Numéro d'ordre : 40678

Université des Sciences et Technologies de Lille
École doctorale : Sciences Pour l'Ingénieur

THÈSE

présentée pour obtenir le titre de docteur
spécialité Informatique

par

VINCENT ARANEGA

TRAÇABILITÉ POUR LA MISE AU POINT DE MODÈLES ET LA CORRECTION DE TRANSFORMATIONS

Thèse soutenue le 28 novembre 2011, devant le jury formé de :

Lionel Seinturier	Professeur Université Lille 1	Président
Benoît Baudry	Chercheur INRIA	Rapporteur
Éric Senn	Maître de conférences Université de Bretagne Sud	Rapporteur
Frédéric Robert	Professeur Université Libre de Bruxelles	Examinateur
Jean-Luc Dekeyser	Professeur Université Lille 1	Directeur
Anne Etien	Maître de conférences Université Lille 1	Co-directrice

Université des Sciences et Technologies de Lille
LIFL - UMR 8022 - Cité Scientifique, Bât. M3 - 59655 Villeneuve d'Ascq Cedex

TRAÇABILITÉ POUR LA MISE AU POINT DE MODÈLES ET LA CORRECTION DE TRANSFORMATIONS

VINCENT ARANEGA

Thèse de doctorat

28 novembre 2011

Vincent Aranega : *Traçabilité pour la mise au point de modèles et la correction de transformations,* Thèse de doctorat, © 28 novembre 2011

ABSTRACT

Model Driven Engineering (MDE) enable the control of the increasing complexity of software and hardware architectures. Indeed, the MDE allows, among others, to represent these architectures in an abstract way and then, through successive transformations, to generate the corresponding source code. To achieve this goal, Gaspard2 was created as a co-design environment of embedded systems. The Gaspard2 environment is based on transformations chains for the automatic generation of code from a MARTE model.

In the context of Gaspard2 as well as more generally in transformation chains, the generated software execution does not always correspond, in terms of performance or behavior, to the one expected by the model designer. Therefore, it is necessary to provide the model designer with a high-level model debugging and optimizing environment closely linked to the performance of generated code.

In this thesis, we provide a model debugging and optimization solution, based on model to model traceability and automatic generated source code instrumentation, regardless of the used transformation language. The solution we propose allows the designers to specify in the high-level models the information to see during the generated software execution. Subsequently, the observations are brought back in the high-level models.

The model debugging and optimizing are based on a strong hypothesis : the transformations chain, which enable the generation, is considered trustworthy. However, during the development of chain, errors can be introduced into the transformations, generating errors in the software source code. Model transformation testing is necessary to obtain trustworthy transformations chain.

In this thesis, we have also worked on the use of our trace to help in the errors localization in model transformations and in transformations chain and for the mutation analysis process. In the context of the errors localization, we used the trace support to refine the model designers search field. That way, it avoids the concepters to find fault in the whole transformation. In the context of mutation analysis, we propose an approach based on our trace mechanism and an abstraction of mutation operators to assist testers in some stages of the mutation analysis which remain manual.

The works presented in this thesis (for the models *debugging* and optimization and those about the transformation test) were implemented in the Gaspard2 environment. As our trace mechanism is independent of any transformation language, this work is reusable for other transformation chain.

RÉSUMÉ

L'ingénierie dirigée par les modèles (IDM) représente une approche aujourd'hui reconnue pour maîtriser la complexité croissante des architectures logicielles et matérielles. En effet, l'IDM permet, entre autre, de représenter ces architectures de façon abstraite puis, grâce à des transformations successives, de générer le code source correspondant. Dans cette optique, Gaspard2 a été conçu comme un environnement de co-design de systèmes embarqués reposant sur une chaîne de transformations permettant la génération automatique de code à partir d'un modèle MARTE.

Dans le contexte de Gaspard2 et plus généralement, des chaînes de transformations proposant une génération automatique de code, l'exécution de ce code ne correspond pas toujours, en termes de performances ou de comportement, à celui attendu par le concepteur. Il est donc nécessaire de fournir au concepteur un environnement de correction et d'optimisation à haut niveau étroitement lié à l'exécution du code généré.

Dans cette thèse, nous fournissons une solution de correction et d'optimisation de modèles reposant sur un mécanisme de traçabilité et d'instrumentation du code généré, et ce indépendamment du langage de transformation utilisé. La solution que nous proposons permet au concepteur de spécifier sur les modèles de haut niveau les informations qu'il voudrait observer lors de l'exécution du code généré. Par la suite, les observations sont remontées au concepteur sur les modèles de haut niveau.

La correction et l'optimisation de l'application générée repose sur une hypothèse forte : la chaîne de transformations, qui a permis la génération, est jugée digne de confiance. Cependant, lors de la phase de développement de la chaîne, des fautes peuvent être introduites dans les transformations, générant ainsi des erreurs, parfois complexes, dans le code de l'application. Tester les transformations est donc nécessaire pour obtenir des chaînes de transformations dignes de confiance.

Dans cette thèse, nous avons aussi travaillé sur l'utilisation de notre mécanisme de trace comme aide à la localisation d'erreurs dans les transformations de modèles et les chaînes de transformations et à l'analyse de mutation adaptée aux transformations de modèles. Dans le contexte de la localisation d'erreurs, la trace permet d'affiner le champ de recherche du testeur et ainsi lui éviter d'avoir à chercher une faute dans toute la transformation ou toute la chaîne. Pour l'analyse de mutation, nous proposons une approche reposant sur notre mécanisme de trace et une abstraction des opérateurs de mutation pour assister les testeurs dans certaines des étapes de cette technique, encore manuelle.

Les travaux présentés dans cette thèse (sur la correction et l'optimisation de modèles et ceux sur le test de transformation) ont été implémentés dans l'environnement Gaspard2. Notre mécanisme de trace étant indépendant de tout langage de transformations, ces travaux sont réutilisables pour d'autres chaînes de transformations.

