesting when $c = true$ since in that case $\exists(P \overset{a}{\hookrightarrow} C, true) = true$.

2. $\exists(a, false) \equiv false$
3. For any NGC $c$
   a. $\neg\neg c \equiv c$
   b. $c \wedge true \equiv c$
   c. $c \wedge false \equiv false$
   d. $c \vee false \equiv c$
   e. $c \vee true \equiv true$

The above simplification rules allow to greatly reduce the size of conditions by eliminating unnecessary nesting levels and short-circuiting branches of conditions.

### 8.4.2 Rule Selection

The second strategy consists in skipping rules that are irrelevant to the analysis to reduce unnecessary complication of the precondition. A postcondition expresses a constraint over nodes and edges of particular types. If a rule $\rho$ does not create nodes or edges of the types involved in the postcondition (or their children types), then it cannot affect the satisfaction or non-satisfaction of that postcondition. In that case it is useless to apply $wlp$ for that rule, and $\rho$ can be skipped in the analysis to avoid complicating the precondition unnecessarily. As a result instead of analysing $T_{\leq N}$, we propose to analyse an alternate transformation $T^{select}_{\leq N}$ where only rules that create elements of the types involved in the postcondition (or their children types) are kept. Formally, we speculate that:

$$wlp\left(T^{select}_{\leq N}, d\right) \Leftrightarrow wlp\left(T_{\leq N}, d\right)$$

This claim is only an intuition at this stage and we do not yet have a formal justification for this proposal. However our experiments showed that in the majority of the cases that we considered ($\sim 70\%$) we obtained $wlp\left(T^{select}_{\leq N}, d\right) = wlp\left(T_{\leq N}, d\right)$. In the minority of cases where the preconditions were different, manual analysis showed that the preconditions were equivalent despite the structural differences. This experimental evidence tends to support our intuition and we may be able to provide formal evidence in future work.

Note that the equivalence does not hold for $scopedWlp$ because it involves the scoping condition which is necessarily different depending on the rules selected in

$T^{select}_{\leq N}$. However our intuition is that in the same way that non-contributing rules can be skipped in $wlp$, such rules can also be eliminated from the scoping condition while maintaining the validity of the resulting precondition, *i.e.* the result is still a liberal precondition that guarantees the satisfaction of the postcondition.

To state our intuition formally, we believe that $scopedWlp\left(T^{select}_{\leq N}, d\right)$ is a liberal precondition that is *weaker than / less constraining than / implied by* $scopedWlp\left(T_{\leq N}, d\right)$, and yet remains a liberal precondition of the unbounded transformation $T$. Therefore, not only is $T^{select}_{\leq N}$ easier to analyse because it involves less rules, the result is also a less constraining precondition that still guarantees the satisfaction of the postcondition.

$$scopedWlp\left(T_{\leq N}, d\right) \;\Rightarrow\; scopedWlp\left(T^{select}_{\leq N}, d\right) \;\Rightarrow\; wlp\left(T, d\right)$$

**Example 8.1 (Rule selection for $wlp$ construction).** We consider the ATL transformation from Example 7.8 which is translated to the following bounded AGT transformation:

$$T_{\leq 1} \;=\; R1_{Inst}\!\downarrow_1;\; R2_{Inst}\!\downarrow_1;\; R1^{t1,refD}_{Res}\!\downarrow_1;\; R1^{t1,refE}_{Res}\!\downarrow_1;\; R2^{t,refD}_{Res}\!\downarrow_1$$

And we consider postcondition $Post_1$ from Example 7.9:

$$Post_1 = \exists\left(\;\boxed{d{:}D} \xrightarrow{refE} \boxed{e{:}E}\;\right)$$

The postcondition only contains nodes of type $D$ and $E$, and edges of type *refE*. We can therefore only consider the rules of the transformation that create elements of these types (or their children types) and analyse the following transformation instead.

$$T^{select}_{\leq 1} \;=\; R1_{Inst}\!\downarrow_1;\; R2_{Inst}\!\downarrow_1;\; R1^{t1,refE}_{Res}\!\downarrow_1$$

Comparing the results of the analysis, we find indeed that $wlp\left(T^{select}_{\leq 1}, Post_1\right) = wlp\left(T_{\leq 1}, Post_1\right)$

□

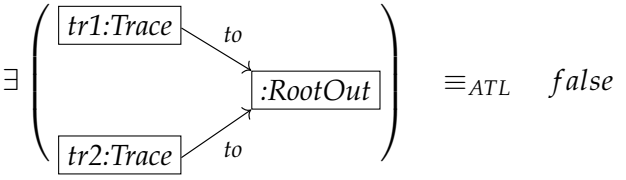
### 8.4.3 ATL Semantics

The third strategy we propose is specific to ATL. It consists in using of the ATL semantics to statically determine if certain NGCs are equivalent to $false$. Due to the properties of *Trace* nodes we know for a fact that certain graph patterns can never be produced by the AGT rules translated from ATL. If such patterns occur in

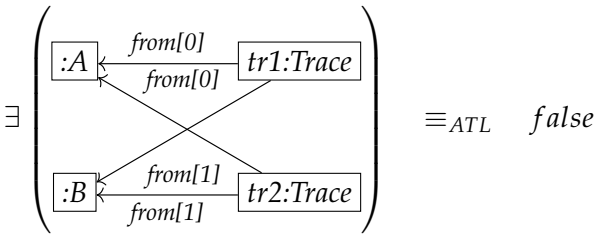
 

a NGC, we know that the NGC can never be satisfied and could be simply replaced with $false$.

A first property of the ATL semantics is that every target object is instantiated by one, and only one, ATL rule. Any target object can never have multiple incoming *to* edges from different trace nodes. Therefore we know statically that any condition $\exists(P \hookrightarrow C, c)$ where $C$ contains a target object with multiple incoming *to* edges as in the following diagram will never be satisfied by any graph along all possible executions of the transformation. Such a condition can therefore be replaced by $false$.




Another property is that there cannot be two trace nodes which have the same ordered set of objects in their *from* reference. This is ensured by the assumption that application conditions of instantiation rules which come from guards of ATL rules are mutually exclusive *i.e.* a set of objects never satisfies the guards of multiple ATL rules simultaneously. As a result, any condition $\exists(P \hookrightarrow C, c)$ where $C$ contains two (or more) trace nodes with the same ordered set of objects in their *from* references can never be satisfied by any graph along all possible executions of the transformation. Such a condition can therefore be replaced by $false$. This the case of the following condition for example:




**Definition 8.1 (ATL-based NGC filtering).** We formalize this strategy as a filtering predicate which returns False if its input morphism violates the ATL semantics.

$$isValid_{ATL}: \quad \mathcal{M} \longrightarrow \mathbb{B} = \{\text{True}, \text{False}\}$$

$$P \hookrightarrow C \longmapsto \begin{cases} \text{False} & \text{if } C \text{ violates the ATL semantics} \\ \text{True} & \text{otherwise} \end{cases}$$

When the filter returns True the condition is preserved. When it returns False, the condition is eliminated and replaced with $false$.

□

### 8.4.4 Element Creation

The last strategy we propose is presented in the context of ATL, but can be applied to arbitrary exogenous AGT transformations if they exhibit the necessary properties.

In ATL transformations each resolving rule creates an edge of a specific type. In certain transformations it may occur that a type *ref* of edges is only created by one (or a small number of) resolving rule(s) $R$. This means that before the execution or iteration of this rule, the transformed graph can never contain edges of the type *ref*. As a result, in the precondition $wlp\,(R{\downarrow_N}, d)$ any NGC $\exists(P \hookrightarrow C, c)$ where $C$ contains edges of type *ref* can never be satisfied and can therefore be replaced by $false$. A similar strategy could also be applied to instantiation rules based on the types of nodes they create, however it is less straightforward since each instantiation rule often creates more than one type of nodes.

**Definition 8.2 (Element creation NGC filtering).** For this strategy we define for a node or reference type $t$ the following filtering function that determines if a condition contains elements of type $t$ or not.

$$notContains_t : \quad \mathcal{M} \longrightarrow \mathbb{B} = \{\text{True}, \text{False}\}$$
$$P \hookrightarrow C \longmapsto \begin{cases} \text{False} & \text{if } C \text{ contains elements of type } t \\ \text{True} & \text{otherwise} \end{cases}$$

When the filter returns True the condition is preserved. When it returns False, the condition is eliminated and replaced with $false$.

□

The strategy could be generalised to arbitrary exogenous AGT transformations where a particular type of elements is only created by a well identified part of the transformation. The precondition of that part of the transformation can be simplified with the method above. The transformation must be exogenous because the strategy relies on the fact that elements of the target metamodel cannot exist in input models. We do not perform this generalisation here and remain within the specific context of ATL.

**Example 8.2 (Element creation NGC filtering).** As an example we consider the ATL transformation of Example 7.8 that was translated to the following AGT program for its analysis with $wlp$:

$$T_{\leq N} = R1_{Inst}\downarrow_N;\ R2_{Inst}\downarrow_N;\ R1_{Res}^{t1,refD}\downarrow_N;\ R1_{Res}^{t1,refE}\downarrow_N;\ R2_{Res}^{t,refD}\downarrow_N$$

Looking at the resolving rules, we notice that only $R1_{Res}^{t1,refE}$ creates references of type *refE*. Therefore after computing $wlp(R1_{Res}^{t1,refE}\downarrow, d)$ we can apply the filter $notContains_{refE}$ and replace all NGCs that contain references of type *refE* with $false$.

To illustrate this, we consider the postcondition $Post_1$ of Example 7.7 for which we had constructed:

$$wlp\left(R1_{Res}^{t1,refE}, Post_1\right) = \forall\left(LHS, \quad \ldots \quad \Rightarrow \left(\exists l_1' \vee \exists l_2' \vee \exists l_3' \vee \exists l_4'\right)\right)$$

Using $notContains_{refE}$ eliminates all conditions that contain $refE$, which leaves only $\exists l_1'$ in the condition and greatly simplifies it:

$$\begin{aligned} wlp\left(R1_{Res}^{t1,refE}, Post_1\right) &= \forall\left(LHS, \quad \ldots \quad \Rightarrow \left(\exists l_1' \vee false \vee false \vee false\right)\right) \\ &= \forall\left(LHS, \quad \ldots \quad \Rightarrow \exists l_1'\right) \end{aligned}$$

□

**Optional Reordering of Rules for ATL**

Based on the element creation strategy, it is possible to go further in the case of ATL. We notice that $wlp$ processes sequences in reverse *i.e.* $wlp\left((P;Q;R),\ d\right) = wlp\left(P, wlp\left(Q, wlp\left(R, d\right)\right)\right)$ starts with processing the last program of the sequence $R$ and moves towards the first one. It is therefore desirable to apply the simplification strategy as early as possible, from the first rule $R$ if possible.

Fortunately, because ATL rules are independent, their corresponding AGT rules can be reordered without affecting the semantics of the transformation. Thus for the previous example, it is possible to move $R1_{Res}^{t1,refE}$ to the end of the sequence so that the simplification strategy can be performed early to avoid unnecessary computations:

$$T_{\leq N} = R1_{Inst}\downarrow_N;\ R2_{Inst}\downarrow_N;\ R1_{Res}^{t1,refD}\downarrow_N;\ R2_{Res}^{t,refD}\downarrow_N;\ R1_{Res}^{t1,refE}\downarrow_N$$

In the above example the reordering is not particularly interesting because each type of reference is created by only one rule[1]. However in transformations where a type of reference is created by more than one rule, it is possible to group and reorder resolving rules according to the type of references that they create. For a general ATL transformation, we propose to create for each reference *ref* in the target metamodel, a program $Cr_{ref}$ which is the sequencing and iteration of rules that create references of type *ref*.

$$Cr_{ref} = R^1_{Res}{\downarrow_N}; R^2_{Res}{\downarrow_N}; \ldots \qquad \text{where } R^k_{Res} \text{ creates references of type } ref$$

Then we sequence the programs $Cr_{ref}$ by descending number of rules involved in each program. We thus reorganise the transformation into the following form:

$$T = Instantiation \ ; \ Cr_{ref_1} \ ; \ Cr_{ref_2} \ \ldots$$

where

$$i < j \quad \Rightarrow \quad |Cr_{ref_i}| \geq |Cr_{ref_j}|$$

$|P|$ being the number of rules in the program $P$.

By organising the transformation in this manner, we can eliminate all NGCs that contain edges of type $ref$ after the completion of each $wlp(Cr_{ref}, d)$ computation. Since the programs $Cr_{ref}$ with a smaller number of rules are sequenced last in the transformation, they are processed first by $wlp$. Therefore the simplification can be applied as early as possible and we can limit the growth of the precondition early in the computation.

A similar grouping and reordering can be done for nodes creation in instantiation rules. However since each instantiation rule can instantiate multiple target metaclasses, we cannot always organise them into programs $Cr_{class}$ because each instantiation may be placed in more than one $Cr_{class}$ programs. Nonetheless, for metaclasses only instantiated by one rule, filtering can be performed. This is notably applicable to typed trace nodes: each instantiation rule $R_{Inst}$ is the only one to instantiate nodes of type *R_Trace*. Therefore after processing $wlp\,(R_{Inst}{\downarrow_N}, d)$ we can apply the filter $notContains_{R\_Trace}$.

This concludes the presentation of the simplification strategies that we propose in order to eliminate irrelevant parts of the conditions resulting from the $wlp$ construction. In the next section we propose to integrate some of these strategies into the $wlp$ construction to apply simplifications early on and avoid unnecessary computations.

[1] the same name *refD* was used for *D::refD* and *E::refD*, but they are different types

## 8.5 Combining $A$ and $L$ for Early Simplification – $Post2Left$

$wlp$ relies internally on two constructions $A$ (Definition 7.13) and $L$ (Definition 7.14). We propose to combine the two constructions into a single construction called $Post2Left$. This combined construction applies the filtering functions defined in the previous strategies as soon as overlaps are computed which allows to eliminate irrelevant overlaps early on and avoid useless computations. While all existing theoretical approaches always define $A$ and $L$ separately for the sake of formal proofs, this is to our knowledge the first proposal combining both constructions in the interest of scalability concerns.

**Definition 8.3 ($Post2Left$ combined construction).** Given an arbitrary morphisms $r : L \hookrightarrow R$ and $b : P \hookrightarrow R$, and an NGC $e$ over $P$, the combined construction $Post2Left(r, b, e)$ is defined such that it is equivalent to the combined application of $A$ and $L$.

Given a rule $\rho = \left\langle LHS \overset{p}{\hookrightarrow} RHS, ac_L \right\rangle$, the morphism $i : \phi \hookrightarrow RHS$ and a postcondition $d$, the combined construction, we have:

$$L(\rho, A(\rho, d)) = Post2Left\,(p, i, d)$$

$$\begin{array}{ccccc} LHS & \xhookrightarrow{\quad p \quad} & RHS & \xhookleftarrow{\quad i \quad} & \phi \\ \blacktriangle & & & & \blacktriangle \\ Post2Left\,(p,i,d) & & & & d \end{array}$$

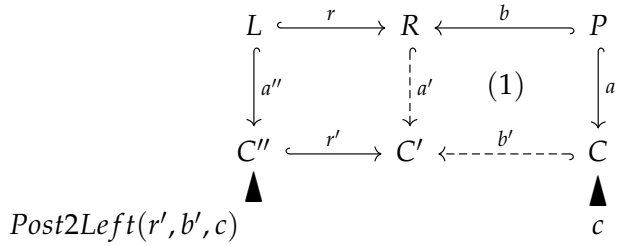
As a result, the precondition for $\rho$ can be constructed using $Post2Left$ instead of $A$ and $L$:

$$wlp\,(\rho, d) = C_\forall\,(\rho,\ ac_L \Rightarrow Post2Left\,(p, i, d))$$

**Construction.** The construction $Post2Left(r, b, e)$ is defined as follows, where $isValid_{ATL}$ and $notContains_t$ are defined respectively in definitions 8.1 and 8.2.

- $Post2Left(r, b, true) = true$

- $Post2Left(r, b, \neg\, c) = \neg\, Post2Left(b, c)$
  $Post2Left(r, b, \bigwedge_i c_i) = \bigwedge_i\ Post2Left(r, b, c_i)$
  $Post2Left(r, b, \bigvee_i c_i) = \bigvee_i Post2Left(r, b, c_i)$




– $Post2Left(r, b, \exists(a, c)) = \bigvee_{(a',b',a'',r') \in \mathcal{F}} \exists(a'', Post2Left(r', b', c))$
where $\mathcal{F} = \{(a', b', a'', r') \mid$

$(a', b')$ jointly surjective,
$a'$ and $b'$ injective,
(1) commutes,
$isValid_{ATL}(a') =$ True,
the pushout complement $(a'', r')$ of $(r, a')$ exists,
$notContains_t(a'') =$ True if a filter $notContains_t$ is applicable $\}$

and $false$ if $\mathcal{F} = \emptyset$

In this construction, $(a', b')$ pairs are constructed just like with *Shift*. However when an overlap yields $a'$ such that the pushout complement does not exist or $isValid_{ATL}(a') =$ False or $notContains_t(a'') =$ False (when applicable), then we do not process the nested condition $c$ for that overlap. In comparison, the conventional *Shift* construction $c$ was always systematically applied to $c$. Our construction allows to make early simplifications and avoid making useless overlapping computations over $c$. Keeping in mind that the nested condition $c$ can be a very large condition resulting from a previous $wlp$ computation, avoiding the processing of $c$ for irrelevant overlaps is a major improvement.

□

**Discussion**

It is important to note that this alternate construction $Post2Left$ is useful beyond ATL transformations. If the analysed transformation is not an ATL transformation, the filters $isValid_{ATL}$ and $notContains_t$ can simply be replaced with vacuous filters that always return True. Or alternatively, the filters can be replaced with other filters implementing other simplification strategies. This could be useful if the analysed transformation has other semantical properties that allow filtering and eliminating conditions like we have shown for ATL. In this manner semantical knowledge about the analysed transformation can be introduced into the $wlp$ construction to avoid unnecessary computation and help with scalability. In that spirit, our

implementation of $Post2Left$ applies filtering in a modular fashion. ATL-specific filters are implemented as modular filters that can easily be removed, composed or replaced with other filters.

Additionally, it is interesting to recall at this point the comparison that we discussed in Section 7.7 between our NGC-based construction of preconditions, and the OCL-based construction proposed in [Cabot *et al.*, 2010a]. The OCL-based construction is based on a set of textual replacement patterns performed on the postcondition to transform it into a precondition. This approach had the advantage of supporting more expressive postconditions because it relied directly on OCL (which is more expressive than AGT). However, because its replacement patterns also need to consider all possible ways in which a rule may affect the postcondition, we find this to be similar to the graph overlapping operation in the NGC-based construction. Therefore we suspect that the OCL-based approach suffers also from the problem of combinatorial explosion that we faced in the NGC-based approach.

As a result, an advantage of our NGC-based approach is the possibility of easily introducing simplifications construction as was demonstrated in this section, something that is not straightforward with the OCL-based approach. Thanks to the graph nature of NGCs, we were able to easily characterise graphs that violate the ATL semantics and eliminate them. This semantic knowledge of ATL would be harder to introduce in the textual replacement patterns of the OCL-based construction because OCL does not provide the same level of abstraction as graphs.

Having proposed $Post2Left$ for the early filtering of conditions as a way to avoid unnecessary computations, we now turn to parallelisation and memory management aspects of the implementation.

## 8.6 Parallelisation and Memory Management

**Parallelising** $wlp$

Given the high complexity of the $wlp$ construction and the rapid growth in the size of the computed conditions, it is necessary to parallelise its implementation in order to obtain results in a reasonable time. The bulk of the computation resides in $Post2Left(r, b, \exists(a, c))$ where the graph overlapping occurs while the handling of the other forms of conditions is straightforward.

In $Post2Left(r, b, \exists(a, c))$, for each overlap result $(a', b', a'', r') \in \mathcal{F}$ we need to perform a new computation $Post2Left(r', b', c)$ over the nested condition $c$. Assuming $c$ is composed of negations, conjunctions and disjunctions of a set of $M$ conditions, then for each overlap pair we need to perform $M$ computations $Post2Left(r', b', \exists(a_i, d_i)$

which are independent of each other. We therefore propose to run them as parallel jobs.

Since each parallel job will also result in new jobs for its nested conditions, then we can expect that a large number of jobs will be created. For this reason, we do not start parallel jobs as soon as they are created to avoid overloading computing resources. Instead we have opted for an architecture with a job queue serving a thread pool of fixed size. New jobs are placed in the job queue, and dequeued as worker threads of the pool become available. This allows to have a fixed number of parallel jobs running at any point in time.

#### Managing Memory Usage

As *wlp* processes rules of the transformation, preconditions and graphs become very large and quickly exhaust the available memory. We have therefore propose to use a mechanism that dumps intermediate results of the computation to files on the disk and reload them into memory as they become needed. This allows for the computation to keep going without ever exhausting the memory (assuming unlimited disk space).

To allow for partial results to be dumped on disk, it was necessary to fragment conditions into separate resources. Even though existing tools propose generic model fragmentation solutions (*i.e.* Neo4EMF[2] and EMF-Fragments[3]), it was not possible to apply them to Henshin conditions due to the existence of manually written Henshin components preventing the generic fragmentation mechanisms. As a result, we implemented a custom fragmentation and dumping mechanism for Henshin conditions built into the *wlp* implementation. While far from being ideal, this solution allowed to reduce memory usage considerably. In the future better solutions from the research domain of model scalability should be investigated.

## 8.7 Conclusion

In this chapter we have focused on the scalability of the *wlp* construction. Given the high complexity of the graph overlapping operation at the heart of the construction, we have proposed several simplification strategies allowing to avoid unnecessary computations and reduce the size of the resulting conditions. First we proposed 4 simplification strategies:

(1) NGC and Standard Logic Properties

[2]Neo4EMF, `http://www.neo4emf.com`
[3]EMF-Fragments, `https://github.com/markus1978/emf-fragments`

(2) Rule Selection

(3) ATL Semantics

(4) Element Creation

Strategies (1) and (2) are general to arbitrary AGT transformations. Strategy (3) is specific to ATL, and strategy (4) was presented in the specific context of ATL but can be generalised to arbitrary exogenous AGT transformations if the necessary properties hold.

Then we proposed a new modified construction $Post2Left$ for $wlp$ that allows to apply the above strategies early and avoid performing unnecessary computations. Finally we proposed parallelisation and memory management strategies to provide reasonable performance despite the high computational complexity.

The experimental assessment of all these proposals will be presented in Chapter 10, where we will demonstrate that our simplification strategies are highly efficient. Meanwhile, we move onto the third and final contribution of this thesis in Chapter 9: a syntactic specification and test oracles approach for model-to-code transformations.

# Chapter 9

# Template-based Specification of Model-to-Text Transformations

## Contents

## 9.1 Introduction

The previous chapters focused on the first problem tackled by this thesis which was the backwards translation of test requirements in the final purpose of producing integration tests of a model transformation chain. We now move to the second problem which is determining the verdict of these integration tests in the case of a code generation chain with suitable test oracles. We determined that for the needs of qualification, the test oracles need to be based on a specification of the ACG focusing on the *syntax* of the generated code.

Consequently we proposed an approach for the specification and test oracles of model-to-test transformations. At the heart of this approach is the notion of *specification templates* which allow specifying the generated source code in terms of its textual concrete syntax. These templates are composed of a combination of verbatim text, queries to the input model, regular expressions and repetition statements. The execution of this specification over a test model provides expected patterns which should be matched in the test output, thus consisting an automatic test oracle. Finally to address qualification needs, we proposed automatic document generation as a way to produce a document consistent and easily integrated with the rest of the qualification evidence.

We explained previously that our approach is tailored for the qualification of QGen, the Simulink to C code generator developed at AdaCore, and in this chapter we detail our proposals in this context. In Section 9.2, we give an overview on the code generation strategy of QGen to understand what we aim to specify. Then in Section 9.3 we introduce the notion of *specification templates* which are the foundation of our approach.

Defining code generation for Simulink requires a complex specification which is why we propose to factorise common definitions and specification elements such as complex regular expressions into libraries of reusable queries. As a result we present this organisation of the specification and its libraries in Section 9.4.

Once the supporting libraries are laid out, we present in Section 9.5 how Tool Operational Requirements (TORs) are formalised as specification templates, and demonstrate in Section 9.6 how they are used as test oracles.

Finally we discuss document generation in Section 9.7, and recall related work in Section 9.8. In particular, we compare our approach with the model-to-text specification approach of [Wimmer and Burgueño, 2013].

## 9.2 Use Case: Simulink to C Code Generation in QGen

### 9.2.1 Semantics of Simulink Models

A Simulink model consists of *computation blocks* each having *inports* and *outports* which are connected via *signals*. Each block reads data from its inport(s), performs a computation, and makes the result available on its outport(s) for successor blocks. Each block has a *type* which defines the computation that is performed. For example a block of type **Sum** computes the sum of its inputs while a block of type **Gain** multiplies its input by a constant. A block may also have *parameters* which are either numeric values used in the computation or enumeration values that configure the computation. For example we will explain shortly that a **UnitDelay** block *delays* its input signal by one iteration, meaning that for the initial iteration an arbitrary value must be used and is specified via a block parameter named *InitialCondition*.

Inports and outports are typed with a *data type*. A data type is defined by 2 aspects:

- The *base type* which can be one of: `double`, `single`, `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`.
- The *dimensions* which are either non-existent for scalar data types, an integer `n` for a vector data type of size `n` or a pair of integers `[m, n]` for a matrix of `m` rows and `n` columns.

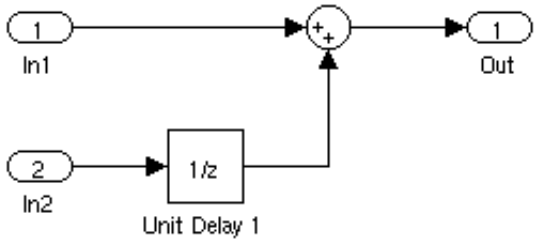



Figure 9.1: Example of a Simulink model

A Simulink model has a graphical representation depicted in Figure 9.1. Blocks are represented as boxes, often using different shapes for different block types. In this example, *In1* and *In2* are blocks of type **Inport** and *Out* is a block of type **Outport** because they represent respectively the inports and the outport of the containing system. *Unit Delay 1* is a block of type **UnitDelay** and the circle-shaped block is a block of type **Sum** with a (non-visible) name *Sum 1*.

Signals are represented as lines connecting blocks. Inports and outports are not explicitly depicted but correspond to the start point and end point of signal

arrows. Block parameters are not always depicted graphically and are usually inspected and modified via a separate window in the user interface. For example, the **UnitDelay** block has the *InitialCondition* parameter set to 0 but this is not depicted graphically.

Port types are also not depicted graphically and can be set via a separate window in the user interface. In the example of Figure 9.1 the port types are as follows:

| **Port** | **Base Type** | **Dimensionality** |
|---|---|---|
| Outport of *In1* | int16 | scalar |
| Outport of *In2* | int16 | vector of size 2 |
| Inport of *Unit Delay 1* | int16 | vector of size 2 |
| Inport 1 of *Sum 1* | int16 | scalar |
| Inport 2 of *Sum 1* | int16 | vector of size 2 |
| Outport of *Sum 1* | int16 | vector of size 2 |
| Inport of *Out* | int16 | vector of size 2 |

The set of blocks and signals specifies a computation algorithm. Simulink models have a synchronous data flow execution semantics consisting of an iterative periodic execution of the computation algorithm. At each iteration, data values are read from the inports of the system and routed along signals to computation blocks. Each computation block reads its input data, performs the computation defined by its type (*i.e.* sum, multiplication *etc.*) and outputs the result on its outport for successor blocks. Some computation blocks store internal persistent state which is preserved between iterations of the algorithm. For example, a block of type **UnitDelay** produces as output the value of its input at the previous iteration step. Therefore at each iteration, the input value of a **UnitDelay** block is stored in the persistent state to be used at the next iteration. Note that blocks may have complex semantics that depends on the dimensions of the data they handle. For example in Figure 9.1, *Sum 1* performs the sum of a scalar with a vector by adding the scalar to each component of the vector.

### 9.2.2 Implementation of Simulink Semantics in Source Code

The semantics of Simulink models can be implemented in several ways into C source code. The implementation strategy chosen in QGen is such that for an input Simulink model, two files are generated: a C header file providing an interface to invoke the generated code, and a C implementation file containing the implementation. The implementation is composed of the following elements:

1. A set of module-level *persistent variables* to store persistent data if any.
2. An `init()` function which sets the initial values of the persistent variables.

3. A `compute(args...)` function which performs one cycle of computation of the model. The inports and outports of the system correspond to input and output arguments of the `compute` function.

   a. For each outport of a computation block, a *local variable* is created.

   b. A first chunk of code statements named the *"compute section"* contains, for each computation block, the code statements that implement it.

   c. The sequencing of code statements should honor the data flow defined by the signals in the model: the code statements of a block should always appear before the code statements of all its successors.

   d. A second chunk of code statements named the *"update section"* contains statements that store persistent data in the module-level persistent variables.

Based on the above implementation scheme, the code that should be generated in the C implementation file for the Simulink model in Figure 9.1 is shown in Listing 9.1. The header file is trivial and only contains the declarations of the functions in the listing.

Listing 9.1: C Implementation of a Simulink model

```
1  /* my_system.c */
2
3  GAINT16 Unit_Delay_1_memory[2];
4
5  void init() {
6    Unit_Delay_1_memory[0] = 0;
7    Unit_Delay_1_memory[1] = 0;
8  }
9
10 void compute(GAINT16 In1, GAINT16 const In2[2], GAINT16 Out[2]) {
11   GAINT16 Unit_Delay_1_out[2];
12   GAINT16 Sum_1_out[2];
13   GAUINT8 i;
14
15   /* Compute Section */
16   for (i = 0; i <= 1; i++) {
17      Unit_Delay_1_out[i] = Unit_Delay_1_memory[i];
18   }
19
20   for (i = 0; i <= 1; i++) {
21     Sum_1_out[i] = In1 + Unit_Delay_1_out[i];
22   }
23
24   for (i = 0; i <= 1; i++) {
25     Out[i] = Sum_1_out[i];
```

```
26    }
27    /* End Compute Section */
28
29    /* Update Section */
30    for (i = 0; i <= 1; i++) {
31       Unit_Delay_1_memory[i] = In2[i];
32    }
33    /* End Update Section */
34 }
```

The above code is ultimately destined to be embedded in a real-time architecture which calls `init` once at the initialisation of the system and then calls `compute` periodically, thus implementing the synchronous data flow semantics of Simulink.

### 9.2.3 General Structure of the Generated Code

In our work we will focus on the implementation file which is the most interesting part. As was explained the implementation file has a fixed structure with well identified code *sections*. That structure is shown in the following listing where the code sections are indicated between brackets.

Listing 9.2: General structure of C code generated by QGen

```
1  [Persistent Variables]
2
3  void init() {
4    [Init Statements]
5  }
6
7  void compute(...) {
8    [Local Variables]
9
10   /* Compute Section */
11   [Compute Statements]
12   /* End Compute Section */
13
14   /* Update Section */
15   [Update Statements]
16   /* End Update Section */
17 }
```

In this fixed structure, the content of each section depends on the model elements in the input model. For each computation element in the input model the ACG generates code elements in one or more of the sections to implement the semantics of the model element. Section contents are as follows:

**Persistent Variables**

If a block retains internal persistent state, code variables are generated in this section to store this state. These persistent variables store data between invocations of the `compute` function. If a block does not retain persistent state, it does not contribute any persistent variables.

**Init Statements**

If a block retains internal persistent state, it contributes statements in this section to initialize its persistent variables.

**Local Variables**

Generally each outport of a block results in a variable declaration in this section. A block may also contribute additional variables necessary for its computation, for example to store parameter values or an intermediate result.

**Compute Statements**

Each block contributes code statements implementing its semantics in this section.

**Update Statements**

When a block retains persistent state, statements are generated in this section to update the content of persistent code variables in preparation of the next `compute` invocation.

In the remainder of this chapter we will focus on specifying the content of each of the above sections. Namely, we will specify for each Simulink block type, the code statements that it contributes in each of the above 5 sections. Not all aspects of code generation will be covered by this specification. For example we will not specify how the sequencing of the generated code statements is determined. We assume that our specification should be complemented with other specifications defining the other aspects of code generation. From the perspective of tool qualification, our specification represents only part of the Tool Operational Requirements (TORs).

In the next section we introduce the notion of specification templates which are the heart of our specification approach.

## 9.3 Specification Templates and Queries

We propose the concept of *specification templates* as a basis for the specification of model-to-text transformations, and we propose to use Acceleo as an implementation technology of specification templates. A specification template has the following general form:

Listing 9.3: General structure of a specification template in Acceleo

```
1 [template templateName (input0 : type0, input1 : type1 ...) ? (oclGuard)]
2 verbatim text, interleaved with [oclQueries/] between brackets,
3 %<regularExpressions>% between percent delimiters
4 and loop statements expressing repeating patterns:
5 [for ( iterator | collection )]
6   This text repeats for all elements of the collection.
7   We can use [oclQueriesInvolvingIterator] here.
8 [/for]
9 [/template]
```

A specification template is composed of the following elements:

1. Input elements: objects typed by metaclasses of the input metamodel of the transformation. Most often templates will have one input, but in certain uses they will have multiple inputs.
2. A guard: an OCL constraint over the input elements that defines when the template is applicable
3. A pattern of text that should be generated when the guard is satisfied: the pattern of text is an arbitrary concatenation of the following four kinds of content:
   a. verbatim text
   b. OCL queries to the input model (enclosed in brackets `[` and `/]`)
   c. regular expressions (enclosed in `%<` and `>%`)
   d. repetition statements (`[for ...]` statements)

Conceptually, a specification template expresses a simple constraint of the following form:

$$\textit{application condition} \quad \Rightarrow \quad \exists\,(\textit{textual pattern})$$

where the application condition is defined by the input elements of the specification template and its guard, while the textual pattern is described by the content of the specification template.

Thus, a specification template applies when its input elements satisfy the guard. In that case, the pattern of text describes the content that should be generated in the output of the model-to-text transformation. Verbatim text should exist as is, literally. OCL queries should be evaluated, and their results should exist in the transformation output. For regular expressions, the corresponding generated text

can by any text that matches the regular expression. As for repetition statements `[for ...]`, they express patterns of text that should appear multiple times in the output. As customary with loop statements of programing languages, repetition statements define an iterator that ranges over a collection created or retrieved from the input model using an OCL expression. The iterator can then be referenced in OCL queries of the body of the repetition statement.

To avoid specification templates from becoming too complex, we will need to encapsulate specification elements such as common regular expressions and code patterns into reusable functions. Some of these elements will be encapsulated in templates, and others will be encapsulated in Acceleo *queries* which are operations taking objects as input and returning a result of a specific type. Essentially queries are similar to operations declared for metaclasses in the input metamodel, and they can be invoked in OCL expressions in the same way. Queries have the following structure:

Listing 9.4: General structure of Acceleo queries

```
1 [query queryName ( arg0 : type0, arg1 : type1 ...) : returnType =
2   oclExpression
3 /]
```

$type_0$ is the *context* of the query. Therefore in any OCL expression with an object `obj` of type $type_0$, the query is invoked like an operation over the type: `obj.`$queryName$`(...)`

We have thus defined the basic elements of our specification approach. Next, we present the general organisation of the specification based on these basic elements.

## 9.4 Organisation of the Specification

The specification is based on an input metamodel representing Simulink models. However besides the mere representation of Simulink models, there are many semantic properties and terminologies of Simulink that will be referenced in many places of the specification. For example, a matrix data type where the first dimension is 1 has only one row and is therefore referred to as a *row matrix* in Simulink. It is useful to define an operation `type.IsRowMatrix()` to formalise this terminology and use it in the specification. This terminology can be encoded in the metamodel, however we propose to define it as part of the specification in the form of Acceleo queries. This allows to develop the terminology as the requirements are being developed instead of having to update the metamodel which would be more tedious.

Additionally, it makes it easier to integrate this terminology in the TORs document in the document generation step of our overall approach.

Similarly, certain common definitions and regular expressions will be used in many places in the specification and will be encapsulated in reusable queries and templates. Besides factorisation, this avoids exposing complex regular expressions in the TORs and keeps them readable.

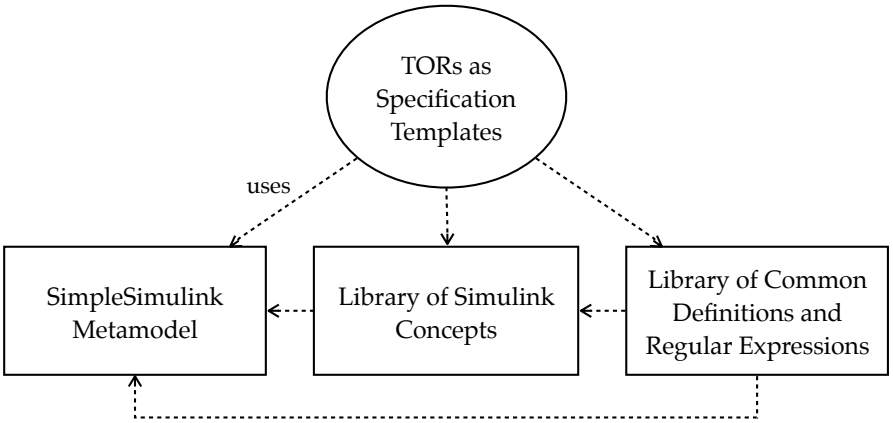



Figure 9.2: Organisation of the specification

As a result, the specification will be organised into the 4 components depicted in Figure 9.2:

***SimpleSimulink* Metamodel**
: A simple metamodel describing the structure of input Simulink models.

**Library of Simulink Concepts**
: A library of reusable queries encapsulating common Simulink concepts and terminology defined over the input metamodel.

**Library of Common Definitions and Regular Expressions**
: A library of reusable queries and templates encapsulating common definitions and regular expressions, defined using the input metamodel and the library of Simulink-specific concepts.

**Tool Operational Requirements (TORs)**
: The TORs formalised as Acceleo specification templates, based on the above components.

In the next sections we detail each of the 4 component of the specification.

### 9.4.1 *SimpleSimulink* Metamodel

The first step in our approach is defining a metamodel to represent the Simulink input language. We choose to only include in the metamodel the basic generic concepts of Simulink and not specific block types and their parameters. This is to obtain a simple core metamodel that does not change as block types are added to to the specification. Block types and their specific information is defined later as part of the specification in Acceleo without having to modify the metamodel. To avoid clutter, *SimpleSimulink* is presented in two separate diagrams: Figure 9.3a shows all metaclasses and all inheritance links and Figure 9.3b shows references between metaclasses, omitting inheritance links and non-essential metaclasses.

Here are the main characteristics of *SimpleSimulink*.

- Despite the usual convention of *camelCase* for attribute and reference names in EMF, *SimpleSimulink* follows the naming conventions of Simulink where relevent. For example, we use the attribute *Dimensions* in the metaclass *Type* instead of the conventional *dimensions* because the former spelling is the one used in Simulink.

- A *System* is composed of *Block*s. *Block*s have *Inport*s representing input data and *Outport*s representing output data. *Signal*s connect *Outport*s to *Inport*s of *Block*s in the *System*s.

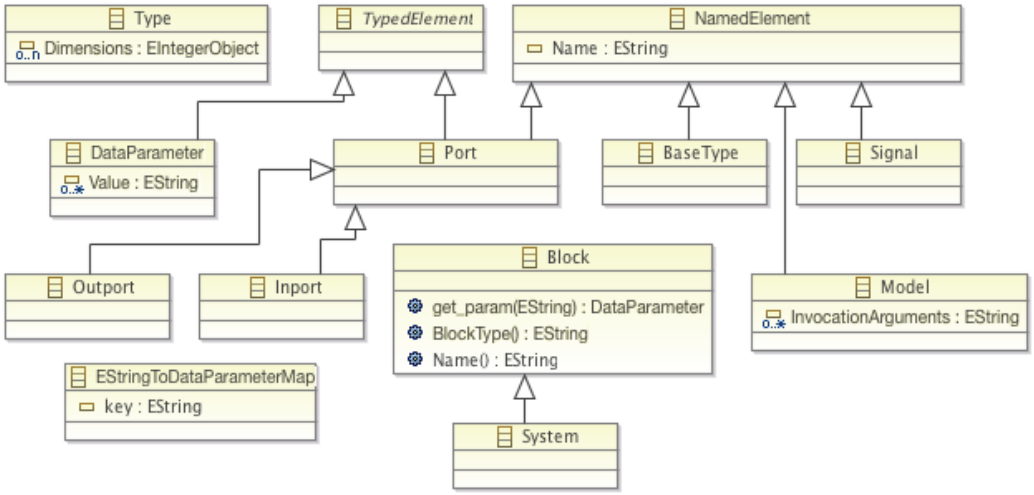
- Only a generic *Block* metaclass is defined, but not specific block types (*e.g.* **Sum**, **Gain** *etc.*). Parameters of a *Block* are stored in a *EStringToDataParameterMap* where keys of the map are names of block parameters (*e.g. InitialCondition*, *Gain etc.*) and values are instances of *DataParameter*. The EOperation *get_param(EString)*[1] retrieves a parameter from the map given its name.

- A *Block*'s type and its name are stored alongside all parameters in the parameter map. EOperations *BlockType()* and *Name()* are provided as shortcuts to retrieve the corresponding values. Roughly,

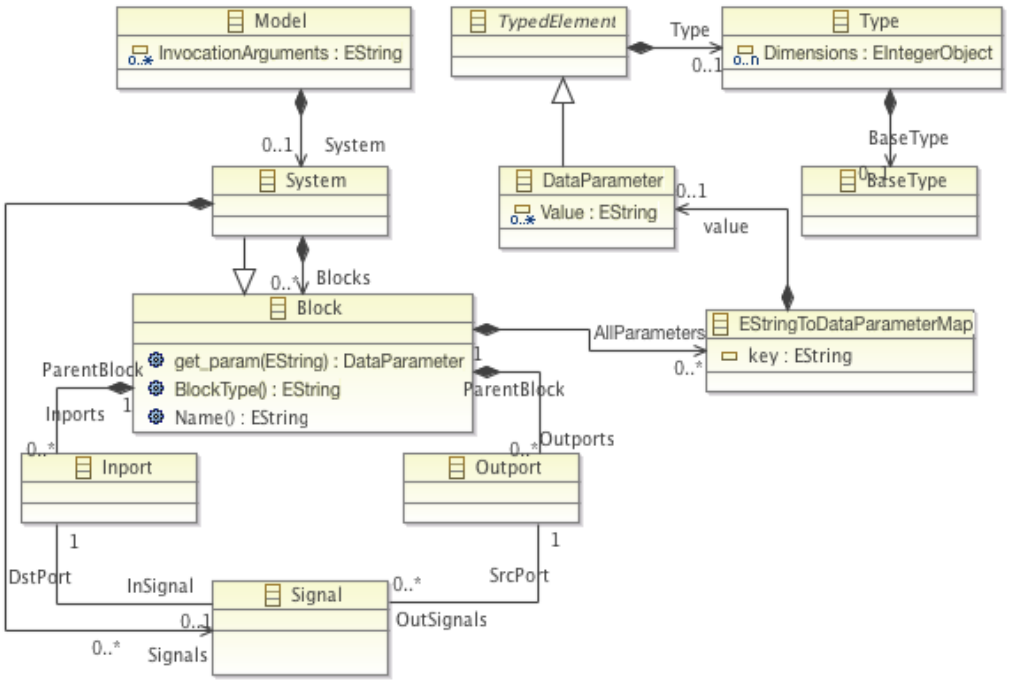
```
BlockType()= get_param('BlockType').Value->at(1)

Name()= get_param('Name').Value->at(1)
```

- Regardless of dimensionality (*i.e.* scalar, vector, matrix), numeric parameter values are represented like in Simulink with a unidimensional array in the EAttribute *Value*. It is the *Type* element with its *Dimensions* EAttribute that determines the dimensionality of the stored value and how the unidimen-

[1] it is named *get_param* after a function in the standard Matlab API with similar functionality




(a) Inheritance links




(b) Reference links (omitting unnecessary metaclasses and inheritance links)

Figure 9.3: *SimpleSimulink* metamodel

sional array should be indexed. The indexing of parameter values will be defined in the library of Simulink concepts. For example:

| Simulink Parameter Value | EMF DataParameter | |
|---|---|---|
| | *Value* | *Type.Dimensions* |
| `0` | `Sequence{'0'}` | `Sequence{}` |
| `0.0` | `Sequence{'0.0'}` | `Sequence{}` |
| `[1 2 3]` | `Sequence{'1', '2', '3'}` | `Sequence{3}` |
| `[1 2 3;`<br>`4 5 6]` | `Sequence{'1', '4', '2',`<br>`'5', '3', '6'}` | `Sequence{2, 3}` |

Having defined the input metamodel, we now define the second component of the specification: the library of common Simulink concepts.

### 9.4.2 Library of Simulink Concepts

The Simulink concepts defined in this library concern type-related notions of dimensionality, equivalence of type dimensions, base types *etc.* The main purpose of this library is defining functions such as `IsMatrix()` and `IsScalar()` allowing to write OCL expressions such as the following to express properties of block ports and parameter values:

```
block.Outport().IsMatrix()and block.InitialCondition().IsScalar()
```

The library is defined as a module of Acceleo queries. We define these concepts in Acceleo rather than in the metamodel to allow developing the terminology as the requirements are being developed instead of having to update the metamodel continuously which is more tedious. Additionally, this makes it easier to integrate these definitions in the Tool Operational Requirements (TORs) document in document generation.

There are many concepts defined in this library, therefore we only give examples of two kinds of such definitions in the following.

#### Type Dimensionality

The following queries define a terminology related to type dimensions.

```
1  [** A type is scalar when no dimensions are specified. /]
2  [query public IsScalar(t : Type) : Boolean =
3  t.Dimensions->size() = 0 /]
4
5  [** A type is a vector when it has exactly one dimension. /]
6  [query public IsVector(t : Type) : Boolean =
7  t.Dimensions->size() = 1 /]
8
9  [** A type is a matrix when it has exactly two dimensions. /]
10 [query public IsMatrix(t : Type) : Boolean =
```

```
t.Dimensions->size() = 2 /]

[** The number of rows of a non-scalar type is the first dimension. /]
[query public NRows(t : Type) : Integer =
if t.Dimensions->size() >= 1 then t.Dimensions->at(1)
else 0
endif /]

[** The number of columns of a non-scalar type is the second dimension.
/]
[query public NColumns(t : Type) : Integer =
if t.Dimensions->size() >= 2 then t.Dimensions->at(2)
else 0
endif /]
```

#### Indexing Numeric *DataParameter*s

As mentioned earlier, parameter values are stored as a unidimensional array of strings and it is the associated *Type* object that determines how to index the value array. The following queries define that indexing so that the values of a vector *DataParameter* `param` could be indexed in a manner similar to OCL collections as `param->at(i)`, and the values of a matrix *DataParameter* could be indexed as `param ->at(i, j)`.

```
[** Get value of parameter at index i, assuming parameter is a vector. /]
[query public at(c : Set(DataParameter), i : Integer) : String =
let param : DataParameter = c->any(true) in
param.Value->at(i) /]

[** Get value of parameter at index (row, col), assuming parameter is a
matrix. /]
[query public at(c : Set(DataParameter), row : Integer, col : Integer) :
    String =
let param : DataParameter = c->any(true) in
param.Value->at((col - 1) * param.NRows() + row) /]
```

Having defined the library of common Simulink concepts, we now move to the third component of the specification: the library of regular expressions.

### 9.4.3 Library of Common Definition and Regular Expressions

#### Common Definitions

Common definitions consist of naming conventions of variables and types in the generate code defined into reusable queries invoked in the specification. The

following listing contains a few examples of common definitions used often in the specification.

Listing 9.5: common/Definitions.mtl

```
1  [** RemoveSpaces() replaces all whitespace in a String with the '_'
2      character /]
3  [query public RemoveSpaces(s : String) : String =
4  s.replaceAll('[ \t\n]+', '_') /]
5
6  [** Name of the output variable of all blocks /]
7  [query public OutVar(block : Block) : String =
8  block.Name().RemoveSpaces() + '_out' /]
9
10 [** The name of the input variable 'i' is the name of the output
11     variable of the preceding block connected to inport 'i' /]
12 [query public InVar(block : Block, i : Integer) : String =
13 block.Inports(i).InSignal.SrcPort.ParentBlock.OutVar() /]
14
15 [** As a shortcut, block.InVar() refers to block.InVar(1) /]
16 [query public InVar(block : Block) : String =
17 block.InVar(1) /]
18
19 [** The translation of a \simulink/ basetype to a C code type /]
20 [template public TypeInCode(baseType : BaseType)]
21 ...
22 [/template]
```

### Regular Expressions

In our approach, we propose to use regular expressions in specification templates in order to abstract parts of the specification deemed out of scope or too complex to detail. This is useful when we want to specify some aspects of the generated code structure, but leave out other aspects that are too complex to detail and are specified by other parts of the specification.