PUBLICATIONS

Une partie du matériel présenté dans ce chapitre a également été publié dans les publications suivantes :

CHAPITRES DE LIVRE [1]

[1] VINCENT ARANEGA, JEAN-MARIE MOTTU, ANNE ETIEN, JEAN-LUC DEKEYSER : Using Trace to Situate Errors in Model Transformations, *Software and Data Technologies, Communications in Computer and Information Science*, 2011, Volume 50, Part 3, p.137 – 149

CONFÉRENCES INTERNATIONALES À COMITÉ DE LECTURE ET ACTES [1]

[2] VINCENT ARANEGA, JEAN-MARIE MOTTU, ANNE ETIEN, JEAN-LUC DEKEYSER, Traceability Mechanism for Error Localization in Model Transformation, *Proceedings of the 4th International Conference on Software and Data Technologies (ICSOFT'2009)*, Sofia, Bulgaria, July 2009

WORKSHOPS INTERNATIONAUX AVEC COMITÉ DE LECTURE [3]

[3] VINCENT ARANEGA, JEAN-MARIE MOTTU, ANNE ETIEN, JEAN-LUC DEKEYSER : Traceability for Mutation Analysis in Model Transformtion, *Post Proceedings of Workshops associated to the 2010 Models Conference*, 2011, p.259 – 273

[4] VINCENT ARANEGA, ANNE ETIEN, JEAN-LUC DEKEYSER, Using an Alternative Trace for QVT, *4th International Workshop on Multi-Paradigm Modeling (MPM'10)*, Oslo, Norway, October 2010

[5] VINCENT ARANEGA, JEAN-MARIE MOTTU, ANNE ETIEN, JEAN-LUC DEKEYSER, Using Traceability to Enhance Mutation Analysis Dedicated to Model Transformation, *Models Workshop on Model-Driven Engineering, Verification, and Validation (MoDeVA'10)*, Oslo, Norway, October 2010

RAPPORT DE RECHERCHE [1]

[6] A. WENDELL O. RODRIGUES, VINCENT ARANEGA, ANNE ETIEN, FRÉDÉRIC GUYOMARC'H, JEAN-LUC DEKEYSER, Enabling Traceability in an MDE Approach to Improve Performance of GPU Applications, *RR-7720*.

REMERCIEMENTS

Je tiens à remercier Jean-Luc Dekeyser pour m'avoir donné l'opportunité d'effectuer cette thèse. Je tiens également exprimer ma profonde gratitude envers Anne Etien pour m'avoir encadré, conseillé et soutenu tout au long de cette thèse et pour m'avoir guidé et aidé lors de l'écriture de ce manuscrit. Je remercie également Benoit Baudry et Éric Senn, pour avoir accepté de rapporter cette thèse, ainsi que pour leurs remarques, questions et recommandations.

Je tiens aussi à remercier Lionel Seinturier pour avoir accepté de présider mon jury de thèse et je remercie également Frédéric Robert pour avoir accepté d'en faire partie.

Les travaux présentés font tous partie du projet Gaspard2 et ils n'auraient pas pu avoir lieu sans le travail fourni par tous les autres contributeurs de ce projet. J'en profite pour exprimer toute ma sympathie et gratitude à l'ensemble des membres de l'équipe DaRT de l'INRIA Lille-Nord Europe et du Laboratoire d'Informatique Fondamentale de Lille, pour leur aide et leur collaboration.

Je souhaiterai aussi remercier mes parents, ma sœur et mes amis pour leur soutien au cour de ces trois années de thèse. Je tiens à remercier tout spécialement Wendell pour son soutien durant cette aventure et pour m'avoir grandement conseillé pendant l'écriture du manuscrit ainsi qu'Emeline pour m'avoir soutenu

TABLE DES MATIÈRES

TABLE DES FIGURES

LISTE DES TABLEAUX

INTRODUCTION

CONTEXTE

Le monde dans lequel nous évoluons tend à devenir de plus en plus complexe. Cela impacte la plupart des domaines de l'informatique, où une tendance à la complexité est observée. Cette complexité se retrouve bien entendu dans le volume ou les types de données traités par les applications, mais aussi dans les architectures sur lesquelles ces applications sont exécutées. D'un côté, les spécificités de ces nouvelles architectures augmentent les possibilités qui sont offertes au développeur, mais entraînent une quantité importante de paramètres à prendre en compte lors d'un développement. D'un autre côté, ces détails et contraintes liées à l'architecture rendent véritablement les phases d'implémentation longues, fastidieuses et difficiles à mettre au point. Les développeurs se concentrent alors sur les détails liés à l'implémentation plutôt que sur les fonctionnalités des applications qu'ils développent.

Aux vues de ces complexités, de nouvelles techniques et paradigmes de programmations sont sans cesse recherchés. Les travaux proposés par l'IDM (Ingénierie Dirigée par les Modèles), prônant une montée en abstraction lors des phases de développement, apparaissent alors comme un moyen de se défaire des détails d'implémentation liés à l'utilisation, par exemple, d'une bibliothèque ou d'un langage de programmation en particulier. La spécification du système peut alors se faire grâce à un modèle de conception, exprimant les différentes fonctionnalités du système modélisé. Comme complément à l'utilisation de modèles pour abstraire les détails d'implémentation, l'IDM propose l'utilisation de transformations de modèles à modèles et de modèles vers texte permettant, entre autre, de passer automatiquement d'un niveau d'abstraction à un autre. Ainsi, les détails d'implémentation liés à l'application peuvent être masqués au concepteur et le code source d'une application modélisée peut être directement généré. Ces chaînes de transformations, lorsqu'elles génèrent le code source d'un programme, peuvent s'apparenter aux compilateurs traditionnels utilisés en programmation. On parle alors de chaîne de compilation IDM. L'utilisation conjointe de la modélisation et des transformations de modèles offrent un support attractif pour le développement rapide d'applications sans avoir à composer avec les détails et spécificités introduits par un langage de programmation.

Dans cette thèse, nous nous intéressons au domaine de l'IDM et plus particulièrement des chaînes de transformations comme celles qu'il est possible de trouver dans l'environnement Gaspard2. Cet environnement, développé par l'équipe INRIA-LIFL DaRT dans laquelle j'ai effectué ma thèse, propose de modéliser à haut niveau des applications massivement parallèles. Des chaînes de compilation IDM constituent la colonne vertébrale de cet environnement. Gaspard2 propose de générer automatiquement le code d'une application en tenant compte des spécificités de l'architecture finale sur laquelle l'application devra être exécutée.

Les utilisateurs de Gaspard2 conçoivent leurs modèles de haut niveau représentant l'application, l'architecture et l'association de l'un

sur l'autre afin de générer des systèmes performants. Concevoir du premier coup des modèles de haut niveau permettant la génération d'un code performant n'est pas une tâche aisée. Les utilisateurs de Gaspard2 ont donc besoin d'aide pour corriger leurs modèles et les améliorer, d'autant qu'il leur ait souvent impossible de toucher le code directement, car ils n'ont pas forcément les compétences pour et tant bien même, cela aboutirait à des situations où le code et le modèle ne sont plus cohérents. Dans cette thèse, nous soutenons qu'une solution pour aider les utilisateurs à accroître les performances du code généré passe par une correction et optimisation de modèles.

Les travaux présentés dans cette thèse tentent de répondre à ces problématiques de correction et d'amélioration de modèles ainsi que de test des chaînes de transformation. Ces besoins, exprimés à travers Gaspard2, sont des besoins inhérents à toute chaîne de compilation IDM. En effet, les chaînes de compilations sont de plus en plus utilisées et de nombreux travaux font état de solutions pour aider à leur conception. Cependant, même si le passage à un niveau d'abstraction supérieur tend à simplifier la conception de systèmes complexes, cela n'exclut pas l'introduction d'erreurs dans les modèles par le concepteur et donc un comportement inattendu de l'application. De même, une modélisation sans erreur n'assure pas que les performances de l'application générée seront optimales.

De nombreuses solutions pour la correction et l'amélioration des performances d'application sont proposées dans le contexte des langages de programmation traditionnels et de la compilation classique. Ces solutions permettent aux développeurs de récupérer un ensemble d'informations à l'exécution et de les retourner sur leur code source pour leur permettre de comprendre et/ou de corriger leurs programmes, correspondant à l'artefact de premier plan du développement. En revanche, lorsque l'on traite avec des chaînes de compilation IDM, l'artefact de premier plan n'est plus le code source du programme, mais le modèle de conception. Utiliser les solutions existantes destinées à la correction ou l'amélioration du code source n'est donc pas pertinent. Ces solutions nécessitent d'être portées à un autre niveau d'abstraction pour bénéficier d'une correction et d'une amélioration de l'application à partir des modèles de conception d'une application et non plus sur son code source.

Évidemment, pour bénéficier d'outils de correction et d'amélioration de modèles, il faut que les chaînes de compilation IDM utilisées soient largement testées et dignes de confiance. Pour pouvoir fournir au concepteur une chaîne de conception possédant cette qualité, il faut fournir au développeur de la chaîne un moyen de trouver les erreurs qu'il aurait pu insérer et assurer que le jeu de tests qu'il utilise est suffisant pour pouvoir décréter sa chaîne de compilation comme digne de confiance.

Les travaux que nous proposons dans cette thèse ont été implémentés et validés dans le contexte de l'environnement Gaspard2 et des chaînes de compilations qu'il propose. Néanmoins, les solutions que nous apportons au cours de ces chapitres ne sont pas uniquement dédiés à Gaspard2. Ils peuvent être réutilisées pour d'autres environnements de modélisation reposant sur des chaînes de compilation IDM.

PROBLÉMATIQUES

Proposer une suite de solutions pour aider à la correction et à l'amélioration de modèles de conception amène à devoir considérer le fossé qu'il existe entre les modèles de conception et l'exécution de l'application générée à partir de ceux-ci, mais aussi à garder un lien entre les différents concepts tout au long de la chaîne. La première question qui se pose à nous est la suivante :

> QUESTION 1. Comment garder un lien entre les différents artefacts tout le long d'une chaîne de transformations ?

Lorsqu'un tel mécanisme existe, il est possible de mettre en relation une partie de l'exécution d'une application avec ses modèles de conception. Dans ce cadre, la possibilité de vérifier la correspondance entre le comportement de l'application et sa modélisation donne un moyen simple et efficace de corriger d'éventuelles erreurs dans les modèles. Pour arriver à ce but, la problématique impliquée est la suivante :

> QUESTION 2. Comment récupérer, sur les modèles de conceptions, des informations obtenues à l'exécution d'une application générée ?

Lorsque l'on passe d'une activité de correction de modèles à une activité d'amélioration de performances de modèles, les informations obtenues lors de l'exécution de l'application sont importantes pour le concepteur. Néanmoins, la modification du modèle en fonction des informations récupérées nécessite une connaissance approfondie de l'architecture d'exécution et une expertise pour pouvoir correctement appréhender ces informations. Soulager le concepteur de cette analyse est un plus qui permettrait au concepteur de modifier facilement son modèle. La question qui se pose alors est la suivante :

> QUESTION 3. Dans une activité d'amélioration de performances, comment tirer automatiquement partie d'informations de l'exécution de l'application et de l'architecture pour modifier efficacement le modèle et atteindre de meilleures performances ?