For example, if we wish to specify that an assignment statement of variable `B` into variable `A` should be generated, a type cast may occur in that statement if `A` and `B` have different types. However we do not want to detail the conditions under which the cast appears. Instead, we only want to express the fact that the cast may optionally appear. This can be specified with the following specification template:

```
A = %<(B|\([a-zA-Z0-9_]+\) B)>%
```

The regular expression surrounded with the symbol "`%`" expresses the fact that the specification allows a cast to appear in the assignment expression but does not

detail precisely under what conditions the cast appears. The specification is thus kept simple and we have expressed the potential existence of a cast in an abstract way. Since such regular expressions will be used a lot across the specification, we propose to encapsulate them in reusable queries or Acceleo templates in the following way (the definition of `Optional_Cast` will be given shortly):

```
A = [Optional_Cast('B')/]
```

In the following, we give an example of a few regular expressions commonly used in the specification. Special characters of regular expressions (*e.g.* "`(`", "`)`", "`?`") are surrounded with the symbol "`%`" so that in a later step of the approach a pattern matcher can distinguish parts of the specification that should be interpreted as regular expressions from parts that should be interpreted as static verbatim text.

### Types

`Any_Type()` matches all possible types in the generated code. Note that the indication **`post`** `(trim())` following the declaration of the template means that the result of the template is always trimmed from leading and trailing whitespace characters.

```
1 [template public Any_Type(arg : OclAny) post (trim())]
2 %<(GAINT8|GAINT16|GAINT32|GAUINT8|GAUINT16|GASINGLE|GAREAL|GABOOL)>%
3 [/template]
```

### Casting

`Optional_Cast(exp : String)` matches either `exp` as is, or an expression that casts `exp` into another type.

```
1 [template public Optional_Cast(exp : String) post (trim())]
2 %<(>%(([Any_Type()/]) [exp/])%<|>%[exp/]%<)>%
3 [/template]
```

### Indexing and Casting

The following set of `at(...)` templates allow to match expressions that access an array source code variable at a specific index, with an optional casting of the read value. For example, if the source code contains a bidimensional array variable `MyVar`, then using `['MyVar'.at(1,1)/]` in a specification template would match any of the following expressions in the code:

```
MyVar[1][1]

((GAINT8) MyVar[1][1])
```

```
((GAREAL) MyVar[1][1])
```

Thus the following regular expressions are used to match the above expressions in the case of unidimensional and bidimensional arrays.

```
1 [template public at1(variableName : String, idx1 : Integer) post (trim())
    ]
2 %<(>%(([Any_Type()/]) %[variableName/]['['/][idx1/][']'/])%<|>%[
    variableName/]['['/][idx1/][']'/]%<)>%
3 [/template]
4
5 [template public at(variableName : String, idx1 : Integer, idx2 :
6 Integer) post (trim())]
7 %<(>%(([Any_Type()/]) %[variableName/]['['/][idx1/][']'/]['['/][idx2/][']
    '/])%<|>%[variableName/]['['/][idx1/][']'/]['['/][idx2/][']'/]%<)>%
8 [/template]
```

At this stage we have defined common Simulink concepts and common definitions and regular expressions. We are now ready to specify the generated code.

## 9.5 Tool Operational Requirements

Specifying the generated code consists in defining for each Simulink block type, the code that is generated in each of the 5 code sections of the generated code: persistent variables, init statements, local variables, compute statements and update statements. For each block type, a directory named after the block type is created `<block type>/` to contain all its specification artifacts.

The generated code depends on the configuration of a block. Each block type has a different set of configuration parameters. For example the **UnitDelay** block type is configured with the *InitialCondition* parameter while the **Gain** block type is configured with the *Multiplication* and *Gain* parameters. Therefore we first need to define for each block type its configuration parameters, in order to reference them in the specification.

### 9.5.1 Defining Configuration Parameters

Block parameters are defined as Acceleo queries in the module `<block type>/Parameters.mtl`. These queries return instances of *DataParameter* for numeric parameters, or directly *String*s for string parameters. Parameter queries call the EOperation *Block::get_param(String)* to retrieve a parameter's value from the generic parameter map defined in the metamodel.

For example, the **UnitDelay** block type has a numeric parameter called *InitialCondition* that specifies the output value of the block on the first iteration of the computation algorithm. We define this parameter with the following query.

Listing 9.6: `UnitDelay/Parameters.mtl`

```
1 [query public InitialCondition(block : Block) : DataParameter =
2 block.get_param('InitialCondition') /]
```

### 9.5.2 Specifying Code Patterns

The generated code is specified in terms of the content of the code sections identified in the general code structure. For each block type, the content of each section is specified by a module named after the code section. A module `<block type>/Definitions.mtl` may contain definitions specific to the block type that are used across multiple code sections, and the following modules specify the content of code sections:

`<block type>/Persistent_Variables.mtl`

`<block type>/Init.mtl`

`<block type>/Local_Variables.mtl`

`<block type>/Compute.mtl`

`<block type>/Update.mtl`

Each module contains one or more specification templates having the same name, but different guards. Each guard describes a different configuration of the block, and thus each specification template defines a different pattern of code for each configuration.

We illustrate this specification scheme for the **UnitDelay** block type.

**Example 9.1 (Specification of UnitDelay).**

**Definitions** A set of definitions specific to the **UnitDelay** type is first defined in `UnitDelay/Definitions.mtl`.

Listing 9.7: `UnitDelay/Definitions.mtl`

```
1 [** Name of the persistent variable /]
2 [query public MemVar(block : Block) : String =
3 block.Name().RemoveSpaces() + '_memory' /]
```

Then, the content of each code section is specified in a separate module as follows.

### Persistent Variables

Listing 9.8: `UnitDelay/Persistent_Variables.mtl`

```
1  [** A persistent variable of the same type as the outport shall be
2      declared /]
3
4  [** When the output data type is scalar, the persistent variable is
5      scalar /]
6  [template public Persistent_Data(block : Block)
7  ? (block.BlockType() = 'UnitDelay' and block.OutDataType().IsScalar()]
8  [block.OutDataType().BaseType.TypeInCode()/] [block.MemVar()/];
9  [/template]
10
11 [** When the output data type is a vector, the persistent variable is
12     a unidimensional array of the same size as the vector. /]
13 [template public Persistent_Data(block : Block)
14 ? (block.BlockType() = 'UnitDelay' and block.OutDataType().IsVector()]
15 [block.OutDataType().BaseType.TypeInCode()/] [block.MemVar()/]['['/][
       block.OutDataType().NumElements()/][']'/];
16 [/template]
17
18 [** When the output data type is a matrix, the persistent variable is
19     a bidimensional array of the same size as the matrix. /]
20 [template public Persistent_Data(block : Block)
21 ? (block.BlockType() = 'UnitDelay' and block.OutDataType().IsMatrix()]
22 [block.OutDataType().BaseType.TypeInCode()/] [block.MemVar()/]['['/][
       block.OutDataType().NRows()/][']'/]['['/][block.OutDataType().NColumns
       ()/][']'/];
23 [/template]
```

### Init

Listing 9.9: `UnitDelay/Init.mtl`

```
1  [template public Init(block : Block)
2  ? (block.BlockType() = 'UnitDelay' and block.OutDataType().IsScalar())]
3  [block.MemVar()/] = [block.InitialCondition().at(1)/];
4  [/template]
5
6  [template public Init(block : Block)
7  ? (block.BlockType() = 'UnitDelay' and block.OutDataType().IsVector())]
8  [for (it : Integer | Sequence{0..block.OutDataType().NumElements() - 1})]
```

```
9     [block.MemVar().at(it)/] = [block.InitialCondition().at(1 + it) /];
10  [/for]
11  [/template]
12
13  [template public Init(block : Block)
14  ? (block.BlockType() = 'UnitDelay' and block.OutDataType().IsMatrix())]
15  [for (r : Integer | Sequence{0..block.OutDataType().NRows() - 1})]
16    [for (c : Integer | Sequence{0..block.OutDataType().NColumns() - 1})]
17      [block.MemVar().at(r,c)/] = [block.InitialCondition().at(1+r, 1+c) /]
            ;
18    [/for]
19  [/for]
20  [/template]
```

Note that the indexing operation **at** in the above templates introduces optional casts that avoid cluttering the specification with details regarding the casting of expressions. The aspects of casting are specified separately, outside the scope of this specification.

Additionally, note that the **for** construct above is an Acceleo loop and not a C language loop. This indicates that the initialisation of the memory variable is not done using a C loop but rather using multiple assignment statements, one for each index of the variable. There is no loop statement in the generated code. The absence of a loop is one of the important pieces of information that this specification conveys regarding the structure of the generated source code.

### Local Variables

Listing 9.10: `UnitDelay/Local_Variables.mtl`

```
1   [template public Local_Variables(block : Block)
2   ? (block.BlockType() = 'UnitDelay' and block.OutDataType().IsScalar())]
3   [block.OutDataType().BaseType.TypeInCode()/] [block.OutVar()/];
4   [/template]
5
6   [template public Local_Variables(block : Block)
7   ? (block.BlockType() = 'UnitDelay' and block.OutDataType().IsVector())]
8   [block.OutDataType().BaseType.TypeInCode()/] [block.OutVar()/]['['/][
        block.OutDataType().NumElements()/][']'/];
9   [/template]
10
11  [template public Local_Variables(block : Block)
12  ? (block.BlockType() = 'UnitDelay' and block.OutDataType().IsMatrix())]
13  [block.OutDataType().BaseType.TypeInCode()/] [block.OutVar()/]['['/][
        block.OutDataType().NRows()/][']'/]['['/][block.OutDataType().NColumns
        ()/][']'/];
```

```
14 [/template]
```

### Compute

Listing 9.11: `UnitDelay/Compute.mtl`

```
1  [template public Compute(block : Block)
2  ? (block.BlockType() = 'UnitDelay' and block.OutDataType().IsScalar())]
3  [block.OutVar()/] = [block.MemVar().Optional_Cast()/];
4  [/template]
5
6  [template public Compute(block : Block)
7  ? (block.BlockType() = 'UnitDelay' and block.OutDataType().IsVector())]
8  for (i = 0; i <= [block.OutDataType().NumElements() - 1/]; i++) {
9    [block.OutVar().at('i')/] = [block.MemVar().at('i')/];
10 }
11 [/template]
12
13 [template public Compute(block : Block)
14 ? (block.BlockType() = 'UnitDelay' and block.OutDataType().IsMatrix())]
15 for (i = 0; i <= [block.OutDataType().NumElements() - 1/]; i++) {
16   for (j = 0; j <= [block.OutDataType().NumElements() - 1/]; j++) {
17     [block.OutVar().at('i', 'j')/] = [block.MemVar().at('i', 'j')/];
18   }
19 }
20 [/template]
```

Note that in the above, the **`for`** construct is a C loop and not an Acceleo loop. Therefore the generated code should contain a **`for`** statement (in the case of vectors and matrices).

### Update

Listing 9.12: `UnitDelay/Update.mtl`

```
1  [template public Update(block : Block)
2  ? (block.BlockType() = 'UnitDelay' and block.OutDataType().IsScalar())]
3  [block.MemVar()/] = [block.InVar()/];
4  [/template]
5
6  [template public Update(block : Block)
7  ? (block.BlockType() = 'UnitDelay' and block.OutDataType().IsVector())]
8  for (i = 0; i <= [block.OutDataType().NumElements() - 1/]; i++) {
9    [block.MemVar().at('i')/] = [block.InVar().at('i')/];
10 }
11 [/template]
```

```
12
13 [template public Update(block : Block)
14 ? (block.BlockType() = 'UnitDelay' and block.OutDataType().IsMatrix())]
15 for (i = 0; i <= [block.OutDataType().NRows() - 1/]; i++) {
16   for (j = 0; j <= [block.OutDataType().NColumns() - 1/]; j++) {
17     [block.MemVar().at('i', 'j')/] = [block.InVar().at('i', 'j')/];
18   }
19 }
20 [/template]
```

We have thus specified the code generation strategy for the **UnitDelay** block using the C concrete syntax, OCL queries to the input model and regular expressions encapsulated in reusable operations.

□

Having presented our specification approach and illustrated it over an example, we now explain how such a specification can be used as an automatic test oracle.

## 9.6 Automatic Test Oracles

### 9.6.1 Determining Test Outcomes

Specification templates are executable. As illustrated in Figure 9.4, the execution of specification templates over input test models yields so-called *expected patterns* of text to be matched in the actual output to determine whether it complies with the specification. Given an input test model, each specification template is executed only for input elements of the template that satisfy its guard. A template is executed by evaluating all the OCL expressions and replacing them with the evaluation results, thus producing the so-called expected pattern. An expected pattern is a block of text composed of verbatim text (where queries have been replaced by query results) and regular expressions. Essentially the expected pattern is a large regular expression that should be matched in the actual output of the test.

In our application of the approach to QGen, each Simulink block in an input test model yields a set of expected patterns, one for each code section: `Persistent_Variables`, `Init`, `Local_Variables` *etc.* As previously explained, the location where each expected pattern should match in the actual output is hard coded in the *Matcher*. The Matcher locates each code section in the actual output, and matches expected patterns in their intended code section. If all expected patterns have a match, the test passes, otherwise the test fails and the non-matching expected patterns and the corresponding specification templates are reported.

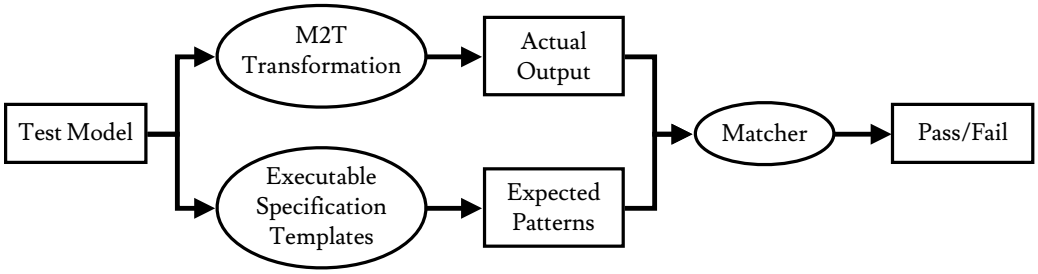



Figure 9.4: Model-to-text specification-based automatic test oracles

Each expected pattern is composed of portions of verbatim text and regular expressions. The `%< >%` delimiters of regular expressions allow the *Matcher* to *escape* verbatim text to prevent special characters such as `*` and `?` from being interpreted as regular expression commands outside of the delimiters.

A final implementation note is that Simulink test models are not directly compatible with the Eclipse Modeling Framework (EMF) and therefore cannot be given as input to Acceleo directly. We developed a simple translation from the Simulink representation to the *SimpleSimulink* EMF metamodel of Section 9.4.1 in the form of a Matlab script called *Simulink2EMF* executed in the Simulink environment. Therefore the actual implementation of our approach is depicted in Figure 9.5.

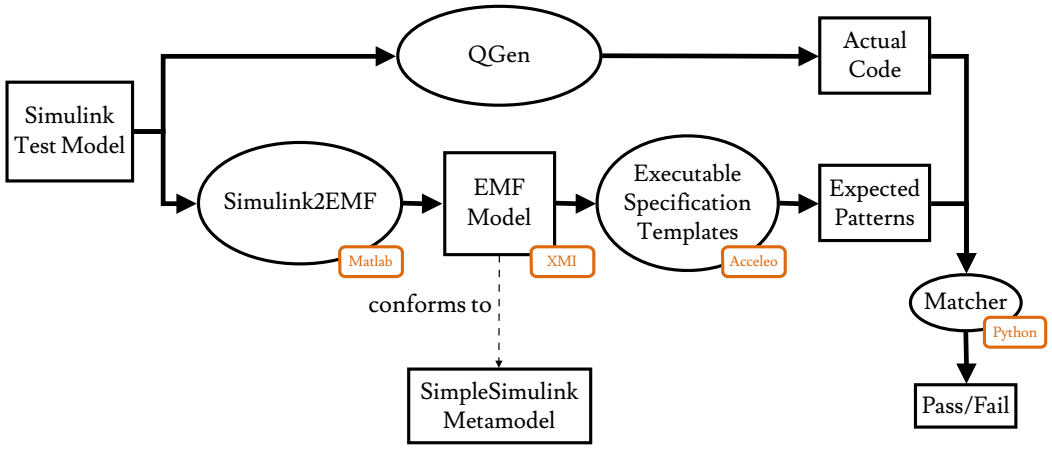
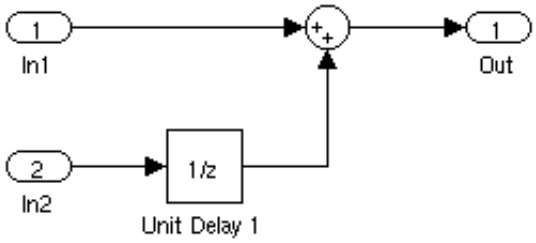



Figure 9.5: Automatic test oracles for Simulink code generation

Now we illustrate our test oracles approach on the example of the **UnitDelay** block type.

**Example 9.2 (Automatic test oracle for UnitDelay).** We recall the Simulink model from Figure 9.1 which was the following:




The data types were the following:

| Port | Base Type | Dimensionality |
|---|---|---|
| Outport of *In1* | int16 | scalar |
| Outport of *In2* | int16 | vector of size 2 |
| Inport of *Unit Delay 1* | int16 | vector of size 2 |
| Inport 1 of *Sum 1* | int16 | scalar |
| Inport 2 of *Sum 1* | int16 | vector of size 2 |
| Outport of *Sum 1* | int16 | vector of size 2 |
| Inport of *Out* | int16 | vector of size 2 |

Executing the specification templates for the above test model yields the following expected patterns for block *Unit Delay 1*:

**Persistent Data**

```
GAINT16 Unit_Delay_1_memory[2];
```

**Init**

```
%<(\(\((GAINT8|GAINT16|GAINT32|GAUINT8|GAUINT16|GASINGLE|GAREAL|
    GABOOL)\))?>%Unit_Delay_1_memory[0]%<\)?>% = %<(\(\((GAINT8|
    GAINT16|GAINT32|GAUINT8|GAUINT16|GASINGLE|GAREAL|GABOOL)\) )?(>%
    0%<|>%0.0%<|>%0.0E+00%<|>%GAFALSE%<)\)?>%;
%<(\(\((GAINT8|GAINT16|GAINT32|GAUINT8|GAUINT16|GASINGLE|GAREAL|
    GABOOL)\) )?>%Unit_Delay_1_memory[1]%<\)?>% = %<(\(\((GAINT8|
    GAINT16|GAINT32|GAUINT8|GAUINT16|GASINGLE|GAREAL|GABOOL)\) )?(>%
    0%<|>%0.0%<|>%0.0E+00%<|>%GAFALSE%<)\)?>%;
```

**Local Variables**

```
GAINT16 Unit_Delay_1_out[2];
```

**Compute**

```
for (i = 0; i <= 1; i++) {
```

```
    Unit_Delay_1_out[i] = %<(\(\((GAINT8|GAINT16|GAINT32|GAUINT8|
        GAUINT16|GASINGLE|GAREAL|GABOOL)\) )?>%Unit_Delay_1_memory[i]
        %<\)?>%;
}
```

**Update**

```
for (i = 0; i <= 1; i++) {
    %<(\(\((GAINT8|GAINT16|GAINT32|GAUINT8|GAUINT16|GASINGLE|GAREAL|
        GABOOL)\) )?>%Unit_Delay_1_memory[i]%<\)?>% = %<(\(\((GAINT8|
        GAINT16|GAINT32|GAUINT8|GAUINT16|GASINGLE|GAREAL|GABOOL)\) )?
        >%In1[i]%<\)?>%;
}
```

The actual code for this test model was presented in Listing 9.1. It is given along with the expected patterns as input to the *Matcher* which checks that all of the above patterns match in the generated code. In this instance, all patterns match indicating that the generated code conforms to the specification regarding the **UnitDelay** block type for this test model. If the expected patterns of all block types involved in the test model also match, then the test passes.

□

### 9.6.2 Resolving Failed Tests

When expected patterns don't match in the generated code, the test is marked as failed indicating a discrepancy between the implementation and the specification. The matcher reports the expected patterns that did not match as well as well as the corresponding specification templates and the input blocks. The developer then needs to compare the specification with the actual code and determine what the discrepancy is.

For obvious cases such as a missing statement or a wrong type in a variable declaration, finding the discrepancy is pretty easy. For more subtle differences such as missing parentheses in expressions, the discrepancy can be hard to find. The developer may inspect the expected patterns to find the culprit, however as seen in Example 9.2, expected patterns involving regular expressions can be very complex. As will be detailed in Chapter 10, this is one of the limitations that has become evident in the experimental evaluation of our approach with developers at AdaCore.

A possible solution would be to adopt a more sophisticated algorithm in *Matcher*. Instead of matching the complete pattern at once, the matcher could divide the pattern into subparts, and try to match them one after the other. When a subpart does

not match, the matcher would indicate the non-matching subpart, thus giving more precise feedback to the developer regarding the location of the discrepancy.

## 9.7 Document Generation: a Higher-Order Transformation

Our specification approach formalises Tool Operational Requirements (TORs) into precise specification templates. TORs must ultimately be integrated with other documents and presented to a certification authority as qualification evidence. To this end, we propose to implement automatic document generation to transform our Acceleo-based specification into another representation suitable for integration in a document.

Thanks to the choice of Acceleo as a host language for the specification, document generation is in fact easy to implement. The Acceleo framework provides an API to parse Acceleo modules and obtain a structured EMF model of the specification. This model conforms to a metamodel of the Acceleo concrete syntax and exposes templates, guards and OCL expressions in a well structured manner. Document generation is then a matter of developing a model-to-text transformation to generate the document corresponding to the specification.

We have chosen Sphinx[2] as a target document format because it is the language used for other qualification documents at AdaCore. We have implemented document generation in Acceleo itself. This means that our document generator is a *higher-order transformation* implemented in Acceleo, that takes as input other Acceleo transformation *i.e.* the TORs formalised as specification templates. This is depicted in Figure 9.6. The document generator is implemented such that the resulting document is consistent with other manually written qualification documents and is easily integrated with them.

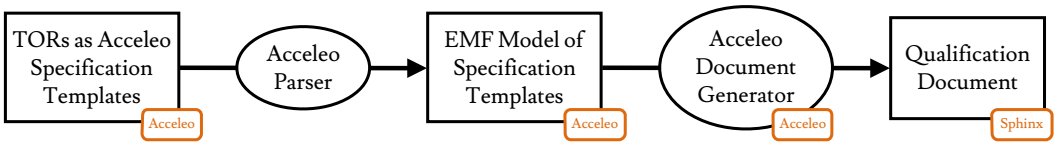



Figure 9.6: Generation a qualification document from the Acceleo-based specification

[2]Sphinx, `http://sphinx-doc.org/`

## 9.8 Related Work

To the best of our knowledge there is only one existing approach proposed in [Wimmer and Burgueño, 2013] for the problem of model-to-text specification and test oracles. We have detailed this approach in Section 3.8.2. Now that we presented our approach, we would like to highlight the similarities and differences with respect to the approach of Wimmer *et al.*. First and foremost we should recall that our approach is not yet generalised to arbitrary model-to-text transformations while the approach of Wimmer *et al.* is a general one. However it is still useful to consider a hypothetical generalisation of our approach and draw some useful comparative conclusions.

Recalling the approach of Wimmer *et al.*, it consists in viewing the generated textual artifacts as a rough model composed of *Folders* containing *Files* containing *Lines*. However the generated content is not modeled beyond this level of detail to preserve its textual nature. Based on this general model, the specification problem is brought back to model-to-model problem, and OCL contracts can be written to specify the generated source code in terms of portions of actual code that should be generated verbatim, combined with regular expressions expressing textual patterns like in our approach.

First, the approaches are similar in the fact that they consider text as unstructured content (except for folder and file organisation). Both approaches use portions of verbatim text and regular expressions to specify the generated text. However our approach combines verbatim text with regular expressions into a specification template while the approach of Wimmer *et al.* embeds them into OCL constraints. To illustrate the difference, we consider a simple example of a UML to Java code generation and consider a portion of the specification stating that for each non-derived *Attribute* of a UML *Class*, a setter method should be generated with a specific name (omitting details about the specific file in which the setter should be generated). Listing 9.13 formalises this requirement in the approach of Wimmer *et al.* where `f.contents()` returns the content of the *File* `f` as a string, and `str.matchesRE(regexp)` returns `true` if the string `str` matches the regular expression `regexp`. Listing 9.14 formalises the same requirement in our approach using a specification template. In both listings, the regular expressions `\s*` and `\s+` respectively match *0 or more* and *1 or more* whitespace characters.