Comme nous l'avons évoqué précédemment, l'assistance et les outils qui découlent des questions précédentes n'ont un sens que si la chaîne de compilation utilisée est correctement testée. Lors d'une phase de test, la détection d'erreur dans les modèles de sortie d'une transformation dépend entièrement d'un problème de comparaison de modèles. S'il est simple d'identifier l'erreur dans les modèles, il est en revanche complexe de déterminer où la faute se situe dans la transformation ou dans la chaîne de transformations. Afin d'assister le testeur lors de son activité de correction de transformations, nous nous sommes posé la question suivante :

> QUESTION 4. Comment aider les développeurs de chaînes de compilation à localiser une erreur dans une transformation ?

La mise en évidence d'erreurs dans les transformations de modèles dépend grandement du jeu de tests utilisé. Si le jeu de tests est suffisamment sensible, celui-ci peut mettre facilement en évidence les fautes dans les transformations de modèles. Parmi les techniques existantes, l'analyse de mutation permet de juger de la qualité d'un jeu de tests de manière plus efficace que les autres techniques. Cette technique procède en plusieurs étapes qui peuvent être effectuées automatiquement, mis à part la dernière étape : l'amélioration du jeu de tests, actuellement toujours effectuée manuellement. Cette dernière étape est très longue et fastidieuse à mettre en œuvre et constitue un frein à la démocratisation de cette technique de qualification de jeu de tests. Cela entraîne l'interrogation suivante :

> QUESTION 5. Comment assister le développeur lors de la phase d'amélioration du jeu de tests, seule phase encore manuelle du processus d'analyse de mutation ?

Comme premiers éléments de réponses, la section suivante esquisse les propositions de cette thèse concernant chacune des questions relevées ci-dessus.

CONTRIBUTIONS

Les contributions principales de cette thèse reposent sur l'utilisation des traces de transformations de modèles, permettant de conserver un lien entre les modèles d'entrée et de sortie d'une transformation. Grâce à elles, nous avons proposé des solutions pour les corrections de modèles et de transformations dans le contexte de chaînes de compilation IDM.

Contributions pour la traçabilité des transformations de modèles à modèles

Dans le domaine de la traçabilité pour les transformations de modèles à modèles, nous avons proposé 2 contributions importantes :

1. proposition d'un mécanisme de trace de transformations de modèles indépendant de tout langage de transformation et proposition de notre mécanisme de trace comme complément au mécanisme de trace proposé par QVT [98], le standard dans les langages de transformation. Ce mécanisme repose sur l'utilisation de deux méta-modèles de trace pour gérer aussi bien une transformation unique qu'une chaîne de transformations.
2. implémentation de notre mécanisme pour QVTo [18] et proposition de l'utilisation d'une trace réduite dans les chaînes de transformations comme moyen d'améliorer la taille générale des traces de transformations de modèles.

Contributions pour la correction de modèles et l'amélioration des performances des applications modélisée

Dans cette thèse, nous avons proposé 3 contributions majeures dans le domaine de l'observation et de la correction de modèles :

1. proposition d'une approche d'optimisation des performances d'une application générée à partir de modèles en utilisant une chaîne de compilation IDM. Notre approche repose sur une visualisation, sur les modèles de conception, d'informations liées à la performance de l'application modélisée et récupérées à l'exécution de celle-ci ;
2. proposition d'une approche pour l'observation partielle de modèles. L'approche permet de mettre en relation l'exécution de l'application générée et les modèles de conception pour observer, à la demande du concepteur, certains éléments de ses modèles de conception ;
3. validation de nos approches d'optimisation de modèles et d'observation partielle de modèles dans le cadre de Gaspard2. Nos approches ont été validées sur deux chaînes différentes : une chaîne de compilation vers du code OpenCL et une chaîne de compilation vers du code PThread.

Contributions pour le test de transformations de modèles

Pour le test de transformations de modèles, nous avons proposés 3 contributions permettant d'aider à conception d'une chaîne de compilation digne de confiance. Ces contributions sont les suivantes :

1. proposition et implémentation d'un algorithme de localisation d'erreur pour les transformations de modèles et les chaînes de transformations de modèles ;
2. proposition et implémentation d'un assistant pour la phase d'amélioration du jeu de modèles de test de la technique d'analyse de mutation ;
3. validation de notre algorithme de localisation d'erreur et de notre assistant pour l'analyse de mutation sur plusieurs transformations et chaînes de transformations.

Plan

Dans la première partie de cette thèse, constituant l'étude de l'existant, nous nous intéressons au concepts de premiers plan de l'IDM et plus particulièrement aux mécanismes de traçabilités dans les transformations de modèles.

Dans un premier temps, dans le chapitre 1 – ingénierie dirigée par les modèles et transformations de modèles, nous discutons rapidement des fondements de l'IDM et en présentant les concepts de chaînes de transformations et de chaînes de compilation IDM, notions clefs qui sont utilisées dans l'intégralité de cette thèse. Par la suite, au chapitre 2 – traçabilité dans les transformations de modèles, nous présentons et comparons les différents mécanismes de trace existants pour les transformations de modèles et les chaînes de transformations et nous montrons le besoin d'un nouveau formalisme pour la trace.

La seconde partie de cette thèse présente notre mécanisme de traçabilité utilisée ensuite dans la totalité de la thèse. Au chapitre 3 – MÉCANISME DE TRACES ADAPTÉ AUX TRANSFORMATIONS ET AUX CHAÎNES, nous proposons notre propre mécanisme de trace pour les transformations de modèles et les chaînes de transformations. Ce mécanisme de trace est entièrement indépendant de tout langage de transformations de modèles. Nous montrons ensuite comment nous avons adapté notre mécanisme pour le langage QVT.

Par le biais de ce chapitre, nous répondons à la première question que nous avons levée dans cette introduction.

La troisième partie de cette thèse explore la correction et l'amélioration de modèles.

Au chapitre 4 – CORRECTION ET OPTIMISATION DE MODÈLES, nous étudions les différents paradigmes et travaux existants. Comptes tenu des limitations de ces travaux, nous proposons au chapitre 5 – OPTIMISATION DE MODÈLES une approche reposant sur l'utilisation de notre mécanisme de trace couplée à un système expert pour permettre la visualisation d'informations de performance obtenues à l'exécution d'une application sur les modèles de conception. Au chapitre 6 – OPTIMISATION DE MODÈLES DANS LE CONTEXTE DE GASPARD2, nous utilisons notre approche d'optimisation de modèles sur un cas d'étude pour essayer d'améliorer les performances d'une application, à partir des modèles de conception. À l'issue de ces deux chapitres, nous pouvons apporter une réponse à la troisième question.

Nous avons ensuite réutilisé une partie de cette approche d'optimisation de modèles au chapitre 7 – OBSERVATION PARTIELLE DE MODÈLES, pour permettre au concepteur de modèles d'observer le comportement de son modèle et d'ainsi lui permettre de corriger d'éventuelles erreurs dans les modèles. Nous validons ensuite au chapitre 8 – DEBUGGING DE MODÈLES DANS LE CONTEXTE DE GASPARD cette nouvelle approche d'observation partielle de modèles sur un nouveau cas d'étude où nous corrigeons un modèle, d'apparence correcte, grâce aux informations obtenues lors de l'exécution de l'application générée. Ces deux chapitres nous permettent de répondre à la deuxième question énoncée dans cette introduction.

Dans la quatrième partie de cette thèse, nous revenons sur l'hypothèse forte sur laquelle repose la troisième partie, à savoir que les transformations ou les chaînes de transformations peuvent contenir des erreurs. Nous traitons alors avec les tests de transformations de modèles et de chaînes de transformations.

Au chapitre 9 – TEST DE TRANSFORMATIONS, nous discutons des différents travaux existants sur les tests de transformations de modèles et des manques actuels pour pouvoir effectuer efficacement une phase de test pour les transformations de modèles. Nous montrons au chapitre 10 – LOCALISATION D'ERREURS comment notre mécanisme de trace peut être utilisé avec un algorithme pour aider le développeur de transformations à situer une faute dans ses transformations ou ses chaînes de transformations. Dans ce chapitre, nous répondons à la quatrième question.

Au chapitre 11 – ANALYSE DE MUTATION, nous montrons comment notre mécanisme de trace peut aider à simplifier et accélérer la dernière phase de la technique d'analyse de mutation, proposant d'améliorer un jeu de tests existant. Ce chapitre sur l'analyse de mutation permet de répondre à la cinquième question.

Finalement, nous concluons dans la dernière partie de cette thèse et nous proposons des perspectives aux travaux que nous avons proposés dans ce document.

Première partie

ABSTRAIRE, MODÉLISER

1

INGÉNIERIE DIRIGÉE PAR LES MODÈLES ET TRANSFORMATIONS DE MODÈLES

Dans ce chapitre, nous détaillons le contexte de cette thèse et nous faisons état des travaux qui ont précédé et guidé cette thèse. Compte tenu du grand nombre de publications sur les briques de base et concepts de l'*ingénierie dirigée par les modèles*, dans ce chapitre, nous en proposons un bref état de l'art. Cet état de l'art s'étale sur deux chapitres, le premier se concentre sur les thématiques générales de l'IDM, alors que le second présente les travaux existants sur les *traces de transformations de modèles*. Dans ce premier chapitre, en section 1.1, nous commençons par nous intéresser aux concepts utilisés pour abstraire un système. Nous présentons ensuite les outils et les techniques utilisées pour manipuler ces abstractions en section 1.2 et en section 1.3 avant de nous attarder sur les chaînes de compilation en IDM et l'environnement Gaspard2, support de notre travail, en section 1.4.

1.1 MODÈLES, MÉTA-MODÈLES, MÉTA-MÉTA-MODÈLES ET MÉTA-MODÉLISATION

Pour représenter et travailler à un niveau d'abstraction plus élevé que celui proposé par la programmation classique, un nouveau paradigme de programmation est proposé, celui de *modèle*. Dans cette section, nous allons définir les concepts de base de l'IDM et les relations qui existent entre eux, car la suite du document s'appuie entièrement sur ces définitions, principalement issues des résultats publiés dans [44]. Nous tenons à faire remarquer qu'il s'agit de concepts généraux qui ne sont pas propre à l'IDM mis à part le vocabulaire utilisé.

1.1.1 *Modèles et systèmes*

Un *modèle* est une abstraction d'un système qui est construite dans un but donné. On dit qu'un modèle *représente* un système. Il existe donc une première relation de « représentation » entre un modèle et un système qui lui est associé. Un modèle est une abstraction, car il synthétise un sous ensemble des informations de ce système et évite certains détails. Un modèle étant construit dans un but précis, il ne contient que les informations pertinentes par rapport à son utilisation et donc, à ce but. Par exemple, une carte routière est une représentation abstraite d'un système réel, à savoir un territoire géographique, qui a pour but de faciliter les trajets et donc de donner des informations sur les routes, les villes ou encore, éventuellement, les dénivellations. Cependant, toutes les informations concernant, par exemple, les langues/dialectes parlés ou les activités sismiques sont omises.

Ces modèles sont définis grâce à des langages particuliers, des *méta-modèles*, qu'il est nécessaire de décrire de manière précise.

1.1.2 *Méta-modèles : langage de modélisation*

Un modèle n'est utilisable et manipulable que si les concepts servant à le décrire sont spécifiés. À l'image du modèle représentant le système, il faut un nouveau concept permettant de *représenter* des modèles. Un modèle est donc représenté par un *méta-modèle*. Il définit les éléments et la structure d'un modèle ainsi que sa sémantique [1]. Le méta-modèle étant comparable à une grammaire de langage, il peut donc représenter plusieurs modèles.