Listing 9.13: Wimmer *et al.* M2T specification example

```
1 Attribute.allInstances()->select(a | a.isDerived)->forAll(a |
2   File.allInstances()->exists(f | f.contents().matchesRE(
3     a.type + '\\s+set' + a.name.toFirstUpper())))
```

Listing 9.14: M2T specification template example

```
1 [template AttributeGetters(a : Attribute) ? (a.isDerived)]
2 %<\s*>%[a.type/]%<\s+>%set[a.name.toFirstUpper()/]
3 [/template]
```

We notice that the approach of Wimmer *et al.* is very expressive since arbitrary OCL constraints can be used by the specification. In comparison, an isolated specification template as above is not very expressive because it can only specify the existence of one pattern. However a generalisation of our approach could embed specification templates into a richer constraint language that can achieve an expressiveness comparable to OCL. If such a generalisation is achieved to reach equivalent expressiveness, it would then be interesting to compare other factors such as the ease of use of both approaches in the specification of the same transformation.

However, in our context of qualification the limited semantics of specification templates plays to our advantage. The fact that our approach uses a *guard* separated from the textual pattern isolates the conditions of applicability of the pattern clearly, while in the approach of Wimmer *et al.* conditions and patterns can be mixed arbitrarily. The simplicity of specification templates makes them easy to understand and presentable as TORs. That would be difficult to achieve with the approach of Wimmer *et al.* and would require defining rules and guidelines to prevent the use of arbitrary OCL constraints and keep constraints to a simple form. This is not unusual and is customary when applying a general approach to a specific context. It would be interesting to apply a constrained version of the approach of Wimmer *et al.* to our use case, and compare it with our approach from the perspective of suitability for qualification.

Finally, it should be noted that the approach of Wimmer *et al.* also allows the specification of text-to-model parsing transformations. This is beyond the scope of our work, and we believe this would not be feasible even with a generalisation of our approach.

## 9.9 Conclusion

In this chapter we have detailed our specification and automatic test oracles approach for model-to-text transformations, applied to the QGen Simulink to C code generator developed at AdaCore. We introduced the notion of a specification template which is at the core of our approach and allows the specification of code in terms of its textual concrete syntax. We proposed to implement specification templates in the Acceleo template language. Applying our approach to QGen required to define a metamodel to describe input Simulink models and to decompose the specification into a library of Simulink-specific concepts and a library of common definitions used across the specification. Finally, we specified part of the TORs of QGen using specification templates to define the code patterns corresponding to each type of Simulink blocks.

Based on this executable specification we have proposed an automatic test oracle approach that executes specification templates to obtain expected patterns which are then matched in the actual output of a test. A test passes when all the expected patterns are successfully matched, indicating that the tested transformation complies with its specification for that particular test.

Finally, we have proposed to generate qualification documents from the Acceleo-based TORs by implementing a higher-order model-to-text transformation: the document generator is implemented in Acceleo and takes as input Acceleo templates. This allows to obtain a TORs document that is consistent with other qualification documents and easily integrated with them.

This concludes the presentation of all the contributions of this thesis which were detailed throughout chapters 6, 7, 8 and 9. We now move to the experimental evaluation of our proposals in Chapter 10.

# Chapter 10

# Experimental Validation

## Contents

## 10.1 Introduction

In this thesis we have tackled two main problems in the context of the testing of model transformation chains. The first problem stemmed from the observation that integration testing of model transformation chains is easier to carry out than unit testing, even though both are necessary for qualification. With existing approaches of the state of the art it is possible to assess the coverage of unit test requirements with existing integration tests, but what is missing is a way to produce new integration tests targeting non-covered unit test requirements. This led us to the first problem of this thesis which is the translation of unit test requirements which are constraints over intermediate representations within the chain, into equivalent constraints over the input of the chain. We have determined that the key to solving this problem is to reason step-by-step on each transformation of the chain, and we have thus proposed an analysis that transforms a postcondition of a transformation into an equivalent precondition over its input. Starting with a specification of the model transformation in the ATL language our approach is composed of 2 steps:

1. $ATL2AGT$: Translate the ATL transformation to the theoretical framework of Algebraic Graph Transformation (AGT)

2. $Post2Pre$: Use the theoretical construction of Weakest Liberal Precondition ($wlp$) to transform the postcondition into an equivalent precondition

Having advocated for the use of integration tests in this first part of the thesis, in the second part we approached the problem of the oracles of these tests. At the request of AdaCore, the industrial partner of the thesis, the problem was approached in the specific context of a Simulink to C code generator called QGen developed at the company. We determined that existing approaches cover the semantical aspect of testing sufficiently. For this reason we chose to focus on the syntactical aspect of code which was less investigated in the literature. Given the context of qualification it was important that our test oracles be based on the Tool Operational Requirements (TORs) which constitute the specification of the code generator and are part of the qualification evidence. The problem was therefore to devise a TOR specification approach focusing on the syntax of the generated source code and supporting automatic test oracles. We proposed such an approach by introducing the notion of executable *specification templates* which describe the syntax of generated code in a manner compatible with qualification needs and that serve as automatic test oracles in integration testing.

Having detailed our solutions in the previous chapters, we now present the validation activities that we carried out to assess our proposals. Section 10.2 will

focus on *ATL2AGT*, Section 10.3 on *Post2Pre* and Section 10.4 will address the specification and test oracles approach.

## 10.2 Validation of *ATL2AGT*

*ATL2AGT* translates ATL transformations into equivalent AGT transformations. In our work we considered the declarative subset of ATL, and identified two major challenges: the translation of ATL resolve mechanisms which do not have an equivalent in the AGT semantics, and the translation of OCL expressions into equivalent Nested Graph Conditions (NGC) in AGT with support for ordered sets in ATL bindings. We proposed a translation that solves both challenges, building on existing work on the translation of OCL to NGC, and we implemented it targeting the Henshin AGT framework. The objective of the validation presented next is to verify that the resulting AGT transformation has the same execution semantics as the ATL transformation.

### 10.2.1 Functional Validation Using Back-to-Back Testing

#### Back-to-Back Testing

*ATL2AGT* is essentially a model transformation where the input and the output are executable. In Section 3.8.1 of our literature review, we had discussed that a common way to verify this kind of transformations is through so-called *back-to-back* testing. We have adopted this method in the testing of *ATL2AGT* as depicted in Figure 10.1. We have considered several test ATL transformations from various sources. Each ATL transformation is translated to an AGT equivalent transformation using our Java implementation *ATL2AGT*. The resulting AGT is validated by so-called *second-order testing*. In second-order testing, a set of test models is created and given as input to both ATL and AGT versions of the transformation, and the output models are compared using EMFCompare[1]. The comparison takes into account the ordering of elements in ordered references so that our solution to the element ordering problem is also validated. Second-order testing is thus the test oracle of our testing approach.

#### Test Data

The ATL test transformations were selected or created manually. They are of 3 categories:

[1] EMFCompare, `https://www.eclipse.org/emf/compare/`

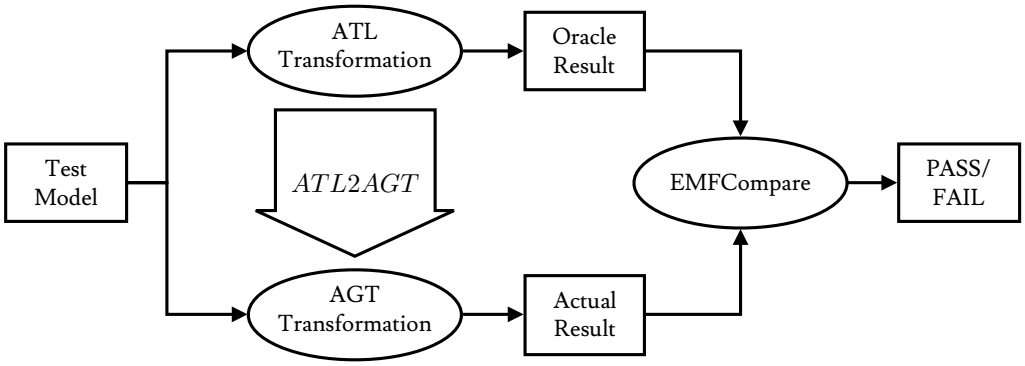



Figure 10.1: Validation of ATL2AGT

(1) Transformations taken from the ATL Zoo [ATL Zoo, accessed 2015] (*e.g.* Families2Persons, Class2Relational) or from existing papers in the literature (*e.g.* ER2REL [Büttner *et al.*, 2012b]).

(2) *SimpleCMG*: A simplified version of a step called Code Model Generation in QGen, the Simulink to C code generator developed at AdaCore. QGen itself is not developed in ATL but we modeled this step in ATL.

(3) Transformations written manually or obtained by modifying the above transformations to test specific ATL features.

In our selection of ATL test transformations, particularly in (3), we were careful in identifying the ATL and OCL features involved in these transformations (*e.g.* default and non-default resolving, use of complex OCL features such as nested iterators *etc.*). This is to ensure that all features are correctly supported, both on the conceptual and implementation level. For each test transformation, the set of test models used in second-order testing was also selected or created manually. Similarly, the selection/creation of test models was driven by the targeted features of ATL and OCL to ensure these features are indeed exercised by second-order tests.

**Implementation of the Validation**

Concretely we have implemented this validation scheme using the JUnit framework[2]. For each transformation we have created one JUnit test case that executes *ATL2AGT* and verifies that no errors have occurred during the translation (*e.g.* unsupported language constructs encountered or unexpected runtime exceptions). Then in second-order testing, for each test model a JUnit test case is created which invokes the ATL and AGT transformations in turn and invokes EMFCompare over

[2]JUnit, `http://junit.org/`

the output models to check for differences. The creation of JUnit test cases is *dynamic* meaning that adding a new test transformation or a new test model is a simple matter of adding the transformation/model file at the appropriate location in the test data hierarchy. This allows to easily extend the test set with manual test data as well as automatically generated test data in the future. Finally, the JUnit framework allowed to easily execute the complete test set and monitor results for non-regression as the prototype was being developed.

### Validation Results

Our prototype was successfully validated with all test transformations of which a subset is reported in Table 10.1. For each transformation the table shows the number of ATL $\text{rules}_{\text{ATL}}$ and bindings, and the number of resulting instantiation and resolving $\text{rules}_{\text{AGT}}$. As stated earlier we have listed the key ATL features supported by *ATL2AGT* and identified in which test transformation they occur in order to make sure all features are tested. The platform used for the experiments is a standard laptop hosting a 3 GHz Intel Core i7 (dual core) and 6 GB of memory allocated to the Java Virtual Machine.

| | **Families2-Persons** | **Class2-Relational** | **ER2REL** | **QGen Code Generation** |
|---|---|---|---|---|
| **Metrics** | | | | |
| ATL rules | 2 | 6 | 6 | 6 |
| ATL bindings | 2 | 22 | 13 | 30 |
| Instantiation rules | 2 | 6 | 6 | 6 |
| Resolving rules | 8 | 23 | 15 | 32 |
| **ATL Features** | | | | |
| Default Resolve | X | X | X | X |
| `resolveTemp` | | | | X |
| `if-then-else` | X | | | |
| Helpers | X | X | | X |
| Attribute binding | X | X | X | X |
| Reference binding | | X | X | X |
| `OrderedSet{}` | | X | | X |
| `union()` | | X | X | X |
| `select()` | | X | | X |
| `collect(),at()` | | | | X |
| **Timing** | | | | |
| Translation time | 230ms | 83ms | 70ms | 819ms |
| ATL run time | 397ms | 459ms | 377ms | 476ms |
| AGT run time | 1.3s | 1.4s | 1.2s | 1.6s |

Table 10.1: List of test transformations and tested features

We note first in Table 10.1 that in *Families2Persons*, the number of resolving rules is high given that they result from only 2 bindings. This is due to the translation scheme that we chose for nested conditional `if-then-else` expressions in bindings. The translation scheme consists in flattening the nested conditional expressions by conjuncting the conditions of each nesting level and creating multiple resolving rules, one for each conditional branch. As a result, a bindings with 1 level of nested `if-then-else` statements yields 4 resolving rules, and 2 such bindings yield the 8 resolving rules that we observe. An alternate translation scheme may be adopted in future work by creating a single resolving rule, and performing the flattening of nested conditions within its application condition. Even though both translation schemes have the same execution semantics, they may have a different impact on the analysis of the resulting AGT. For example in the *Post*2*Pre* analysis, reducing the number of rules may yield a faster analysis and a result that is easier to interpret. In a broader scope, an interesting future research topic would be investigating how different translation strategies could affect aspects of the analysis of the resulting AGT.

### 10.2.2 Discussion on the Thoroughness of Validation

We have adopted a validation approach based on testing. As was discussed on several occasions in this thesis, the quality of the test data is essential to the quality of testing, and it is ensured by assessing the test set with respect to an appropriate test adequacy criteria. However in our validation we were unable to apply such a systematic strategy mainly for lack of time and manpower. As a result we wonder if the lack of thoroughness of our validation hinders the use of our approach.

We use $ATL2AGT$ for the generation of integration tests targeting the coverage of unit test cases of model transformation chains. In that context the consequences of an error in $ATL2AGT$ are not critical because during the execution of integration tests, unit test cases are evaluated to confirm that they have been covered. In other words, we always confirm the quality and relevance of an integration test after its creation. This *safety net* reduces the need for strong correctness guarantees on $ATL2AGT$ and alleviates the lack of systematic validation.

However the use of $ATL2AGT$ in a different context than the one in this thesis may require a stronger validation of its correctness. For example, if a formal proof of correctness is performed on the AGT transformation resulting from $ATL2AGT$ and the correctness of the original ATL transformation is claimed by extension, then a more thorough validation of $ATL2AGT$ would be necessary.

In any case, the implementation of our testing strategy discussed earlier allows an easy integration with systematic criteria-driven test generation approaches if deemed necessary in the future.

### 10.2.3 Discussion on Performance

In Table 10.1 we have provided rough measurements of the time taken by the translation itself as well as the execution of ATL and AGT transformations in second-order testing. The measurements of the translation time indicate that there is no major performance cost involved in *ATL2AGT*. As for second-order testing, we notice that AGT transformations are consistently around 3 times slower than their ATL counterparts. The performance aspects of resulting AGT transformations are not a major concern in our work since the purpose of the translation is the analysis of resulting AGT transformations and not their execution. Nonetheless, it is interesting to make a few remarks regarding the potential causes of the slowness of our AGT transformations.

1. The translation of OCL expressions to NGC application conditions as per [Radke *et al.*, 2015b] sometimes results in application conditions that are sub-optimal in terms of execution performance. Simplification of conditions into equivalent and more efficient ones is also proposed in [Radke *et al.*, 2015b] but was not implemented in our work.
2. The ordering application conditions that we added to support ordered sets have a performance overhead because of the nested $\forall$ condition which must be evaluated at each candidate match and potentially requires the iteration of the entire ordered set.

$$orderingAC = \exists \left( \boxed{s:A} \xrightarrow{refB[i]} \boxed{qNode:B}, \forall \left( \boxed{s:A} \xrightarrow{refB[j]} \boxed{qNode_1:B} \mid j < i,\ wasResolved_{R1}^{t1,refE}(qNode_1) \right) \right)$$

3. Rule matching and internal trace management in the ATL virtual machine that we used (EMFTVM [EMFTVM, accessed 2015]) is highly optimised whereas in our AGT transformations it is explicit and non-optimised.
4. The test models that we used have a small size and therefore the initialization durations of the ATL and Henshin execution engines have a high significance in the measurement. Henshin embeds a JavaScript engine to evaluate attribute conditions, which we suspect to have a high initialization overhead.

At this stage the above factors are merely hypothetical as no precise performance analysis was performed. In any case, we reiterate that the more important

aspect of *ATL2AGT* in the scope of this thesis is the analysis of the resulting AGT transformation which is discussed in the next section dedicated to *Post2Pre*.

## 10.3 Validation of *Post2Pre*

*Post2Pre* is a set of constructions that transform a postcondition $d$ of a transformation $T$ into a precondition that ensures that satisfaction of the postcondition. The constructions are based on the formal Weakest Liberal Precondition construction $wlp\,(T, d)$ defined in the literature on AGT. Since this construction can theoretically be infinite, we proposed to consider a bounded version of the transformation $T_{\leq N}$ where $N$ is an arbitrary bound over the number of iterations of rules in the transformation and we introduced the theoretical construction $wlp\,(T_{\leq N}, d)$ that is always finite. To make the result applicable to the original unbounded transformation $T$, we also proposed a new construction, $scopedWlp\,(T_{\leq N}, d)$ that is a liberal (but non-weakest) precondition of $T$ and thus ensures the satisfaction of the postcondition $d$. Since the above constructions have high complexities due to their combinatorial nature, we proposed several simplification strategies to tame the combinatorial explosion.

The above constructions and simplification strategies were implemented in a Java tool based on the Henshin framework. The validation discussed in the following sections is twofold:

**Functional Validation** aims at verifying that the preconditions computed by our implementation exhibit the theoretical properties of $wlp$ and $scopedWlp$ defined in Chapter 7. To do so we considered ATL transformations translated to AGT with *ATL2AGT*, and a set of postconditions of these transformations for which we computed preconditions with our implementation. The verification consisted in manually interpreting and analysing each precondition with respect to the transformation and the postcondition. For this reason functional tests consist of small transformations (up to 3 ATL rules), small $N \leq 3$ and small postconditions (up to 2 levels of nesting) to allow manual interpretation of the result. As will be detailed shortly, visualisation and evaluation utilities were developed to assist the manual inspection.

Functional validation also aims at verifying that the simplification strategies that we introduced to limit combinatorial explosion do not affect the correctness of the result. This verification consisted in computing preconditions with and without the strategies and verifying that the results are equivalent.

**Scalability Analysis** aims at understanding the behavior of the implementation with larger data and assess the efficiency of our proposed strategies in taming the combinatorial nature of the algorithm. These tests consist of either larger transformations (up to 6 ATL rules), or the same small transformations of functional tests considered with larger $N$ or more complex postconditions. In this case the resulting preconditions are too complex to be inspected manually. Instead we will be interested in collecting metrics during the execution and analysing them.

In the following functional validation is the subject of the first section, while scalability analysis will require more discussion and will be developed over the remaining sections.

### 10.3.1 Functional Validation

Since our work on *Post2Pre* was limited to structural aspects, we only considered transformations that do not involve any manipulation of scalar attributes (*i.e.* integers, strings *etc.*). Each test in our validation is a triplet $\langle T, N, d \rangle$ based on one of the following ATL transformations:

1. *SimpleATL*: The simple ATL transformation used in all examples of Chapter 7.
2. *PointsToLines*: Another simple ATL transformation which takes as input a set of points, and produces as output lines connecting all pairs of points.
3. *SimpleCMG*: A simplified version of a step called Code Model Generation in the Simulink to C code generation chain of QGen. QGen itself is not developed in ATL but we modeled this step in ATL.

Table 10.2 summarises the characteristics of the tests based on the above transformations. Most of our tests are based on *SimpleATL*.

| $T$ | **Number of ATL Rules** | **Number of AGT Rules** | $N$ | **Number of Postconditions** |
|---|---|---|---|---|
| *SimpleATL* | 2 | 5 | 1 | 13 |
| | | | 2 | 1 |
| | | | 3 | 1 |
| *PointsToLines* | 3 | 7 | 1 | 1 |
| | | | 2 | 2 |
| *SimpleCMG* | 6 | 24 | 1 | 2 |

Table 10.2: Functional validation tests of *Post2Pre*

### Validation of Core Constructions

What we consider to be the core construction is the $wlp\,(\rho, d)$ computation for one rule $\rho$ and one postcondition $d$. It is composed of the graph overlapping algorithm, the pushout complement algorithm, and the navigation and construction of nested conditions. We validated each of these sub-parts with sets of small inputs for which we could verify the result manually. Then we also manually validated $wlp\,(\rho, d)$ with individual rules and small postconditions, and even for the sequencing of 2 rules $wlp\,(\rho_1, wlp\,(\rho_2, d))$ for which manual validation was still possible. Beyond that, manual validation was not feasible and it was necessary to execute $wlp$ for complete transformations in order to reach understandable results.

### Validation with Complete Transformations

To validate our implementation for complete transformations, we considered all tests $\langle T, N, d\rangle$ in Table 10.2, computed $wlp\,(T_{\leq N}, d)$ and manually interpreted the results to ensure that each precondition ensures the satisfaction of the postcondition within the chosen bound. This interpretation is similar to the ones we presented in the examples 7.8, 7.9, 7.10, 7.11 and 7.13 detailed in Chapter 7. In fact, the preconditions discussed in these examples are all taken from the functional tests of Table 10.2 and produced with our implementation. We provide some more examples in Annex A. Since *scopedWlp* yields larger results that are difficult to interpret manually, only the smallest tests were considered for its validation.

To assist with the manual interpretation of preconditions, we have developed a translation of Henshin nested graph conditions to a graphical representation using LaTeX[3] and GraphViz[4]. The examples of Chapter 7 and of Annex A are all produced thanks to this visualisation utility. Additionally *Post2Pre* also generates wrapper programs that allow evaluating a postcondition and the computed precondition over input and output models of the transformation to help with the manual interpretation.

For all the considered tests, the manual validation showed that the preconditions produced with the implementation conformed to the theoretical specifications of $wlp$ and *scopedWlp*. All tests were executed with the simplification strategies enabled which means that this validation also covers the strategies. However we have also performed a separate validation targeting only the simplification strategies. We detail this validation next.

[3] LaTeX, `https://www.latex-project.org/`
[4] GraphViz, `http://www.graphviz.org/`

**Validation of Simplification Strategies**

In Section 8.4 of this thesis we introduced the following 4 simplification strategies to tame the combinatorial nature of the *Post*2*Pre* constructions:

S1. NGC and Standard Logic Properties

S2. Rule Selection

S3. ATL Semantics

S4. Element Creation

It is essential to validate that these strategies do not alter the correctness of the resulting precondition. To do so we have performed two kinds of executions for each test: executions with simplification strategies disabled and executions with strategies enabled. For each pair of executions we have verified that the preconditions resulting from both executions are exactly the same using automatic model differencing in EMFCompare. We have applied this approach to strategies S3 and S4 together because they do not affect the resulting precondition. However we have considered S2 separately because as explained in Section 8.4.2 the resulting precondition may be different, in which case we manually analysed the results to understand the differences. As for S1, the strategy is deeply integrated in the implementation and it was not possible to enable/disable it separately.

First we considered the strategies S3 and S4. While keeping strategy S2 deactivated, we considered a subset of 13 functional tests (first line of Table 10.2) and ran them with and without the strategies. Automatic differencing with EMFCompare showed that in all tests the resulting preconditions were always exactly the same, confirming that these strategies do not affect the result.

Having validated S3 and S4 we kept them activated to get faster executions and considered strategy S2. Similarly, we considered the same subset of functional tests and ran them with and without strategy S2. The result with the simplification strategy is $wlp\left(T^{select}_{\leq N}, d\right)$ while the result without the strategy is $wlp\left(T_{\leq N}, d\right)$. Out of 13 considered tests, 8 yielded identical results and 5 exhibited structural differences. In the latter 5 cases manual inspection showed that $wlp\left(T^{select}_{\leq N}, d\right)$ also ensures the satisfaction of the postcondition and is therefore a valid result.

Additionally, in the 5 cases of structural differences, we were able to confirm experimentally the theoretical property speculated in Section 8.4.2 which is that despite structural differences, we have $wlp\left(T_{\leq N}, d\right) \Leftrightarrow wlp\left(T^{select}_{\leq N}, d\right)$. However this is only observed experimentally and not yet proven theoretically. Noticing also that $wlp\left(T^{select}_{\leq N}, d\right)$ is always simpler than (or identical to) $wlp\left(T_{\leq N}, d\right)$, this means that the strategy S2 of rule selection produces a simpler and yet equivalent result.

In future work it would be interesting to formally prove the speculated theoretical result so that it may be used with confidence.

With these functional tests we have validated that our implementation yields correct results. We now move to the second part of the validation which is the assessment of the scalability of the implementation.