1. La sémantique généralement admise pour les méta-modèles est une sémantique informelle.

Un lien fort existe entre un méta-modèle et les modèles qu'il décrit, on parle de *conformité*. Un modèle est *conforme* à un méta-modèle si tous les éléments du modèle sont définis par le méta-modèle. En revenant sur l'exemple de la carte routière du paragraphe précédent, le méta-modèle de la carte est sa légende qui spécifie la signification des différents symboles utilisés pour construire le modèle. La carte est *conforme* au langage représenté par la légende ainsi que toutes les cartes qu'il est possible de créer à partir de cette légende. Cette notion est essentielle à l'IDM, mais ne lui est pas exclusive. En effet, n'importe quel langage de programmation est conforme à sa grammaire. Par exemple, un programme JAVA est conforme à la grammaire de JAVA.

1.1.3 *Méta-méta-modèle : langage de méta-modélisation*

Encore une fois, à l'image de la relation entre méta-modèle et modèle, pour pouvoir utiliser et manipuler un méta-modèle, il faut un langage qui peut le décrire. Il est donc possible de voir le méta-modèle comme un modèle conforme à son propre méta-modèle. Dans ce cas, il s'agit d'un *méta-méta-modèle*. Afin de ne pas remonter dans les niveaux d'abstraction indéfiniment et de ne pas définir un meta-méta-méta-modèle, les méta-méta-modèles sont conçus pour être auto-descriptifs, c'est-à-dire qu'ils sont capables de se définir eux-mêmes. En revenant sur l'exemple du programme JAVA, la grammaire de JAVA est définie en utilisant une EBNF et qui fait office de méta-méta-modèle. De même, l'EBNF est comparable à un langage décrivant des langages. Il est donc capable de se décrire lui-même. C'est ce même principe d'auto-description que l'on retrouve dans l'IDM.

1.1.4 *Langages de modélisation et outils de modélisation*

Une multitude de langage de modélisation existent dans la littérature. Actuellement, avec les facilités proposées par les outils de méta-modélisation et de modélisation, il est aisé de définir et construire son propre méta-modèle et par conséquent, son propre langage de méta-modélisation. Cependant, la profusion de langage et de formalisme, même au niveau d'un même projet, peut rendre difficile la compréhension entre les différents acteurs du développement. Dans un soucis d'uniformisation des langages de modélisation, UML a été proposé comme standard par l'OMG [2].

2. Object Management Group

Dans cette section, nous allons rapidement présenter UML qui est une brique de base utilisé dans notre travail. Il en est de même pour le standard MARTE, également proposé par l'OMG, qui fournit des concepts de haut niveau pour la représentation de systèmes embarqués.

UML L'Unified Modelling Language est un langage de modélisation graphique standardisé par l'OMG [101]. Il est défini par un méta-modèle auquel lui est associé une représentation graphique. UML permet une modélisation uniforme quel que soit le système modélisé. Cependant, sa sémantique, trop faible, est un véritable inconvénient. La notion de *profil* lui a donc été ajoutée afin d'enrichir la sémantique des concepts proposés par UML par l'intermédiaire de *stéréotypes*. En utilisant les stéréotypes, il est possible d'ajouter des attributs et relations aux éléments qu'il raffine grâce à l'utilisation de références et de *tagged values*. Actuellement, de nombreux outils, propriétaires (par exemple : MagicDraw UML [3], Entreprise Architect [4]) ou libres tel que Papyrus UML [5], proposent une implémentation de ce standard.

3. http://www.nomagic.com

4. http://www.sparxsystems.com.au/

5. http://www.papyrusuml.org/

MARTE Modeling and Analysis of Real-Time and Embedded Systems est un profil standard proposé par l'OMG [102] visant principalement à ajouter à UML des concepts pour la conception et l'analyse des systèmes embarqués temps réel. UML fournit donc les concepts de bases et MARTE les étend. De cette façon, MARTE permet de modéliser des applications, des architectures ainsi que les relations entre elles et bien d'autres choses comme, par exemple, le temps.

Dans cette thèse, d'autres formalismes et méta-modèles sont aussi définis. Cependant, nous ne les détaillons pas dans cette section, mais nous les présenterons au fur et à mesure du document.

1.2 TRANSFORMATIONS DE MODÈLES À MODÈLES

Les transformations de modèles à modèles permettent de produire et de modifier automatiquement des modèles et présentent donc un intérêt majeur de l'IDM [112]. Cette manipulation automatique est essentielle pour plusieurs raisons :

- Les modèles peuvent être relativement complexes, impliquant une complexité équivalente pour les manipuler manuellement.
- Les modèles à traiter peuvent être nombreux et un même traitement peut devoir être appliqué plusieurs fois pour chacun d'eux.
- Le gain de temps obtenu grâce au travail à un niveau d'abstraction plus élevé est perdu si le passage vers différents niveaux n'est pas automatisé.

Dans la suite de la section, nous présentons les différents types de transformations de modèles avant de présenter rapidement quelques langages et outils de transformation de modèles les plus utilisés en IDM.

1.2.1 *Principe de la transformation de modèles*

De manière générale, une transformation est définie à partir de ses méta-modèles source et destination. Elle établit un ensemble de relations exprimant les actions à effectuer sur les modèles d'entrée pour produire les modèles de sortie correspondant. Pour écrire une transformation, différentes approches peuvent être employées. Il est possible de trouver une classification de ces différentes approches dans [28, 27].