### 10.3.2 Overview of the Scalability Problem and Current Status

In Section 8.4 we had explained that the *graph overlap operation* involved in $wlp$ is highly combinatorial which results in a significantly larger precondition after processing a rule of the analysed transformation. Since $wlp$ is recursive, this leads to a larger number of overlap operations in the processing of the next rule, cycling back to a larger precondition, and so on. As a result, we observe concretely that more memory is required to store the growing preconditions and more time is required to operate on them, resulting ultimately in an exhaustion of all available memory before the computation can terminate. In the original $wlp$ construction without our simplification strategies, the growth of preconditions is only countered by the elimination of conditions that do not have a pushout complement, or in other words, conditions that cannot be satisfied by valid executions of the transformation. However this elimination generally occurs in an advanced stage of $wlp$, a stage that is never reached because memory is exhausted before.

From this overall description of the behavior of the algorithm we extract 3 issues and tackle them separately:

**Prob1.** The graph overlapping operation generates large preconditions containing many nested conditions.

**Prob2.** Memory is depleted due to storing and processing large preconditions.

**Prob3.** The analysis takes a long time.

We have tackled each of these issues with a proposal:

**Sol1.** We proposed simplification strategies to avoid generating large preconditions. Two of these strategies consist of elimination filters based on the ATL semantics which act like the pushout complement filter but can eliminate condition at an early stage of the computation.

**Sol2.** We proposed a fragmentation scheme for conditions allowing to dump parts of the precondition on disk when memory is saturated,

**Sol3.** We proposed a parallelisation scheme for $wlp$ allowing to execute independent parts of the algorithm concurrently.

The current status of the implementation and validation of these proposals is the following. *Sol1* has been implemented completely and validated thoroughly. *Sol2* has been largely implemented but still suffers from implementation errors preventing a complete validation at this stage. *Sol3* has been completely implemented and but validated only broadly because it interacts with *Sol2* which is not mature enough yet (memory consumption increases with parallelisation).

In the following sections we will first observe the manifestation of the scalability issues *Prob1-3* on a concrete execution where our solutions have been deactivated. Then we will investigate *Sol1* and quantify and compare the benefits of our proposed simplification strategies. Finally we will report on *Sol2* and *Sol3* which are not thoroughly validated at this stage.

### 10.3.3 Concrete Observation of Scalability Issues

To observe the identified scalability issues, we will consider the ATL transformation $T = SimpleATL$ and postcondition $Post_1$ that we used in Example 7.9 of Chapter 7.

$$T = SimpleATL = R1_{Inst}\downarrow;\ R2_{Inst}\downarrow;\ R1_{Res}^{t1,refD}\downarrow;\ R1_{Res}^{t1,refE}\downarrow;\ R2_{Res}^{t,refD}\downarrow$$

$$Post_1 = \exists\left(\ \boxed{d{:}D} \xrightarrow{refE} \boxed{e{:}E}\ \right)$$

In Example 7.9 we had computed $wlp\,(T_{\leq 1}, Post_1)$ with a bound $N = 1$ and interpreted the result. We now attempt to compute $wlp\,(T_{\leq 2}, Post_1)$ with bound $N = 2$ while keeping our simplification strategies, memory fragmentation and parallelism schemes **deactivated**[5]. During this execution we monitor the following parameters:

1. The size of each postcondition $d$ and the computed intermediate precondition $wlp\,(R\downarrow_2, d)$ in terms of the number of nested conditions $\exists(C, c)$ in the post/pre-conditions.
2. The number of graph overlapping computations performed during the execution. This number is reset to 0 after each completion of $wlp\,(R\downarrow_2, d)$ for each rule $R$. This metric gives an idea of the amount of computation required for each rule.
3. Memory consumption in terms of the percentage of allocated memory in the total available memory (6 GB in this execution).

[5] except for the simplifications based on simple first-order logic properties which cannot be disabled independently in our implementation

The first observation is that we were forced to interrupt the execution prematurely because it reaches a point where the memory becomes saturated and the computation is unable to continue. This occurs during the processing of the 4th rule iteration, after successfully processing 3 rule iterations $R\downarrow_2$. For these first 3 successful computations, the sizes of the computed preconditions are as follows:

| Rule Iteration | Size of Postcondition | Size of Precondition |
|---|---|---|
| $R2_{Res}^{t,refD}\downarrow_2$ | 1 | 427 |
| $R1_{Res}^{t1,refE}\downarrow_2$ | 427 | 430 |
| $R1_{Res}^{t1,refD}\downarrow_2$ | 430 | 37843 |
| $R2_{Inst}\downarrow_2$ | 37843 | Memory Saturated |

Except for the second rule, we observe a rapid growth of the size of the precondition. As a result the amount of processing needed for each rule grows during the execution. We can observe this in Figure 10.2 where we have plotted the number of overlap computations performed during the execution, and that number is reset to 0 upon the completion of $wlp\,(R\downarrow_2, d)$ for each rule. In that graph the first rule is not visible because it only requires 404 overlap operations in total and thus ends too quickly to be visible in the metrics. Because of the large precondition resulting from this first rule, the second and third rules require respectively $\sim$150000 and $\sim$75000 overlap operations which are large numbers compared to the first rule. Finally, the processing of the forth rule goes up to $\sim$650000 overlap operations and never completes because we interrupt the execution at that point due to memory saturation as explained next.

We observe that towards the end of the execution in Figure 10.2 the slope of the curve decreases, meaning that the advancement of the computation slows down. This is because at that point the amount of allocated memory reaches $\sim$90% as shown by Figure 10.3 which causes the Java garbage collector to trigger too often. As a result, the computation is slowed down by the repeated interruptions of the garbage collector. Continuing the execution beyond that stage is pointless because the computation will keep getting slower and slower until ultimately memory is completely exhausted and the program crashes. Finally, we observe that the overall execution lasted about $\sim$25 minutes, which is significantly long given the simplicity of the processed example.

Moreover, we have attempted the same experiment on a larger platform of about 70 GB of memory and have observed the same behavior. In that case the execution progresses farther and reaches a larger number of graph overlap operations, but ultimately we observe the same stalling effect when the amount of allocated memory reaches $\sim$90% and the program crashes once the memory is completely exhausted. This indicates that simply increasing the computation power of the platform does

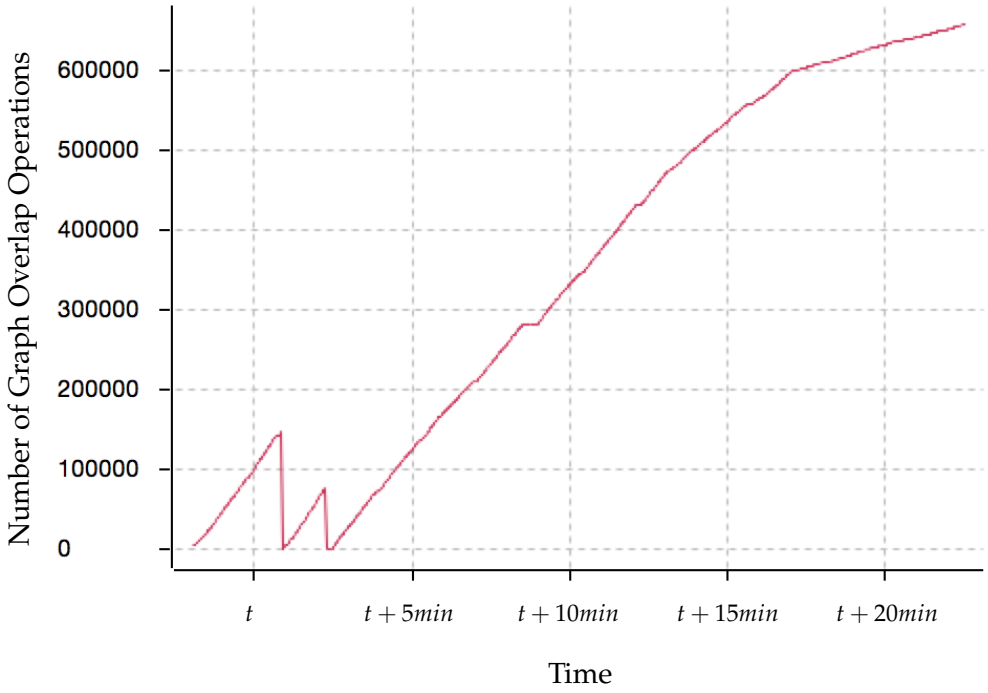



Figure 10.2: Number of overlap operations during $wlp\,(T_{\leq 2}, Post_1)$

not solve the problem.

These observations lead us to the 3 problems formulated earlier, *Prob1-3*: the large size of the precondition, the exhaustion of memory and the long duration of the overall analysis. In the next section we tackle *Prob1* with the simplification strategies of *Sol1* and quantify the reduction that they provide in terms of the size of the precondition and the amount of computation required for the analysis.

### 10.3.4 Assessment of Simplification Strategies

The combinatorial enumeration involved the graph overlap operation yields large preconditions. To reduce the size of preconditions, we proposed in Section 8.4 the following 4 simplification strategies:

S1. NGC and Standard Logic Properties.

S2. Selection of rules contributing to the postcondition.

S3. Eliminating conditions that violate ATL semantics. We refer to this strategy as the *ATLSem* filter.

S4. Eliminating conditions that contain elements that can no longer exist in graphs. We refer to this strategy as the *ElemCr* filter.

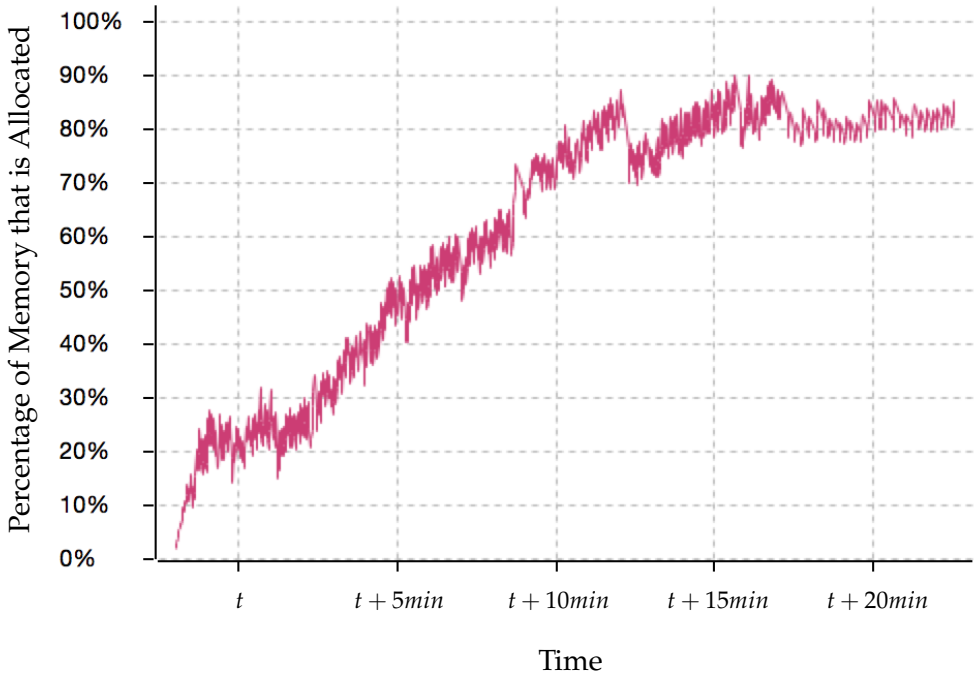



Figure 10.3: Memory usage during $wlp\,(T_{\leq 2}, Post_1)$

Now we would like to assess the efficiency of our solutions. To quantify the effect of these strategies we proceed similarly to functional validation, by enabling and disabling filters in different executions and by comparing metrics measured during executions. As mentioned previously, S1 is not configurable in our implementation, which means it will remain *enabled* in all executions.

### Experimental Protocol

We perform 5 executions of $wlp\,(T_{\leq 1}, Post_1)$ configured as follows[6]:

| Execution Identifier | S1 | S2 | S3 | S4 |
|---|---|---|---|---|
| *ExNoFilters* | ✓ | × | × | × |
| *ExATLSem* | ✓ | × | ✓ | × |
| *ExElemCr* | ✓ | × | × | ✓ |
| *ExBoth* | ✓ | × | ✓ | ✓ |
| *ExAll* | ✓ | ✓ | ✓ | ✓ |

Since S3 and S4 are of similar nature, we study them separately, while S2 will only be activated in the last execution. During each execution we measure the following parameters:

[6] we choose $N = 1$ because as seen in the previous section, with $N = 2$ and without simplification strategies the execution ends with memory saturation, preventing a complete comparison

| | |
|---|---|
| $S_{Pre}$ | The size of each computed intermediate precondition $wlp\,(R{\downarrow_2}, d)$ defined as the number of nested conditions $\exists(C, c)$ in the precondition. |
| $N_{Ov}$ | The total number of graph overlap operations performed during $wlp\,(R{\downarrow_2}, d)$ of each rule $R$. |

We also measure the number of conditions eliminated during the enumeration of overlaps, classified by the reason of their elimination. We measure this number for each rule $R$ separately:

| | |
|---|---|
| *ElPo* | A pushout complement could not be found for the condition (in *ExNoFilters*) |
| *ElATLSem* | The condition violates ATL semantics regarding trace nodes (in *ExATLSem*) |
| *ElElemCr* | The condition contains elements that can no longer exist in conditions (in *ExElemCr*) |

All experiments are conducted on the same machine: a standard laptop hosting a 3 GHz Intel Core i7 (dual core) and 16 GB of memory of which 6 GB are allocated to the Java Virtual Machine executing the analyses.

**Analysis of Experimental Results**

First we analyse $S_{Pre}$, the size of the intermediate preconditions which is in direct relation with memory consumption. Figure 10.4 plots $S_{Pre}$ for each of the 5 executions. Noting that the vertical scale is logarithmic, the linear curve that we observe for *ExNoFilters* indicates an exponential increase of the size of the precondition during the execution. In *ExATLSem* the growth is reduced, but still evolves exponentially. It is in *ExElemCr* and *ExBoth* that the exponential growth is really prevented, leading us to believe that the element creation filter is more effective than the ATL semantics filter. Evidently the best result is achieved in *ExAll* when all simplifications are activated. Note that in *ExAll* only rules that create elements involved in the postcondition are processed, and they are indicated by the dotted frame. Processing less rules means performing less overlap operations and avoiding the growth of the precondition. By the time we reach the last rule in *ExAll*, the size of the precondition is divided by 100 compared to *ExNoFilters* which is a significant reduction directly resulting in a drop in memory consumption. We conclude that our strategies are highly effective since they allow a reduction of memory consumption by orders of magnitude.

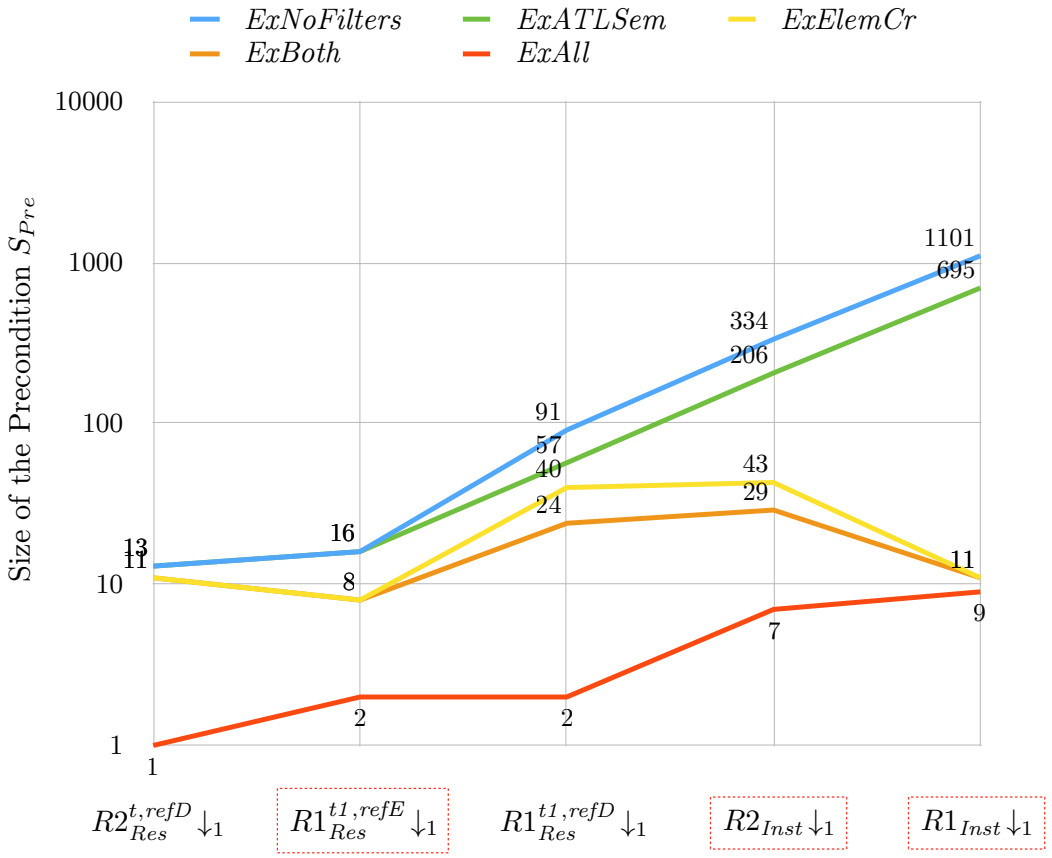



Figure 10.4: Size of intermediate preconditions $S_{Pre}$

It should be noted that the size of the precondition at the end of the all executions in the graph is not the size of the actual $wlp$ result. The final precondition is filtered one last time by eliminating all conditions containing elements of the target metamodel. This is because the precondition can only contain elements of the input metamodel since the transformation is exogenous. The final size indicated on the graphs is prior to the ultimate filtering whereas after that filtering all executions yield the exact same result[7] of size 7 which was presented in Example 7.9. Based on this, the last precondition in *ExNoFilters* of size 1101 is ultimately filtered down to a precondition of size 7 indicating that a large portion of the computations was in fact useless. Our filters *ExATLSem* and *ExElemCr* allow to avoid these useless computations early on and prevent accumulating useless results in the intermediate preconditions.

We now move to Figure 10.5 where we plotted $N_{Ov}$, the number of overlaps performed to process each rule which is an indication of the amount of computation and the amount of time required to process each rule. We also plot the total

[7]Though not always the case, the activation of rule selection in *ExAll* yields the same result in this particular example

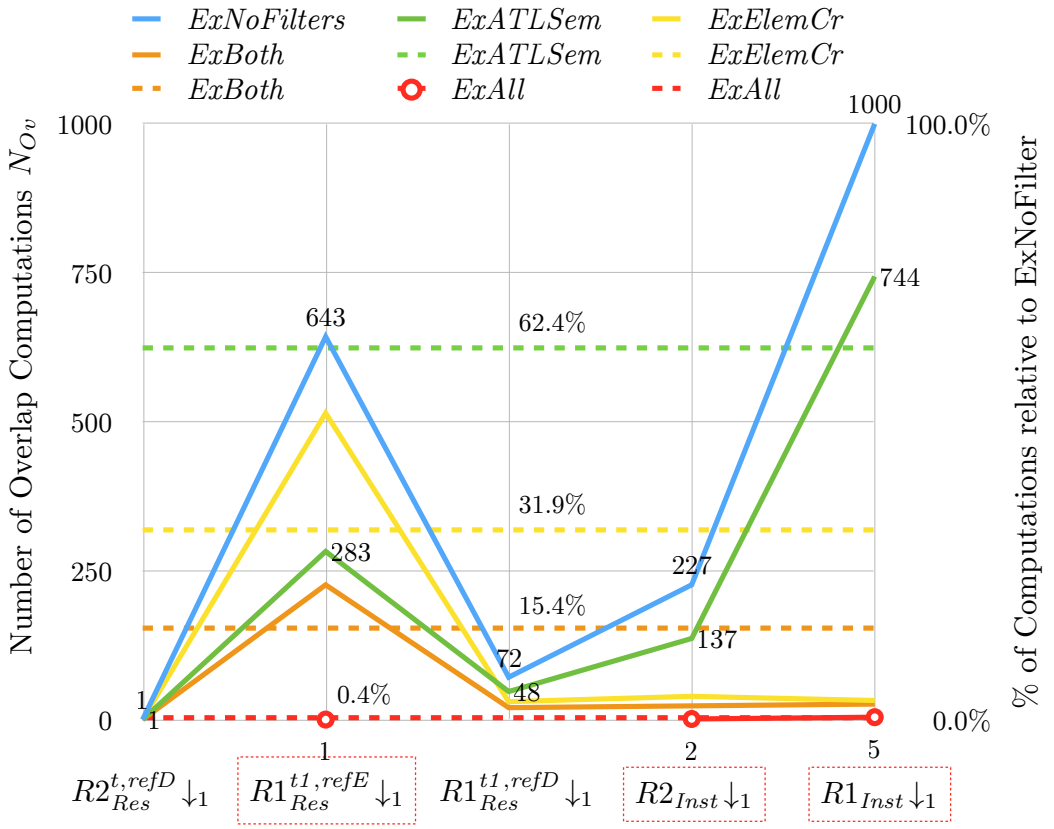



Figure 10.5: Number of overlap computations during execution $N_{Ov}$

amount of computation in each execution as the percentage of the total number of overlaps relative to the total number of overlaps in *ExNoFilters* to get an assessment of each complete execution. In *ExBoth* we observe a significant drop in the amount of computation, with only 15.4% of computation needed to compute $wlp$. This is a substantial decrease directly resulting in a major reduction of the execution time of the analysis. This is even more extreme in *ExAll* where we manage to compute the same result with only 0.4% of the amount of computation.

However it should be noted that the extreme gain in *ExAll* is due to strategy S2 of rule selection which is only efficient when the number of selected rules is small. If the postcondition involves more metamodel elements and requires for example the selection of all rules, then S2 brings absolutely no benefit and the execution would be equivalent to *ExBoth*, which still exhibits a significant gain.

Finally, we plot in Figure 10.6 the number of eliminated conditions in each execution to compare the elimination due to our filters with the inherent elimination due to the non-existence of a pushout complement. We notice that without our filters pushout-elimination only occurs in the last two rules. In fact on other examples we observe generally that pushout-elimination only occurs once we start

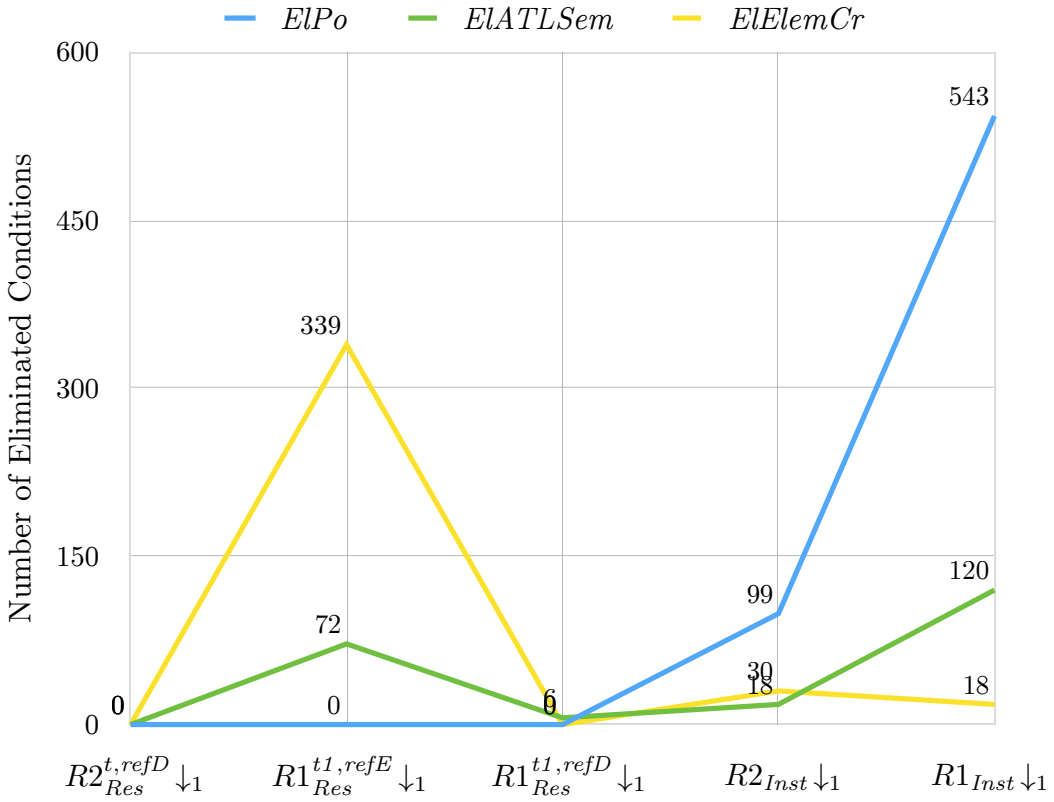



Figure 10.6: Overlaps eliminated by pushout and ATL semantics filters

processing the instantiation phase of the ATL transformations. By that time the intermediate preconditions would have already reached very large sizes. With our filters we observe that elimination starts early at the second rule, and in significant numbers, which allows to prevent the growth of preconditions as early as possible. This is why we manage to obtain the significant reduction in memory consumption and execution time observed earlier.

### 10.3.5 Larger Examples and Discussion on Complexity

We now go back to the earlier example of $wlp\left(SimpleATL_{\leq 2}, Post_1\right)$ which lead to memory saturation in Section 10.3.3 without our simplification strategies. We now carry out the same analysis with the strategies S1, S3, S4 enabled and S2 (rule selection) disabled (same configuration as *ExBoth*) and on the same execution platform, a 3 GHz Intel Core i7 (dual core) with 6 GB of memory allocated to the Java Virtual Machine. With our strategies, the analysis completes successfully after 37 seconds with a maximum memory usage of 15%. The execution with all strategies, including rule selection (same configuration as *ExAll*) completes in under 2 seconds with no significant memory usage which is an impressive result.

We also obtain good results with the *SimpleCMG* transformation which is a simplified version of a code generation step in QGen, the Simulink code generator developed at AdaCore. This transformation consists of 6 instantiation rules and 18 resolving rules. With a small postcondition containing 3 nodes and 2 edges and all simplification strategies, we are able to compute the precondition $wlp\left(SimpleCMG_{\leq 1}, d\right)$ with a bound $N = 1$ in 20 seconds. Thanks to rule selection this execution actually operates on only 10 rules of the transformation (6 resolving rules and 4 instantiation rules) instead of the total of 24 rules.

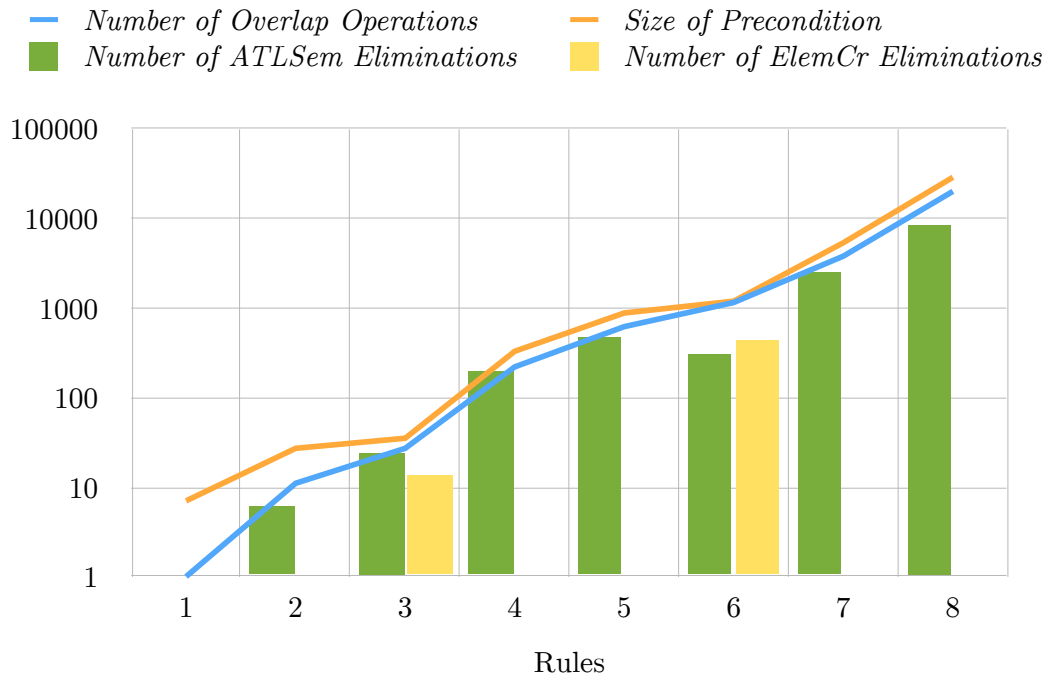



Figure 10.7: Metrics of $wlp$ execution for *SimpleCMG*

However when considering a slightly larger postcondition of 4 nodes and 3 edges, the number of selected rules increases to 15 (10 resolving rules and 5 instantiation rules). In that case the analysis cannot complete successfully and we run into memory saturation again. Figure 10.7 plots the metrics of the execution and we can see an exponential increase (the vertical scale is logarithmic) of both the size of the intermediate preconditions as well as the number of overlap operations, despite significant eliminations in our filters. Carrying out the experiment on a larger platform of 70 GB of RAM leads to the same outcome: an exponential increase of the number of overlap operations ending with memory exhaustion and a crash. We conclude that even though our filters are highly effective and push the limits of computability further than the original construction, they do not change the exponential nature of the algorithm and do not allow a true scalability for transformations and postconditions of arbitrary size.

In the future it would be interesting to investigate strategies that can have a greater impact on the complexity of the algorithm. For example as discussed in Section 8.4, the size of the graphs involved in nested conditions in terms of the number of nodes and edges plays a significant role in the exponential nature of the algorithm: we conjecture that if $E$ is the number of elements in the overlapped graphs, the number of computed overlaps is a function of $2^E$. We also observe that during the processing, the graphs of conditions first increase in size (in terms of the number of nodes and edges) during the resolving phase and then decrease in size during the instantiation phase. As a result it may be interesting to exploit this property and process rules of the instantiation phase sooner than we do currently do. Such an optimisation would act on the size of graphs which plays a significant role in the complexity of the construction, and would perhaps provide better results in conjunction with the simplification strategies proposed in this thesis.

And finally, the notion of *loop invariants* should be investigated in the future because it allows to avoid iterating over looped rules as we currently do. A loop invariant is typically provided manually by the user and allows to *short-circuit* the loop by avoiding to iterate the construction over it and using the loop invariant instead. This problem is not yet investigated much in AGT, but it is well known in traditional imperative programing languages. In the latter context, the literature indicates that loop invariant are difficult to generate automatically for arbitrary loops, but for loops with a particular form or exhibiting particular properties, automatic loop invariants are feasible. Since in our AGT transformations (translated from ATL) loops only contain rules and these rules have a specific well known form (instantiation and resolving rules), then we believe that it may be possible to propose an automatic construction of loop invariants. If feasible, this would certainly be a promising solution to the scalability issues of $wlp$.

We now move to the final aspects of the implementation which are its parallelisation and the management of the memory.

### 10.3.6 Parallelism and Memory Management

In Section 8.6 we had proposed to parallelise the implementation of the $wlp$ construction to allow the concurrent execution of independent graph overlapping operations. Parallelisation was successfully implemented and we have observed significant reduction of the overall execution time of the analysis. In Figure 10.8 we show the execution time for $wlp\left(SimpleATL_{\leq 2}, Post_1\right)$ for increasing values of the number of parallel computation threads, measured on a machine with 16 processors. We can see that parallelism reduces execution time up to 4 threads beyond

which the execution time no longer improves. This is because at that point all independent overlapping operations have been run in parallel and the threads added beyond 4 essentially do not have jobs to perform. However in larger problems such as *SimpleCMG* where a larger number of overlap operations needs to be performed, we can observe improvement of execution time beyond 4 threads and up to 16 threads.

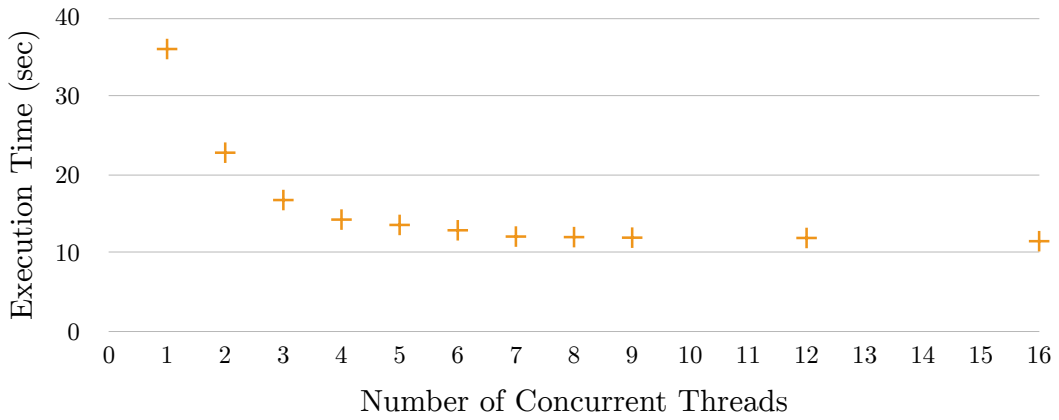



Figure 10.8: Evolution of the execution time with the number of parallel threads

In Section 8.6 we had also raised the issue of the high memory consumption of the implementation and proposed the solution of fragmenting the computed data into separate resources. This allows to dump portions of temporarily unneeded data on the disk when memory is close to saturation. When the dumped data is subsequently needed again, it is automatically reloaded on demand. Even though existing tools propose generic fragmentation solutions for EMF, they were not directly applicable to the Henshin framework that we use in our implementation. As a result, we implemented custom fragmentation/dumping/loading mechanisms for Henshin conditions built into our $wlp$ implementation.

At the current stage the implementation of these mechanisms is not yet robust enough and still exhibits runtime errors that are still under investigation. As a result all of the experiments presented in this section were conducted with memory management mechanisms disabled to avoid the aforementioned errors. However we indicate that the errors encountered concern implementation details and we believe that the proposed memory management solution, once implemented correctly, can allow the analysis to handle larger problems without memory exhaustion. In fact, early observations indicate that before the occurrence of errors, memory consumption is indeed significantly reduced by our memory management mechanisms.

### 10.3.7 Concluding Remarks

In this section we have validated several aspects of *Post*2*Pre*. First we verified that our implementation complies with the theoretical constructions and their properties. Then we analysed the scalability of our approached and showed that the simplification strategies that we proposed improve significantly the scalability of our analysis: inputs that caused memory exhaustion after close to an hour of computation can now be successfully processed in seconds thanks to our simplification strategies.

In particular, we had introduced filters eliminating irrelevant parts of preconditions based on knowledge of the ATL semantics of the analysed transformations. It is interesting to note that we were able to introduce this semantical knowledge into the analysis and achieve significant performance improvement without compromising the theoretical correctness of the analysis. This is thanks to the fact that conditions are expressed using graphs in AGT, which allows to easily characterise conditions that violate ATL semantics and eliminate them without loss of correctness. This is an advantage with respect to existing precondition synthesis approaches based on OCL [Cabot *et al.*, 2010a] where identifying OCL conditions that are incompatible with ATL semantics would be significantly harder.

Finally, despite our simplification strategies and the parallelism and memory management mechanisms, the fact remains that at the current stage, the construction is still exponential in complexity and cannot yet truly scale to real world transformations and conditions. We believe that a step towards true scalability would be through the investigation of loop invariants that would avoid the complex parts of the construction.

This concludes the validation of the second set of contributions of this thesis which aimed at transforming a postcondition of a transformation into an equivalent precondition. This was the key to transforming unit test requirements of a transformation chain up to equivalent constraints on the input of the chain and ultimately into integration tests that satisfy unit testing needs. We now focus on the latter integration tests and move to the validation of the third and final contribution of this thesis which proposed a specification approach supporting automatic test oracles for integration tests of a code generation chain.

## 10.4 Validation of Model-to-Code Specification and Test Oracles Approach

The last contribution of this thesis is a model-to-text specification and test oracles approach for the specification of Tool Operational Requirements in the context of the qualification of QGen, the Simulink to C code generator developed at AdaCore. The approach relied on the concept of *specification templates* that we introduced to specify the generated code in terms of its concrete syntax. Specification templates consisted of portions of verbatim concrete code interspersed with queries to the input model, regular expressions expressing general expected patterns, and repetition statements expressing the repetition of code patterns. Executing specification templates over test models provided expected patterns which should be matched in the generated code, thus constituting an automatic specification-based test oracle. Finally, a document generator was proposed to transform this specification into a document suitable for integration with the qualification evidence of the code generator.

To validate our proposals, we deployed this approach within the team developing the QGen toolset at AdaCore and collected feedback on its use. In the following we discuss this deployment and report on the received feedback.

### 10.4.1 Deployment of the Approach

Our approach was deployed in the form of an integrated Eclipse plugin called the *TOR Toolkit* encapsulating all of the components of the approach that we detailed in Chapter 9: the *SimpleSimulink* metamodel formalising input Simulink models and the libraries of Simulink definitions and common regular expressions used in the specification. Based on these elements the TORs would be written in the form of specification templates using the Acceleo editor. The TOR Toolkit also provided the functionalities of executing existing integration tests and validating their results with our TOR-based test oracles approach. Finally the toolkit also allowed to automatically transform the TORs written in Acceleo into a qualification document.

This experiment was launched in a relatively advanced stage of the development of QGen which was initially based on a combination of pseudo-formal and natural language specifications. At that stage a significant part of the implementation was completed and a considerable set of Simulink test models was available. The users of the TOR Toolkit had the tasks of writing TORs based on the existing informal specification, and applying the automatic TOR-based test oracles to the existing tests.

Three people (including the author of this thesis) were involved in the experiment and provided feedback on various aspects of the approach. Out of the ∼100 Simulink block types supported by QGen, 38 block types were specified with our approach, with varying degrees of completeness as will be explained in the next section. We considered block types with varying degrees of complexity in terms of the number of configuration parameters of each block type and in terms of the complexity of the generated code.

### 10.4.2 Feedback and Lessons Learned

In the following we detail the feedback gathered throughout the experiment and present it in terms of advantages and shortcomings (Pros/Cons) organised according to the three main aspects of our approach: TOR specification, TOR-based automatic test oracles and automatic document generation. Many of the observations are qualitative and lack support with quantitative data. This is because of the subjective nature of certain aspects of the problem, but also because the experiment was interrupted before significant quantitative data could be gathered. In fact the experiment was discontinued upon reaching the conclusion that even though our approach performs well for its intended scope, other problems should also be addressed before it can be realistically adopted. This will be detailed as we present the feedback of the experiment next.

In the following, Pros and Cons are interspersed to allow comparing advantages with closely related shortcomings.

#### TOR Specification

Pro1. *A precise specification language and terminology*

Compared to natural language specification, our approach had the advantage of providing a precise language and terminology yielding concise and unambiguous requirements. However this is an advantage of model-driven specification approaches in general and is not specific to our approach.

Pro2. *Inclusion of natural language*

In the context of qualification where TORs may be reviewed in later stages by non-experts, the ability to easily attach natural language explanations as comments to the precise specification was a major advantage.

Pro3. *Concrete syntax of generated code*

Writing requirements in terms of the patterns of code to be generated was perceived by users as a convenient way to express requirements. When the

approach was presented to qualification experts at AdaCore and in other organisations, they also found it is an appropriate format to present to qualification authorities.

Con1. *Verbatim text can hinder readability*

Verbatim text, in particular whitespace characters, can be tricky to use in the specification. One whitespace character in the specification means that exactly one whitespace character is required in the output. This prevents the user from adding whitespace between code elements to improve the readability of the specification.

A possible solution would be to change the semantics of specification templates such that any number of consecutive whitespace characters in the specification implicitly means that one or more whitespace characters are required in the output. This semantics would have to be also implemented in our test oracles machinery. Such a solution makes sense in the specific context of C code generation since in C any number of whitespace characters can be used to separate syntax elements. For arbitrary model-to-text transformations this solution may not apply.

Pro4. *Enumeration of supported configurations*

The limited scope of each specification template defined by the guard of the template required users to enumerate all supported configurations of each Simulink element into individual specification templates. This allowed in several cases to identify configurations missing or wrongly handled in the implementation because the initial informal specification was not detailed enough.

For example, block types such as **Compare To Constant**, **Unit Delay** and **Delay** support taking a scalar input and automatically/implicitly convert it to vector or matrix output. QGen already supported these configurations however when they were explicitly enumerated in the TORs, the specifier realised that these configurations should not be supported according to an informal user requirement that implicit dimension conversions should not be allowed for input signals. As a result these configurations were removed from the supported configurations in the TORs[8] and QGen was updated to reject these configurations even though they could be supported technically.

However despite its benefits, the enumeration of supported configuration was extremely tedious and error-prone as discussed next.

[8] they were moved to a list of configurations to be rejected by QGen

Con2. *Combinatorial enumeration is tedious and error-prone*

In Section 9.5 we had anticipated that our approach requires enumerating all possible configurations for each Simulink block type into distinct specification templates. As foreseen, this proved to be a tedious and error prone task. For blocks with many configuration parameters the number of possible configurations grows large and it is difficult to enumerate them manually without making mistakes. As a result, the specification for complex block types was often lacking configurations and even deliberately abandoned because of the large number of configurations to be addressed. For example, for the most complex blocks the number of configurations to be enumerated could reach around 25, which is not easy to perform manually. This is a major obstacle to the realistic adoption of our approach. For this reason we discuss this aspect and potential solutions in further detail in Section 10.4.3.

**Automatic TOR-based Test Oracles**

Pro5. *Detection of errors*

Our automatic test oracles successfully detected discrepancies between the actual generated code and the TORs specification that were traced back to implementation errors. The errors included for example wrong typing of generated variables (*i.e.* using a bidimensional array while a unidimensional array was required). This confirms that the core ideas of our approach are valid.

Not many errors were detected overall, but this is not due to a shortcoming of the test oracles *per se*, but rather a result of the incompleteness of the specification and of the available test set as will be discussed in Con4.

Con3. *Error feedback of test oracles is insufficient*

When a test fails, the feedback given to the developer is not sufficiently detailed to support his investigation. In case of failure the test oracle reports the expected patterns that could not be matched in the output of the test as well as the corresponding specification templates. The developer then needs to compare the specification with the actual code and determine what the discrepancy is. For obvious cases such as a missing statement or a wrong type in a variable declaration, this is pretty easy. However for more subtle differences such as misplaced parentheses in expressions, the discrepancy can be hard to find. The developer may inspect the expected patterns to find the culprit, however as seen in Example 9.2, expected patterns involve regular expressions that can be very complex.

A possible solution would be to adopt a more sophisticated algorithm in the *Matcher* component of test oracles which is responsible for matching an expected pattern in the test output. Instead of matching the complete pattern at once, the Matcher could divide the pattern into subparts, and try to match them one after the other. When a subpart does not match, the matcher would indicate the non-matching subpart, thus giving more precise feedback to the developer regarding the location of the discrepancy.

Pro6. *Requirements-based test results*

Using the requirements themselves as test oracles is a strong argument from a qualification perspective because it establishes requirements as the direct deciders of the result of the verification. Any divergence between the specification and the implementation introduced by a modification of the former or the latter, is immediately detected by the test oracles. However as discussed next, this argument alone is not sufficient: the quality of error detection depends on the completeness of the specification and the quality of the test set.

Con4. *Error detection is sensitive to the completeness of the specification and to the quality of the test set*

Overall, not many errors were uncovered by our test oracles. This was mainly due to two issues (a) the incompleteness of the specification mentioned in Con2 and (b) the incompleteness of test models in terms of coverage of possible configurations. An example of (a) is when a test model contains a configuration that is not included in the specification, a warning can be issued indicating this fact but the result cannot be validated: errors may be missed. Conversely in (b) if the specification contains a configuration that never occurs in any test model, that configuration is never tested: errors may be missed. At the time of deployment of the experiment, the test set was not mature enough which prevented errors from being detected.

Issue (b) can be solved with the test coverage analysis framework that we discussed in the first part of this thesis for the coverage of unit test requirements with integration tests. The framework introduced in Section 4.2.3 is originally based on [Bauer *et al.*, 2011] and verifies that for each test requirement, both unit and integration test requirements, there is at least one test model that satisfies it. Therefore to solve issue (b), we can opt for the simple test adequacy criteria that each specification template should be covered by at least one test, *i.e.* we produce (probably automatically) one test requirement for each specification template. Then the coverage analysis framework would automatically report non-covered test requirements for which new test models would be created. This coverage framework was not imple-

mented within this thesis, but once implemented and deployed, it can solve issue (b).

As for issue (a), *i.e.* the incompleteness of the specification, it is a well-known problem inherent to specification-based test oracle approaches in general. As discussed in Section 3.6.2 of the state of the art, we encounter this aspect in existing specification-based test oracle approaches [Guerra and Soeken, 2015] where incompleteness of the specification also prevents errors from being detected. We believe there is no general way to ensure the completeness of any kind of specification, and that this problem should be dealt with in a manner specific to the application context. In our context of Simulink code generation we will discuss possible solutions in Section 10.4.3.

**Document Generation**

Pro7. *Flexibility of the generated document format*

In our experiment we tried different formats and organisations of the TORs in the generated document. No definite format was ultimately retained mainly because we focused on the more important issues highlighted above. However the ease in which the document format could be modified thanks to the implementation of the document generator as an Acceleo transformation was a clear advantage.

**Conclusions of the Feedback**

In light of the reported feedback, we conclude that for the scope of problems initially targeted by our work, our specification and test oracles approach performs well. The original goal was to provide a specification language for writing TORs destined for qualification and an associated automatic test oracles approach, and in that scope our proposals were found appropriate. For the shortcomings Con1 and Con3 which fall within that scope, we mentioned possible improvements to address the use of whitespace in the specification and improve the feedback of test oracles.

However Con2 and Con4 uncovered the following problems which are beyond the original scope and that should be addressed for a realistic adoption of our approach:

(1) In the context of Simulink, our approach requires the manual enumeration of a large number of configurations which is a tedious and error prone operation.

(2) As a result of (1) we cannot ensure the *completeness* of the specification, which is essential to obtain a high fault revealing power.

(3) The test set must cover all configurations identified in the specification to ensure they are all exercised.

Problem (3) already has a solution with the coverage analysis framework discussed in Section 4.2.3 to support the first part of the thesis regarding the coverage of unit test requirements with integration tests. Solving (3) is therefore a matter of implementing that framework which was not done within this thesis. As for problems (1) and (2), they have potential solutions based on existing work discussed in the next section.

### 10.4.3 Addressing the Variability and Completeness of the Specification

We have seen in our experiment that the completeness of the specification is essential to the detection of errors with specification-based test oracles. Ensuring the completeness of the specification can be difficult when dealing with languages with high variability, *i.e.* language elements can have many different semantics based on their configuration. This is the case of Simulink.

Each block type in Simulink has parameters controlling its behavior. The types and dimensions (*i.e.* scalar, vector, matrix) of input and output data can also affect that behavior. Each combination of parameter values and data dimensions defines a configuration of the block that can have a different semantics. In extreme cases[9], a block can have at most as many different configurations as the cartesian product of the sets of possible parameter values and data dimensions.

In our specification it was necessary to enumerate all possible configurations of each block type, identifying valid an invalid ones. Given its combinatorial nature, this task was very tedious and highly error prone since it is easy to forget configurations, or include configurations that are invalid. Most importantly, ultimately there is no guarantee that this specification is *complete* and that no configurations were omitted.

This is a known problem that was studied in existing research concerning Simulink and dataflow languages with high variability in general [Dieumegard *et al.*, 2012; Dieumegard *et al.*, 2014b; Dieumegard *et al.*, 2014a]. These works target semantic specification and not the syntactic specification that we sought in our work. However they do address the aspect of variability with an approach inspired from variability modeling in Software Product Line (SPL) approaches. They propose to

[9] some combinations of the cartesian product may be invalid

specify the configuration parameters of each language element as variation points in a variability model, along with constraints on valid combinations of variation points. This model can then be used in several ways:

(1) Configurations are enumerated manually and an automatic verification based on the variability model checks if the set of configurations is complete and reports missing configurations.

(2) Configurations are generated automatically based on the model.

In (1) the user can group configurations together into equivalence classes if the specification handles them in a similar fashion. In (2) configurations are generated systematically and the resulting set of configurations may contain a number of separate but equivalent configurations. Additionally, the user has to write a specification template with the pattern of generated code for each enumerated configuration. Therefore he would have to write a large number of such specification templates which may still be tedious. Given this factor we believe that option (1) may be a better solution as it allows to combine similar configurations.

These problems remain to be investigated as future work. We believe that an SPL-based approach combined with our syntactic specification approach would solve the issues that were raised in the experimental deployment of our proposals.