Le processus général d'une transformation consiste à transformer un modèle M_1 en un modèle M_2 qui sont conformes à leurs méta-modèles respectifs MM_1 et MM_2. La transformation est dite *endogène* si $MM_1 = MM_2$, sinon elle est dite *exogène* [89]. Dans [28, 27], Czarnecki et al. différencient, sans les nommer, deux types de transformations, les transformations produisant un nouveau modèle et celles modifiant un modèle existant. Dans cette thèse, nous adoptons la dénomination *in* vers *out* pour parler de transformations de modèle produisant un nouveau modèle et *inout* ou transformation *sur place* pour parler de transformations de modèles modifiant des modèles existants.

Une classification des différents types de transformation peut aussi être faite selon le changement de niveau d'abstraction. Une transformation est dite *horizontale* lorsque le modèle produit par la transformation a le même niveau d'abstraction que celui d'entrée et *verticale* lorsque le modèle produit introduit un changement de niveau d'abstraction [89]. Par exemple, la génération d'un modèle spécifique conforme à un méta-modèle MM_b à partir d'un modèle plus général conforme à un méta-modèle MM_a introduit un changement de niveau d'abstraction. Cette transformation est donc une transformation *in* vers *out* verticale.

Le plus fréquemment, comme dans l'architecture MDA[6] [117] proposée par l'OMG, les transformations de modèles sont utilisées pour changer de niveau d'abstraction et produire un modèle dédié à une plate-forme [55]. De cette façon, il est possible d'automatiquement produire des modèles prenant en compte les spécificités et subtilités liées à une plate-forme tout en se concentrant uniquement sur les fonctionnalités du système modélisé.

6. Model Driven Architecture

1.2.2 QVT, *un standard pour les transformations de modèles*

Plusieurs langages de transformations existent tel que ATL[7] [72], EML[8] [75] ou encore Kermeta [91]. Cependant, un standard est proposé par l'OMG : QVT[9]. QVT repose sur l'utilisation du langage OCL[10] [100] pour naviguer à travers les modèles.

Afin d'écrire les transformations, le standard QVT définit trois sous-langages : Relation, Operational Mapping et Core. Chacun de ces sous-langages repose sur un paradigme de transformation de langage différent à savoir *déclaratif*, *impératif* et *hybride* [79].

Les sous-langages Relation et Core sont purement déclaratifs. Le sous-langage Core est volontairement simple et sert de base au sous-langage Relation. Cependant, comme certaines fois il est difficile de

7. ATLAS Transformation Language
8. Epsilon Merging Language
9. Query Views Transformation [98]
10. Object Constraint Language

proposer une solution entièrement déclarative pour une transformation, QVT propose deux mécanismes pour étendre RELATION et CORE : le sous-langage impératif OPERATIONAL MAPPING et un mécanisme pour invoquer des fonctionnalités utiles à la transformation implémentée dans un langage quelconque.

Le standard QVT propose une définition syntaxique et sémantique des trois sous-langages, mais ne donne aucune direction concernant l'implémentation. Ainsi, plusieurs implémentations existent actuellement, à savoir, QVTo [18] proposé par l'éditeur BORLAND et SMART-QVT [13] proposé par FRANCE TÉLÉCOM. Ces deux implémentations concernent uniquement le sous-langage OPERATIONAL MAPPING et non pas les autres sous-langages. Finalement, comme implémentation du sous-langage RELATION, on compte QVTr développé par OBEO [96].

1.3 TRANSFORMATIONS DE MODÈLES VERS TEXTE

Les transformations de modèles à modèles ne sont pas les seuls types de transformation que l'on peut retrouver en IDM. Les transformations de modèles vers texte prennent des modèles en entrée et produisent du texte qui peut être, par exemple, du code source ou encore de la documentation. Dans la section suivante, nous présentons brièvement les types de mécanismes utilisés par les transformations de modèles vers texte.

Principe de la transformation de modèles vers texte

Le processus général d'une transformation de modèles vers texte consiste à générer un morceau de code pour un objet ou un ensemble d'éléments des modèles sources. Afin de réaliser une telle tâche, deux approches existent : la première repose sur l'utilisatoin d'un patron *visiteur*, alors que la seconde repose sur l'utilisation de *template* [28].

La première approche, assez peu utilisée, définit un mécanisme de parcours de la représentation interne d'un modèle et d'écriture du code (texte) dans un flux de sortie. Cependant, la représentation interne doit être assez proche de la structure du code visé, ce qui rend cette approche peu flexible.

La seconde approche est celle que l'on retrouve dans la plupart des outils de génération de code. Un *template* consiste en un bloc de texte contenant des méta-codes permettant d'effectuer diverses tâches sur les modèles sources et ce, potentiellement itérativement. Finalement, la structure du *template* est, généralement, assez proche du code généré.

1.4 CHAÎNES DE TRANSFORMATIONS ET COMPILATION

Les transformations de modèles apportent un moyen efficace d'automatiser certaines actions sur les modèles. Avec les transformations de modèles à modèles, il est possible de passer d'un formalisme à un autre ou d'effectuer des opérations de raffinement sur le modèle. Avec les transformations de modèles vers texte, il est possible de générer du code source à partir des modèles. Cependant, lorsque les domaines manipulés sont extrêmement complexes, la complexité des transformations est, indubitablement, affectée. En effet, dans [126], les auteurs ont montré que la vision MDA : « PIM to PSM » est trop simpliste et ont mis en avant le besoin de chaîner les transformations pour le dévelop-

pement de systèmes complexes. En effet, il arrive que pour passer d'un formalisme à un autre, compte tenu de la complexité des domaines manipulés, plusieurs formalismes intermédiaires soient utilisés pour permettre une transition plus aisée et simplifier l'écriture individuelle des transformations. De cette façon, la sortie d'une transformation est directement consommée par une autre pour former une chaîne de transformations. Cependant, le chaînage des transformations impose une contrainte forte, pour pouvoir chaîner deux transformations, il faut que le méta-modèle de sortie de la première transformation soit inclus dans le méta-modèle de la seconde transformation.