## 10.5 Conclusion

In this chapter we have detailed the experimental validation of the three main contributions of this thesis. First we validated $ATL2AGT$ with a testing approach that considers a variety of ATL test transformations and translates them to AGT. As an oracle of this testing strategy we used *second-order* testing where we compared the execution of both ATL and AGT versions of the transformation over the same input, and verified the results are identical using systematic model differencing with EMFCompare. With this strategy we confirmed that $ATL2AGT$ produces AGT transformations that are semantically equivalent to the original ATL transformations.

Then we discussed the validation of $Post2Pre$, which consisted of two parts. The first part was functional validation and aimed at showing that our implementation complies with the theoretical properties of $wlp$ and $scopedWlp$. We showed this by considering a set of small transformations and postcondition, and by manually verifying that the preconditions produced with our implementation ensure indeed the satisfaction of the postconditions. The second part was the analysis of the scalability of our implementation for larger problems and the assessment of the simplification strategies that we proposed as a way to allow scalability. First we

identified and observed the main issue which is the large size of the intermediate preconditions computed during the analysis which ultimately leads to an exhaustion of the available memory. Then we proceeded to a systematic assessment of our proposed simplification strategies in terms of the reduction in the size of intermediate preconditions and in the amount of computation required to produced them. The analysis showed that our strategies are highly efficient at eliminating irrelevant parts of preconditions early in the analysis and ultimately achieve a reduction by two orders of magnitude in the size of preconditions. This greatly reduced memory consumption and avoided large amounts of useless computations, allowing our implementation to analyse problems that were previously impossible to complete due to memory saturation.

However the fact remained that our strategies do not alter the exponential nature of the construction that was still observable on larger problems. Even though further simplification strategies targeting more critical parameters of the algorithm such as the size of the analysed graphs could be sought, we believe that a more promising research track would be the investigation of loop invariants which would short-circuit the processing of loops and perhaps avoid the exponential parts of the analysis altogether.

Finally, we discussed the experimental deployment of our last contribution, the model-to-code specification and test oracles approach, within the team developing the QGen toolset at AdaCore. The conclusions drawn from that experiment were that the core concept of specification templates that we proposed and the associated automatic test oracles and document generator are a good fit for the set of targeted problems. However we also found that deploying the approach without providing the means to ensure the completeness of the specification and the completeness of the test with respect to the coverage of that specification greatly diminishes its benefit in practice. The completeness of the specification can be accomplished with existing approaches to Simulink-based specifications based on variability modeling, while the completeness of the test set can be addressed with the integration/unit test coverage framework discussed in the first contributions of this thesis but not yet implemented in practice.

In conclusion, the validation of each of our contributions uncovered remaining open points for which we have identified promising solutions to be investigated. We believe that in the future, combining our contributions into an integrated approach would constitute a solid answer to the challenges of testing model transformation chains and would provide strong arguments regarding the coverage and exhaustiveness of testing in a qualification driven process.

# Chapter 11

# Conclusions and Future Work

## Contents

## 11.1 Summary of Contributions

This thesis was conducted on the subject matter of the qualification of Automatic Code Generators (ACGs), a topic of high interest in the avionics domain today because of the cost reduction that it can bring to the manufacture of critical airborne software. Indeed generating the source code of a critical application with a qualified ACG allows the elimination of costly verifications of the generated source code. However the qualification of ACGs remains to this day a high-priced process that imposes many constraints on the development of the ACG and requires thorough verification activities. Within this process we have focused in particular on the issues of testing and aimed at proposing efficient testing techniques that would provide the high confidence required for qualification.

The first part of the thesis focused on achieving the confidence of unit testing in Model Transformation Chains (MTCs) through integration testing which is easier to carry out. We determined this can be done with current advances by extracting unit test requirements and unit test oracles and assessing their satisfaction throughout integration testing. However once non-satisfied test requirements are identified, the existing approaches stop short of providing a way to create new integration tests targeting them. Thus the first problem tackled by this thesis was to propose an approach to the backward translation of unit test requirements into equivalent constraints on the input of the MTC that can support the creation of new integration tests.

While the first part of the thesis focused on the production of integration tests for MTCs, the second part investigated the corresponding oracles of these tests in the case of ACGs. Given the context of qualification and certification, we determined that these oracles need to be based on a syntactic specification of the generated source code, that is easily understandable by different stakeholders. Thus the second problem tackled by this thesis was to propose such a syntactic specification and test oracles approach.

The following sections summarise the solutions that we proposed to the identified problems and the main contributions of this thesis in that context.

### 11.1.1 Backward Translation of Test Requirements

To address the first problem, we proposed an iterative approach that performs the backwards translation of test requirements step by step over the model transformations of the chain. Thus in the scope of this thesis we focused on defining and

validating one step of this analysis: backward translation of constraints through one model transformation.

Our solution was based on the formal framework of Algebraic Graph Transformation (AGT) which provides the means to manipulate and reason on constraints. Based on an ATL specification of the transformation, we proposed an approach in two steps:

1. Translate the ATL transformation to an Algebraic Graph Transformation (AGT).
2. Consider the unit test requirement as a postcondition of the transformation and translate it into an equivalent precondition using the formal constructions of the *weakest liberal precondition* ($wlp$) in AGT.

With this analysis we obtain a constraint over the input of the transformation, which can again be processed by the same analysis. Iterating the approach for all transformations of the MTC would allow to reach the input language and create a new integration test that covers the non-satisfied unit test requirement.

In that approach several original contributions were put forward:

**Translation of ATL and OCL to AGT**

We defined a translation of the purely declarative subset of ATL to the formal framework of AGT. The main challenge of this work was to support the ATL resolve mechanisms which do not have an equivalent in the AGT framework. We addressed this challenge by organising the resulting AGT in 2 steps, instantiation and resolving, and relying on trace nodes to emulate the resolve mechanisms.

As for the handling of OCL, existing translations to AGT did not support ordered sets since the semantics of graph matching in AGT do not guarantee ordered matching. We addressed this limitation in the context of ATL binding queries by supplementing the existing translations with additional conditions ensuring the orderly matching of elements in ordered sets.

This work allowed us to achieve backward propagation of constraints in this thesis, and we believe it has applications beyond this scope because it enables the use of other AGT analyses on ATL transformations, as will be discussed in future work.

The translation of ATL and OCL to AGT was the subject of a publication at ICMT'15 [Richa *et al.*, 2015] which received the *Best Paper Award* of the conference.

**Translation of Postconditions to Preconditions**

In the translation of postconditions to preconditions, we found that the $wlp$

construction can be theoretically infinite for the transformations that we analyse. To address this problem we proposed to analyse a bounded version of the transformation by introducing the new construct of bounded iteration $P\downarrow_N$ and defining its corresponding finite $wlp$ construction.

Since this result is obtained with the bounded version of the transformation, it is not always valid for the unbounded original transformations. Thus we defined an alternate construction $scopedWlp$ that turns the bounded weakest liberal precondition into a non-weakest liberal precondition applicable to the original unbounded transformation.

With these proposals, we successfully translated postconditions of ATL transformations into preconditions, thus achieving the backwards translation of test requirements. Moreover, the formal concepts that we introduced were defined and proven formally for arbitrary AGT transformations. They can thus be used beyond the scope of ATL transformations.

**Simplification Strategies for $wlp$**

Finally we proposed simplification strategies to tame the computational complexity of $wlp$. Some of the strategies were applicable to arbitrary AGT transformations while others relied on the ATL semantics to eliminate irrelevant results and avoid useless computations. What was notable in that context was the possibility of introducing semantical knowledge of ATL into the construction without compromising its correctness.

With these proposals we were able to successfully perform analyses which were previously infeasible due to the high complexity.

The above contributions were validated experimentally and implemented in the form of a tool called *ATLAnalyser*[1].

### 11.1.2 Specification-based Test Oracles for Integration Tests

After focusing on the creation of test requirements in the first part of the thesis, the second part was dedicated to determining the verdict of these tests. We thus proposed a syntactic specification and test oracles approach allowing to specify and verify the concrete syntax of the generated code. At the heart of this approach is the concept of *specification templates* whose execution provides the means to validate test outputs.

The main contributions of this work were:

[1] *ATLAnalyser*, `https://github.com/elericha/atlanalyser`

**Specification Templates**

We proposed the concept of specification templates as a way to specify textual patterns in terms of verbatim test, queries to the input model, regular expressions, and repetition statements. This allows to describe the generated source code in terms of its concrete textual syntax.

**Automatic Test Oracles**

We proposed an automatic test oracles procedure based on the execution of specification templates. Given an input test model, executing specification templates provides expected patterns that are matched in the test output to determine its compliance with the specification.

**Generation of Qualification Document**

We provided a way to transform our specification into a requirements document consistent and easy to integrate with the rest of the qualification evidence.

Our approach was developed for and deployed experimentally in the specific context of QGen, the Simulink to C ACG developed at AdaCore. Nonetheless we believe the core concept of specification templates and its associated oracle procedure can be generalised to arbitrary model-to-text transformations.

## 11.2 Summary of Scope of Applicability

Having recalled the main contributions of our work, we now recall the scope of applicability of each part of our approach in the following set of tables. The symbol ✓ indicates that a feature is supported, and the symbol × indicates that a feature is not supported.

**Translation of ATL to AGT**

| | **Scope** |
|---|---|
| **Transformation Kind** | ✓ Exogenous |
| **ATL Execution Modes** | ✓ Normal mode<br>× Refining mode |
| **ATL Subset** | ✓ Purely declarative subset |
| **ATL Rules** | ✓ Matched rules<br>× Called rules<br>× Lazy rules |

| | |
|---|---|
| **Rule Blocks** | ✓ `from/to` blocks<br>× `do` blocks |
| **Bindings** | ✓ Bindings of references of all multiplicities<br>✓ Bindings of attributes of single multiplicity<br>× Bindings of attributes of non-single multiplicity |
| **Resolve Mechanisms** | ✓ Default resolve mechanism<br>✓ Non-default resolve mechanism: `resolveTemp` |
| **Other Features** | ✓ Non-recursive ATL helpers |

**Translation of OCL to NGC**

The supported OCL subset is *Essential OCL* which is detailed in [Radke *et al.*, 2015b], in addition to the support of ordered sets within ATL binding expressions. The following table gives a non-exhaustive summary of this subset.

| | **Scope** |
|---|---|
| **Basic Types** | ✓ Predefined types `Integer`, `Real`, `String`, `Boolean`<br>✓ Metamodel-defined classes |
| **Collection Types** | ✓ `Set`<br>✓ `OrderedSet` within ATL binding expressions<br>× `Sequence` and `Bag` |
| **Navigation** | ✓ References of all multiplicities<br>✓ Attributes of single multiplicity<br>× Attributes of non-single multiplicities |
| **Collection Operations** | ✓ `size`, `isEmpty`, `notEmpty`, `includes`, `excludes`, `includesAll`, `excludesAll`, `union`, `intersection`, `-`, `symmetricDifference`, `including`, `excluding`, `exists`, `forAll`, `select`, `reject`, `collect` |
| **Logic** | ✓ 2-valued First Order Logic<br>× `null` value |
| **Ordering Features** | ✓ Order preservation in ATL bindings<br>✓ `at` operation after reference navigation |
| **Other Features** | ✓ `if-then-else` expressions |

**Translation of Postconditions to Preconditions**

| | **Scope** |
|---|---|
| **Transformations** | Purely structural transformations:<br>✓ Node (object) and edge (reference) manipulation<br>× Attribute manipulation |
| **Post/Pre-conditions** | ✓ Full Nested Graph Conditions |

**Simplification Strategies for Postcondition to Precondition Translation**

| | **Scope** |
|---|---|
| **Standard NGC and Logic Properties** | ✓ All transformations |
| **Rule Selection** | ✓ All transformations |
| **ATL Trace Node Semantics** | ✓ ATL transformations |
| **Element Creation** | ✓ Exogenous transformations |
| $Post2Left$ **Combined Construction** | ✓ All transformations |

**Specification and Test Oracles for Model-to-Text Transformations**

| | **Scope** |
|---|---|
| **Transformations** | ✓ Model-to-text transformations |
| **Expressiveness** | ✓ Simple positive constraints:<br>$condition \Rightarrow \exists\,(textual\ pattern)$<br>× Negation, conjunction, disjunction of textual patterns<br>× Specifying location of textual pattern within generated artifacts |

## 11.3 Limitations and Future Work

The solutions that we proposed address for the large part the issues raised in this thesis, however they suffer from several limitations that we develop next with possible future improvements.

**Translation of ATL to AGT**

Even though the translation that we proposed is currently limited to purely declarative ATL transformations, we do not see a major obstacle in extending it to

imperative features of ATL such as do-blocks and lazy rules. Support for these features can be provided by exploiting to a larger extent the imperative features of sequencing and iteration in AGT. This would probably require enriching trace nodes with additional information to support the new features. Moreover, in the translation of OCL, the solution that we proposed for ordered sets can easily be extended to support arbitrary order relationships specified with the operation `sortedBy()`.

However we acknowledge that supporting the full semantics of OCL would be very difficult, if not impossible to achieve in AGT. For example advanced OCL features such as `iterate()` and `closure()` operations, and corner cases of the semantics such as exception elements `invalid` and `null` would be very hard to emulate in AGT. In fact the latter two elements make of OCL a four-valued logic [Brucker *et al.*, 2014] which would be difficult to emulate in the two-valued logic of NGC. However the necessity of such advanced notions in the highly constrained development environment of qualified tools is still an open discussion point.

**Translation of Postconditions to Preconditions**

In the translation of postconditions to preconditions, a first conceptual limitation of our approach is the fact that it requires a bound $N$. We also did not provide a way to determine a relevant value for this bound. We believe that a first solution would be to provide heuristics based for example on the number of elements of each type in the postcondition and the number of instances of these types created by each rule of the transformation. Additionally the bound $N$ for all iterations could be replaced by a set of bounds $N_i$, one for each iterated rule, to provide finer scoping of the analysis.

However we believe that a more promising research track is the investigation of *loop invariants* which would eliminate the need for the bound $N$ completely. Invariants have been investigated in AGT and shown to be difficult to obtain automatically for arbitrary programs [Pennemann, 2009], which is why other approaches require users to provide them manually for loops [Poskitt, 2013]. However, research on classical programing languages like C, Ada and Java, indicates that for programs with particular properties, loop invariants can be inferred automatically [Furia and Meyer, 2010]. In our analysis of ATL transformations translated to AGT, loops have a specific form and role, which suggests that it may be possible in the future to infer their invariants automatically.

From an implementation perspective, another limitation of our approach is its current inability to scale for real-size use cases and complete code generation chains. Indeed despite the efficiency of our simplification strategies demonstrated in the experimental validation, real-size transformations and postconditions could not be

processed by our prototype. We believe that loop invariants could also be the key to enabling the scalability of our analysis as they avoid the recursive application of the analysis to loops.

#### Syntactic Model-to-code Specification and Test Oracles

A first limitation of our syntactic specification approach is its limited expressiveness. The approach was designed specifically for the needs of QGen and was sufficient for the large part. However we believe that extending the specification language and its semantics with more constructs (*e.g.* the creation of files, the conjunction/disjunction of specification templates *etc.*) would allow generalising the approach to arbitrary model-to-text transformations.

Another limitation of our approach was that when applied to Simulink, it required the manual enumeration of a large number of configurations. This proved to be a tedious and error prone operation, with no guarantee regarding the completeness of the resulting set of configurations. In that context we have identified relevant work in the literature [Dieumegard *et al.*, 2012; Dieumegard *et al.*, 2014b; Dieumegard *et al.*, 2014a] that can help ensure completeness by modeling the variability of Simulink elements with appropriate formalism such as *feature models*. This work could also be the basis for an interactive specification approach where the user is assisted by the tool in the identification of configurations, making the specification less tedious and less error prone.

## 11.4 Long-term Perspectives

Long-term perspectives of the research conducted in this thesis include applications of its contributions beyond the original scope, as well as larger research questions in the industry of safety-critical systems.

#### AGT Analyses and Formal Proof of Correctness for ATL

Our translation of ATL to AGT enables the application of AGT-based analyses other than the translation of postconditions that we applied. For example critical pair analysis [Ehrig *et al.*, 2012b] could allow to statically detect conflicts between ATL rules, which correspond in fact to errors in the specification.

Additionally, the weakest precondition construction that we developed in the context of testing model transformation chains can be used for its original purpose: conducting the formal proof of correctness of programs [Dijkstra, 1975], *i.e.* proving that under the assumption of a precondition, a program always guarantees its

postcondition. This method has been successfully applied to AGT programs [Pennemann, 2009; Habel and Pennemann, 2009; Poskitt, 2013]. Therefore our work on translating ATL to AGT and implementing weakest precondition construction enables the formal proof of correctness of ATL transformations.

Additionally, in [Pennemann, 2009] theorem provers based on AGT were shown to be more efficient than classical first-order logic theorem provers. This suggests that the AGT-based proof of correctness of ATL transformations and model transformations in general would be more efficient than existing correctness approaches which are based on first-order logic theorem provers [Büttner *et al.*, 2012a].

However it should be noted that AGT-based theorem provers are still at an early stage of development. The prover used in [Pennemann, 2009] did not support scalar object attributes such as integers and strings which is a strong limitation. Therefore even though on the long run AGT-based provers would be efficient, they must support a wider range of semantics before they can be realistically applied to model transformations.

**Safety-Critical Product Lines**

In this thesis we investigated the problem of qualifying *one* development tool. However tool providers like AdaCore have multiple customers, each with different needs requiring slightly different versions of the same general functionality. For this kind of situations, software engineering advances have proposed Software Product Line (SPL) engineering [Pohl *et al.*, 2005] as a paradigm to develop a set of software products sharing common features, with reduced development cost and increased quality.

The application of SPL engineering in a safety-critical context regulated by strict standards raises difficult questions [Hutchesson and McDermid, 2013]. Moreover, in this thesis we only discussed the standards DO−178C and DO−330 of the avionics domain. The automotive and railway domains also have their own safety standards that bear many similarities with the standards that we studied. That factor could also be included as an aspect of variability in the product line, allowing a tool provider to provide qualified tools not only with varying functionality, but also targeting different safety standards.

# Appendices

# Appendix A

# Examples of Weakest Liberal Preconditions

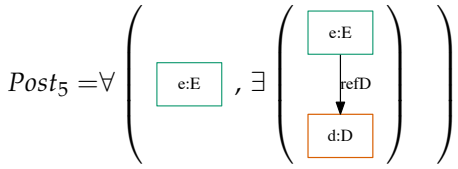
## Contents

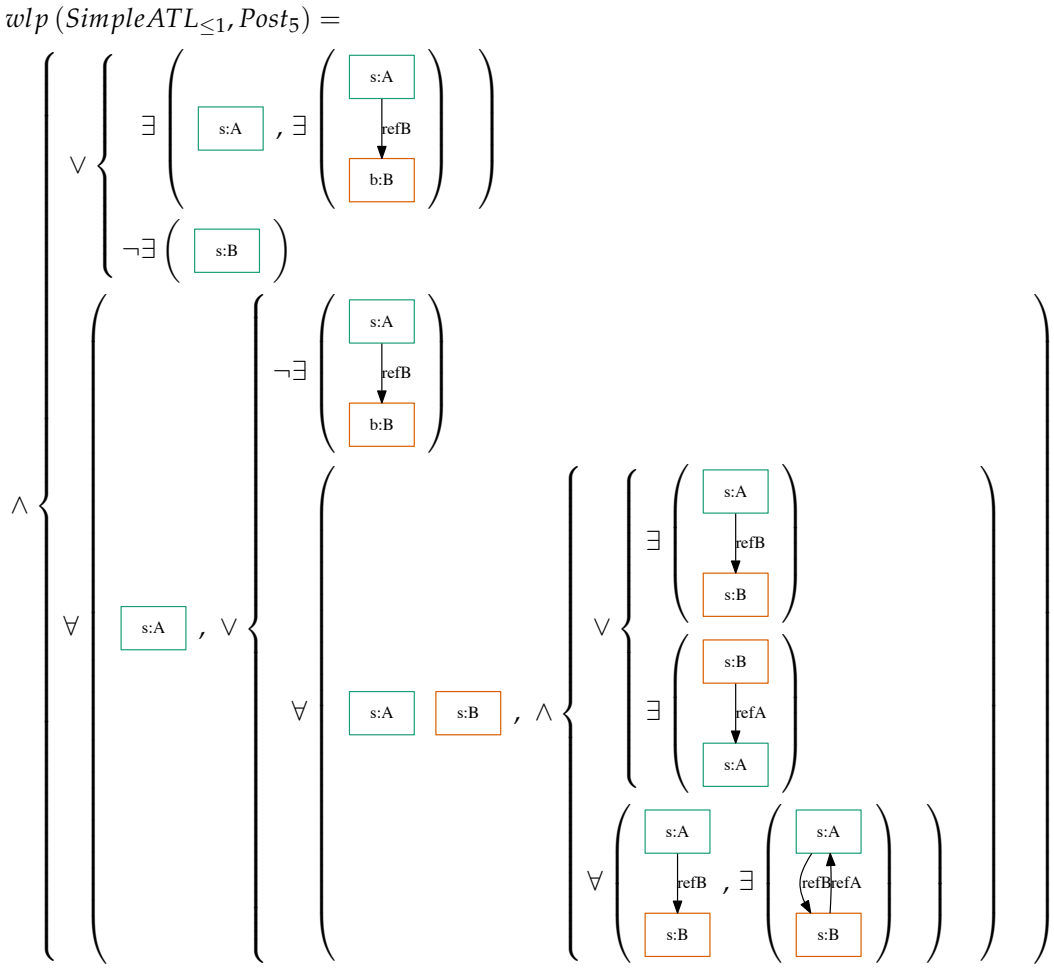
## A.1 Reminder of the ATL Transformation

The ATL transformation *SimpleATL* used in examples 1 to 3 is the following:

```
1  rule R1 {
2  from s  : IN!A
3     (s.refB->exists(b | true))
4  to t1 : OUT!D
5     (refD <- t2,
6      refE <- s.refB),
7   t2 : OUT!D }
8
9  rule R2 {
10 from s : IN!B
11 to t : OUT!E
12    (refD <- thisModule.resolveTemp
13                  (s.refA, 't2') ) }
```

## A.2 Example 1




The precondition requires that:

(1) either there exist no objects of type *B*

(2) or that for all objects of type *B*, there is an object of type *A* connected to it with references *refA* and *refB*

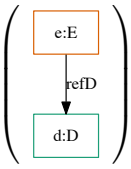
(1) means that **R2** is never triggerred, and there are no object *E* in the output model. This means that the ∀ statement of the postcondition is satisfied vacuously.

(2) means that for all objects *B* triggerring **`R2`**, there is a corresponding trigger of **`R1`** (because of the existence of object *A refB*) and the non-default resolve can execute (because of *refA*) yielding reference *refD*. Therefore this ensure the satisfaction of the postcondition.

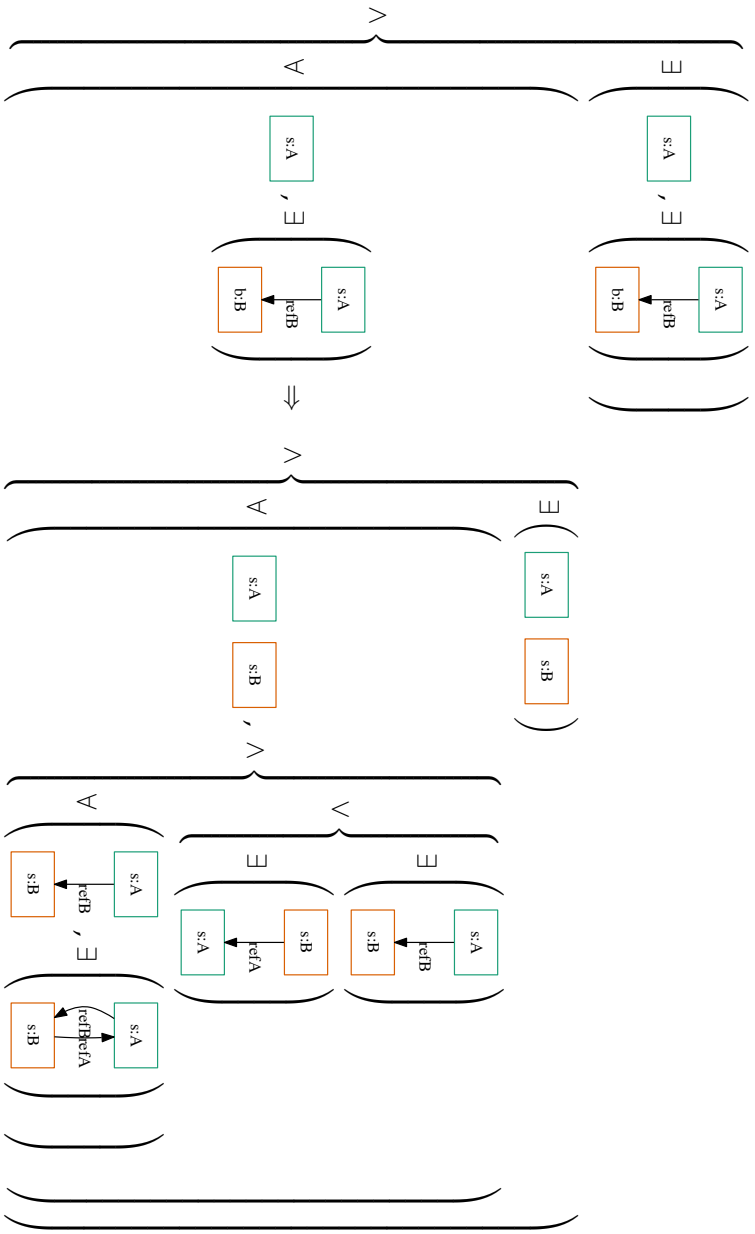
Given the above, the precondition ensures the satisfaction of the postcondition.

## A.3 Example 2

$Post_6 = \exists$




$wlp\,(SimpleATL_{\leq 1}, Post_6) =$



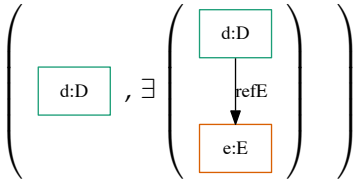

The precondition states that:

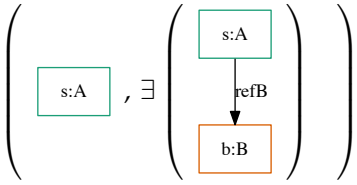
(1) There exists an object $A$ such that the application condition of **R1** is satisfied (there is a reference *refB*)

(2) and for all objects $A$, there exists an object $B$ that is connected to $A$ with *refA*

The combination of both conditions ensures that **R1** and **R2** are both triggered, and the non-default resolve can be performed thanks to the existence of *refA*. This ensures that objects $D$ and $E$ are created and connected with *refD* due to the non-default resolve. Hence the precondition ensure the satisfaction of the postcondition.

## A.4 Example 3

$Post_8 = \forall$ 


$wlp\,(SimpleATL_{\leq 1}, Post_8) = \neg\exists$ 


Rule **R1** always produces two objects of type *D*:

1. The first object is connected to an instance of *E* through *refE* thanks to the default resolve
2. The second object is not connected to an instance of *E*

Since the second object violates the postcondition, then the only way to ensure the satisfaction of the postcondition is with a precondition that prohibits the execution of rule **R1**. Indeed the precondition states that there are no objects *A* that have a *refB* which ensures that **R1** can never trigger and that the postcondition is always satisfied.

# Bibliography

[Acceleo, accessed 2015] Acceleo. Acceleo Model-to-Text Transformation Framework. `http://www.eclipse.org/acceleo`, accessed 2015.

[AdaCore, accessed 2015] AdaCore. QGen, a qualifiable code generator from Simulink to MISRA C and SPARK/Ada. `http://www.adacore.com/qgen`, accessed 2015.

[AGG, accessed 2015] AGG. The Attributed Graph Grammar (AGG) development environment. `http://user.cs.tu-berlin.de/~gragra/agg/index.html`, accessed 2015.

[Alloy, accessed 2015] Alloy. Alloy: a language & tool for relational models. `http://alloy.mit.edu/`, accessed 2015.

[Anastasakis *et al.*, 2007a] Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. UML2Alloy: A challenging model transformation. In Gregor Engels, Bill Opdyke, DouglasC. Schmidt, and Frank Weil, editors, *Model Driven Engineering Languages and Systems*, volume 4735 of *Lecture Notes in Computer Science*, pages 436–450. Springer Berlin Heidelberg, 2007.

[Anastasakis *et al.*, 2007b] Kyriakos Anastasakis, Behzad Bordbar, and Jochen M Küster. Analysis of model transformations via alloy. In *Proceedings of the 4th MoDeVVa workshop Model-Driven Engineering, Verification and Validation*, pages 47–56, 2007.

[Anastasakis *et al.*, 2010] Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. On challenges of model transformation from UML to Alloy. *Software & Systems Modeling*, 9(1):69–86, 2010.

[Aranega *et al.*, 2014] Vincent Aranega, Jean-Marie Mottu, Anne Etien, Thomas Degueule, Benoit Baudry, and Jean-Luc Dekeyser. Towards an Automation of the Mutation Analysis Dedicated to Model Transformation. *Software Testing, Verification and Reliability*, pages 0–0, April 2014.

[Arendt *et al.*, 2014] Thorsten Arendt, Annegret Habel, Hendrik Radke, and Gabriele Taentzer. From core OCL invariants to nested graph constraints. In Holger Giese and Barbara König, editors, *Graph Transformation*, volume 8571 of *Lecture Notes in Computer Science*, pages 97–112. Springer International Publishing, 2014.

[ATL Zoo, accessed 2015] ATL Zoo. ATL Transformation Zoo. `http://www.eclipse.org/atl/atlTransformations/`, accessed 2015.

[Baudry *et al.*, 2010] Benoit Baudry, Sudipto Ghosh, Franck Fleurey, Robert France, Yves Le Traon, and Jean-Marie Mottu. Barriers to systematic model transformation testing. *Commun. ACM*, 53(6):139–143, June 2010.

[Bauer *et al.*, 2011] Eduard Bauer, JochenM. Küster, and Gregor Engels. Test suite quality for model transformation chains. In Judith Bishop and Antonio Vallecillo, editors, *Objects, Models, Components, Patterns*, volume 6705 of *Lecture Notes in Computer Science*, pages 3–19. Springer Berlin Heidelberg, 2011.

[Bergmann, 2014] Gábor Bergmann. Translating ocl to graph patterns. In Juergen Dingel, Wolfram Schulte, Isidro Ramos, Silvia Abrahão, and Emilio Insfran, editors, *Model-Driven Engineering Languages and Systems*, volume 8767 of *Lecture Notes in Computer Science*, pages 670–686. Springer International Publishing, 2014.

[Bézivin *et al.*, 2006] Jean Bézivin, Fabian Büttner, Martin Gogolla, Frederic Jouault, Ivan Kurtev, and Arne Lindow. Model transformations? transformation models! In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *Model Driven Engineering Languages and Systems*, volume 4199 of *Lecture Notes in Computer Science*, pages 440–453. Springer Berlin Heidelberg, 2006.

[Biermann *et al.*, 2012] Enrico Biermann, Claudia Ermel, and Gabriele Taentzer. Formal foundation of consistent EMF model transformations by algebraic graph transformation. *Software & Systems Modeling*, 11(2):227–250, 2012.

[Brottier *et al.*, 2006] E. Brottier, F. Fleurey, J. Steel, B. Baudry, and Y. Le Traon. Metamodel-based test generation for model transformations: an algorithm and a tool. In *Software Reliability Engineering, 2006. ISSRE '06. 17th International Symposium on*, pages 85 –94, nov. 2006.

[Brucker and Wolff, 2008] AchimD. Brucker and Burkhart Wolff. Hol-ocl: A formal proof environment for uml/ocl. In JoséLuiz Fiadeiro and Paola Inverardi, editors, *Fundamental Approaches to Software Engineering*, volume 4961 of *Lecture Notes in Computer Science*, pages 97–100. Springer Berlin Heidelberg, 2008.

[Brucker *et al.*, 2014] Achim D. Brucker, Frédéric Tuong, and Burkhart Wolff. Featherweight OCL: A proposal for a machine-checked formal semantics for OCL 2.5. *Archive of Formal Proofs*, January 2014. `http://afp.sf.net/entries/Featherweight_OCL.shtml`, Formal proof development.

[Büttner *et al.*, 2011] Fabian Büttner, Jordi Cabot, and Martin Gogolla. On validation of ATL transformation rules by transformation models. In *Proceedings of the 8th International Workshop on Model-Driven Engineering, Verification and Validation*, MoDeVVa, pages 9:1–9:8, New York, NY, USA, 2011. ACM.

[Büttner *et al.*, 2012a] Fabian Büttner, Marina Egea, and Jordi Cabot. On verifying atl transformations using 'off-the-shelf' SMT solvers. In RobertB. France, Jürgen Kazmeier, Ruth Breu, and Colin Atkinson, editors, *Model Driven Engineering Languages and Systems*, volume 7590 of *Lecture Notes in Computer Science*, pages 432–448. Springer Berlin Heidelberg, 2012.

[Büttner *et al.*, 2012b] Fabian Büttner, Marina Egea, Jordi Cabot, and Martin Gogolla. Verification of ATL transformations using transformation models and model finders. In Toshiaki Aoki and Kenji Taguchi, editors, *Formal Methods and Software Engineering*, volume 7635 of *Lecture Notes in Computer Science*, pages 198–213. Springer Berlin Heidelberg, 2012.

[Cabot *et al.*, 2010a] Jordi Cabot, Robert Clarisó, Esther Guerra, and Juan de Lara. Synthesis of OCL pre-conditions for graph transformation rules. In Laurence Tratt and Martin Gogolla, editors, *Theory and Practice of Model Transformations*, volume 6142 of *Lecture Notes in Computer Science*, pages 45–60. Springer Berlin Heidelberg, 2010.

[Cabot *et al.*, 2010b] Jordi Cabot, Robert Clarisó, Esther Guerra, and Juan de Lara. Verification and validation of declarative model-to-model transformations through invariants. *Journal of Systems and Software*, 83(2):283 – 302, 2010. Computer Software and Applications.

[Cadoret *et al.*, 2012] Fabien Cadoret, Etienne Borde, Sébastien Gardoll, and Laurent Pautet. Design patterns for rule-based refinement of safety critical embedded systems models. In *17th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS 2012, Paris, France, July 18-20, 2012*, pages 67–76, 2012.

[Cariou *et al.*, 2004] Eric Cariou, Raphaël Marvie, Lionel Seinturier, and Laurence Duchien. OCL for the specification of model transformation contracts. In *in Proceedings of Workshop OCL and Model Driven Engineering*, 2004.

[Cariou *et al.*, 2009] Eric Cariou, Nicolas Belloir, Franck Barbier, and Nidal Djemam. OCL contracts for the verification of model transformations. *OCL workshop of MoDELS*, 2009.

[Deckwerth and Varró, 2014] Frederik Deckwerth and Gergely Varró. Attribute handling for generating preconditions from graph constraints. In Holger Giese and Barbara König, editors, *Graph Transformation*, volume 8571 of *Lecture Notes in Computer Science*, pages 81–96. Springer International Publishing, 2014.

[Dieumegard *et al.*, 2012] Arnaud Dieumegard, Andres Toom, and Marc Pantel. Model-based formal specification of a DSL library for a qualified code generator. In *Proceedings of the 12th Workshop on OCL and Textual Modelling, Innsbruck, Austria, September 30, 2012*, pages 61–62, 2012.

[Dieumegard *et al.*, 2014a] Arnaud Dieumegard, Andres Toom, and Marc Pantel. Formal specification of block libraries in dataflow languages. In *ERTS*$^2$ *2014*, 2014.

[Dieumegard *et al.*, 2014b] Arnaud Dieumegard, Andres Toom, and Marc Pantel. A software product line approach for semantic specification of block libraries in dataflow languages. In *18th International Software Product Line Conference, SPLC '14, Florence, Italy, September 15-19, 2014*, pages 217–226, 2014.

[Dijkstra, 1975] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, August 1975.

[ECL, accessed 2015] ECL. ECL$^i$PS$^e$ constraint programming system. `http://eclipseclp.org`, accessed 2015.

[Ehrig *et al.*, 2006] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of algebraic graph transformation*, volume 373. Springer, 2006.

[Ehrig *et al.*, 2012a] Hartmut Ehrig, Ulrike Golas, Annegret Habel, Leen Lambers, and Fernando Orejas. M-adhesive transformation systems with nested application conditions. part 1: Parallelism, concurrency and amalgamation. 2012.

[Ehrig *et al.*, 2012b] Hartmut Ehrig, Ulrike Golas, Annegret Habel, Leen Lambers, and Fernando Orejas. M-adhesive transformation systems with nested application conditions. part 2: Embedding, critical pairs and local confluence. *Fundamenta Informaticae*, 118(1):35–63, 2012.

[EMFTVM, accessed 2015] EMFTVM. ATL EMF Transformation Virtual Machine (research VM). `http://wiki.eclipse.org/ATL/EMFTVM`, accessed 2015.

[Fleurey *et al.*, 2004] F. Fleurey, J. Steel, and B. Baudry. Validation in model-driven engineering: testing model transformations. In *Model, Design and Validation, 2004. Proceedings. 2004 First International Workshop on*, pages 29 – 40, nov. 2004.

[Fleurey *et al.*, 2007] Franck Fleurey, Benoit Baudry, Pierre-Alain Muller, and Yves Le Traon. Towards Dependable Model Transformations: Qualifying Input Test Data. *Journal of Software and Systems Modeling (SoSyM)*, 2007.

[Furia and Meyer, 2010] CarloAlberto Furia and Bertrand Meyer. Inferring loop invariants using postconditions. In Andreas Blass, Nachum Dershowitz, and Wolfgang Reisig, editors, *Fields of Logic and Computation*, volume 6300 of *Lecture Notes in Computer Science*, pages 277–300. Springer Berlin Heidelberg, 2010.

[Gogolla *et al.*, 2005] Martin Gogolla, Jørn Bohling, and Mark Richters. Validating uml and ocl models in use by automatic snapshot generation. *Software & Systems Modeling*, 4(4):386–398, 2005.

[González and Cabot, 2012] CarlosA. González and Jordi Cabot. ATLTest: A white-box test generation approach for ATL transformations. In RobertB. France, Jürgen Kazmeier, Ruth Breu, and Colin Atkinson, editors, *Model Driven Engineering Languages and Systems*, volume 7590 of *Lecture Notes in Computer Science*, pages 449–464. Springer Berlin Heidelberg, 2012.

[González and Cabot, 2014] CarlosA. González and Jordi Cabot. Test data generation for model transformations combining partition and constraint analysis. In Davide Di Ruscio and Dániel Varró, editors, *Theory and Practice of Model Transformations*, volume 8568 of *Lecture Notes in Computer Science*, pages 25–41. Springer International Publishing, 2014.

[Gonzalez *et al.*, 2012] C.A. Gonzalez, F. Buttner, R. Clariso, and J. Cabot. EMFtoCSP: A tool for the lightweight verification of EMF models. In *Software Engineering: Rigorous and Agile Approaches (FormSERA), 2012 Formal Methods in*, pages 44–50, June 2012.

[Guerra and Soeken, 2015] Esther Guerra and Mathias Soeken. Specification-driven model transformation testing. *Software & Systems Modeling*, 14(2):623–644, 2015.

[Guerra *et al.*, 2010] E. Guerra, J. de Lara, D. Kolovos, and R. Paige. A visual specification language for model-to-model transformations. In *Visual Languages and Human-Centric Computing (VL/HCC), 2010 IEEE Symposium on*, pages 119–126, 2010.

[Guerra *et al.*, 2013] Esther Guerra, Juan Lara, Manuel Wimmer, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Johannes Schönböck, and Wieland

Schwinger. Automated verification of model transformations based on visual contracts. *Automated Software Engineering*, 20(1):5–46, 2013.

[Guerra, 2012] Esther Guerra. Specification-driven test generation for model transformations. In Zhenjiang Hu and Juan Lara, editors, *Theory and Practice of Model Transformations*, volume 7307 of *Lecture Notes in Computer Science*, pages 40–55. Springer Berlin Heidelberg, 2012.

[Habel and Pennemann, 2005] Annegret Habel and Karl-Heinz Pennemann. Nested constraints and application conditions for high-level structures. In Hans-Jörg Kreowski, Ugo Montanari, Fernando Orejas, Grzegorz Rozenberg, and Gabriele Taentzer, editors, *Formal Methods in Software and Systems Modeling*, volume 3393 of *Lecture Notes in Computer Science*, pages 293–308. Springer Berlin Heidelberg, 2005.

[Habel and Pennemann, 2009] Annegret Habel and Karl-heinz Pennemann. Correctness of high-level transformation systems relative to nested conditions. *Mathematical. Structures in Comp. Sci.*, 19(2):245–296, April 2009.

[Habel *et al.*, 2006a] Annegret Habel, Karl-Heinz Pennemann, and Arend Rensink. Weakest preconditions for high-level programs. In Andrea Corradini, Hartmut Ehrig, Ugo Montanari, Leila Ribeiro, and Grzegorz Rozenberg, editors, *Graph Transformations*, volume 4178 of *Lecture Notes in Computer Science*, pages 445–460. Springer Berlin Heidelberg, 2006.

[Habel *et al.*, 2006b] Annegret Habel, Karl-Heinz Pennemann, and Arend Rensink. Weakest preconditions for high-level programs: Long version. Technical Report 8/06, University of Oldenburg, 2006.

[Hayhurst *et al.*, 2001] Kelly J Hayhurst, Dan S Veerhusen, John J Chilenski, and Leanna K Rierson. *A practical tutorial on modified condition/decision coverage*. National Aeronautics and Space Administration (NASA), Langley Research Center, 2001.

[Henshin, accessed 2015] Henshin. The Henshin project. `http://www.eclipse.org/henshin`, accessed 2015.

[Hoare, 1969] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[Hutchesson and McDermid, 2013] Stuart Hutchesson and John McDermid. Trusted product lines. *Information and Software Technology*, 55(3):525 – 540, March 2013.

[Jouault and Kurtev, 2006] Frédéric Jouault and Ivan Kurtev. Transforming models with ATL. In Jean-Michel Bruel, editor, *Satellite Events at the MoDELS 2005 Con-*

*ference*, volume 3844 of *Lecture Notes in Computer Science*, pages 128–138. Springer Berlin Heidelberg, 2006.

[Jouault *et al.*, 2008] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL: A model transformation tool. *Science of Computer Programming*, 72(1–2):31 – 39, 2008. Special Issue on Second issue of experimental software and toolkits (EST).

[Leveque *et al.*, 2011] Thomas Leveque, Jan Carlson, Séverine Sentilles, and Etienne Borde. Flexible semantic-preserving flattening of hierarchical component models. In *37th EUROMICRO Conference on Software Engineering and Advanced Applications, SEAA 2011, Oulu, Finland, August 30 - September 2, 2011*, pages 31–38, 2011.

[Mottu *et al.*, 2006] Jean-Marie Mottu, Benoit Baudry, and Yves Traon. Mutation analysis testing for model transformations. In Arend Rensink and Jos Warmer, editors, *Model Driven Architecture – Foundations and Applications*, volume 4066 of *Lecture Notes in Computer Science*, pages 376–390. Springer Berlin Heidelberg, 2006.

[Mottu *et al.*, 2008] J.-M. Mottu, B. Baudry, and Y. Le Traon. Model transformation testing: oracle issue. In *Software Testing Verification and Validation Workshop, 2008. ICSTW '08. IEEE International Conference on*, pages 105–112, 2008.

[Mottu *et al.*, 2012] J.-M. Mottu, S. Sen, M. Tisi, and J. Cabot. Static analysis of model transformations for effective test generation. In *Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on*, pages 291–300, 2012.

[Myers *et al.*, 2011] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011.

[OMG, 2014] OMG. Object Constraint Language (OCL). `http://www.omg.org/spec/OCL`, 2014.

[Ostrand and Balcer, 1988] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating fuctional tests. *Commun. ACM*, 31(6):676–686, June 1988.

[Pennemann, 2009] Karl-Heinz Pennemann. *Development of Correct Graph Transformation Systems*. PhD thesis, Universität Oldenburg, 2009.

[Pohl *et al.*, 2005] Klaus Pohl, Günter Böckle, and Frank Van Der Linden. *Software product line engineering: foundations, principles, and techniques*. Springer, 2005.

[Poskitt and Plump, 2013] Christopher M. Poskitt and Detlef Plump. Verifying total correctness of graph programs. In *Revised Selected Papers, Graph Computation Models (GCM 2012)*, Electronic Communications of the ECEASST 61. 2013.

[Poskitt *et al.*, 2014] Christopher M Poskitt, Mike Dodds, Richard F Paige, and Arend Rensink. Towards rigorously faking bidirectional model transformations. In *AMT 2014–Analysis of Model Transformations Workshop Proceedings*, page 70, 2014.

[Poskitt, 2013] Christopher M. Poskitt. *Verification of Graph Programs*. PhD thesis, University of York, 2013.

[Radke *et al.*, 2015a] Hendrik Radke, Thorsten Arendt, Jan Steffen Becker, Annegret Habel, and Gabriele Taentzer. Translating essential OCL invariants to nested graph constraints focusing on set operations: Long version. *ICGT'15*, 2015.

[Radke *et al.*, 2015b] Hendrik Radke, Thorsten Arendt, JanSteffen Becker, Annegret Habel, and Gabriele Taentzer. Translating essential ocl invariants to nested graph constraints focusing on set operations. In Francesco Parisi-Presicce and Bernhard Westfechtel, editors, *Graph Transformation*, volume 9151 of *Lecture Notes in Computer Science*, pages 155–170. Springer International Publishing, 2015.

[Richa *et al.*, 2014] Elie Richa, Etienne Borde, Laurent Pautet, Matteo Bordin, and Jose F. Ruiz. Towards testing model transformation chains using precondition construction in algebraic graph transformation. In *Third Workshop on the Analysis of Model Transformations, AMT'14*, 2014.

[Richa *et al.*, 2015] Elie Richa, Etienne Borde, and Laurent Pautet. Translating ATL model transformations to algebraic graph transformations. In *Proceedings of the 8th international conference on Model Transformation*, ICMT'15. Springer, 2015.

[Sen *et al.*, 2008] S. Sen, B. Baudry, and J.-M. Mottu. On combining multi-formalism knowledge to select models for model transformation testing. In *Software Testing, Verification, and Validation, 2008 1st International Conference on*, pages 328–337, April 2008.

[Sen *et al.*, 2009] Sagar Sen, Benoit Baudry, and Jean-Marie Mottu. Automatic model generation strategies for model transformation testing. In RichardF. Paige, editor, *Theory and Practice of Model Transformations*, volume 5563 of *Lecture Notes in Computer Science*, pages 148–164. Springer Berlin Heidelberg, 2009.

[Sen *et al.*, 2012] Sagar Sen, Jean-Marie Mottu, Massimo Tisi, and Jordi Cabot. Using models of partial knowledge to test model transformations. In Zhenjiang Hu and Juan Lara, editors, *Theory and Practice of Model Transformations*, volume 7307

of *Lecture Notes in Computer Science*, pages 24–39. Springer Berlin Heidelberg, 2012.

[Soeken *et al.*, 2010] Mathias Soeken, Robert Wille, Mirco Kuhlmann, Martin Gogolla, and Rolf Drechsler. Verifying UML/OCL models using boolean satisfiability. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 1341–1344, 3001 Leuven, Belgium, Belgium, 2010. European Design and Automation Association.

[Stevens, 2008] Perdita Stevens. A landscape of bidirectional model transformations. In Ralf Lämmel, Joost Visser, and João Saraiva, editors, *Generative and Transformational Techniques in Software Engineering II*, volume 5235 of *Lecture Notes in Computer Science*, pages 408–424. Springer Berlin Heidelberg, 2008.

[Stuermer *et al.*, 2007] Ingo Stuermer, M. Conrad, Heiko Doerr, and P. Pepper. Systematic testing of model-based code generators. *Software Engineering, IEEE Transactions on*, 33(9):622–634, Sept 2007.

[Sturmer and Conrad, 2003] I. Sturmer and M. Conrad. Test suite design for code generation tools. In *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, pages 286–290, 2003.

[Troya and Vallecillo, 2011] Javier Troya and Antonio Vallecillo. A rewriting logic semantics for ATL. *Journal of Object Technology*, 10(5):1–29, 2011.

[Wimmer and Burgueño, 2013] Manuel Wimmer and Loli Burgueño. Testing M2T/T2M transformations. In Ana Moreira, Bernhard Schätz, Jeff Gray, Antonio Vallecillo, and Peter Clarke, editors, *Model-Driven Engineering Languages and Systems*, volume 8107 of *Lecture Notes in Computer Science*, pages 203–219. Springer Berlin Heidelberg, 2013.

[Zhu *et al.*, 1997] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, December 1997.