1.4.1 *Lier, ordonner, organiser les transformations*

Lorsque l'on travaille avec plusieurs transformations que l'on veut chaîner, la question de la construction de la chaîne de transformations se pose. En effet, produire une chaîne de transformations revient à orchestrer un ensemble de transformations pour décider quel en sera l'enchaînement en tenant compte de la contrainte de précédence imposée par les domaines manipulés. Dans [126], Vanhoof et al. ont donné le processus général de construction d'une chaîne de transformations. Ils ont ensuite étendu ce travail et dans [127], ils ont proposé l'écriture d'un nouveau langage, reposant sur un profil UML, permettant de construire facilement des chaînes de transformation.

Les chaînes de transformations peuvent comprendre plusieurs types de transformations écrites dans différents langages de transformation. En effet, les contraintes permettant de lier une transformation à une autre ne concerne que les méta-modèles d'entrée et de sortie des transformations. En suivant ces contraintes de séquentialité, il est aussi possible d'utiliser des transformations de modèles vers texte dans une chaîne de transformations de modèles. Une chaîne de transformations permet donc d'utiliser plusieurs langages de transformation différents ainsi que plusieurs types de transformations différents.

Les chaînes de transformations sont couramment utilisées pour produire le code source d'une application à partir de ses modèles de conception. Dans la plus part des solutions existantes actuellement, le code source produit est juste un squelette représentant la structure générale de l'application modélisée. Cependant, il existe aussi des solutions à base de chaînes de transformations générant le code source de l'application entièrement, c'est-à-dire, prêt à être compilé. Dans cette thèse, nous parlons de *chaîne de compilation* IDM pour parler de ce types de chaînes de transformations puisque leur structure est à l'image des différentes phases de compilation d'un compilateur.

1.4.2 *Transformations localisées*

Les chaînes de transformations permettent de séparer facilement les préoccupations [126] et facilitent ainsi l'écriture individuelle de chaque transformation. Cependant, la prise en compte des exigences et des contraintes de chaque transformation implique une réutilisation difficile des transformations compte tenu de la liaison forte qu'il existe entre chaque transformation et celle qui la précède. Ainsi, dans [129], les auteurs ont proposé un moyen de spécifier les éléments requis par la transformation et ceux qu'elle fournit. De cette manière, il est possible de considérer chaque transformation comme un bloc plus ou moins

indépendant des autres transformations qu'il est possible d'ordonner en fonction de ses attentes et de ses productions. Cependant, ce travail ne s'intéresse pas au chaînage des transformations complètement indépendante.

Afin de simplifier la réutilisation de transformations dans une même chaîne, les travaux proposés en [41] ont défini l'utilisation d'un nouveau principe d'écriture des transformations appelées « transformation localisée ». Chacune des transformations proposées est vue comme une unique transformation fin grain et ne travaille que sur une partie réduite du méta-modèle d'entrée. Pour proposer une telle flexibilité, un concept d'extension de transformation pour un méta-modèle donné est proposé. De cette façon, une transformation étendue se compose de deux parties, une transformation localisée n'impliquant qu'un petit nombre de concepts et une copie implicite pour tous les concepts du méta-modèle source de la transformation étendue qui ne sont pas compris dans celui de la transformation localisée. Le mécanisme de la transformation *sur place* est, en quelque sorte, étendu pour les transformations où le méta-modèle source et destination sont différents.

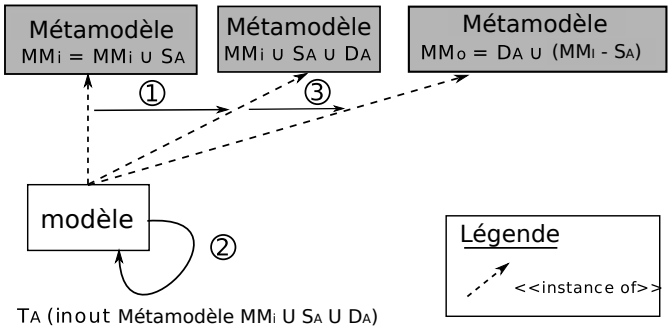


FIGURE 1: Exemple d'exécution d'une chaîne de transformations localisées

À titre d'exemple, la figure 1 présente la stratégie de transformation adoptée par les transformations localisées. La transformation T_A illustrée correspond à la version étendue sur le MM_i d'une transformation localisée T_A de S_A vers D_A. Elle transforme un modèle conforme à un méta-modèle $MM_i \cup S_A$ en un modèle conforme à un méta-modèle $MM_i \cup D_A$. En tenant compte des remarques précédentes, la transformation T_A est écrite en utilisant les concepts de S_A et de D_A. Exactement, T_A est une version étendue de la transformation que l'on applique. Ainsi, la copie implicite permet d'appliquer T_A sur des modèles instances de $MM_i \cup S_A$ pour créer des modèles conformes à $MM_i \cup D_A$. D'un point de vue pratique, afin de mettre en œuvre la copie implicite et ne prendre en compte que les éléments modifiés par la transformation, T_A est considérée comme une transformation sur place (spécifiée comme une transformation *sur place* sur la figure) sur $MM_i \cup S_A \cup D_A$. L'exécution d'une transformation localisée peut donc théoriquement se décomposer en 3 étapes. Les modèles source sont copiés puis rendus conformes au méta-modèle $MM_i \cup S_A \cup D_A$ grâce à une opération de *shift* de méta-modèle (étape 1). La transformation est ensuite exécutée (étape 2). Une fois la transformation terminée, le modèle est ensuite *shifté* pour qu'il soit conforme au méta-modèle $D_A \cup (MM_i - S_A)$, soit le méta-modèle de sortie (étape 3).

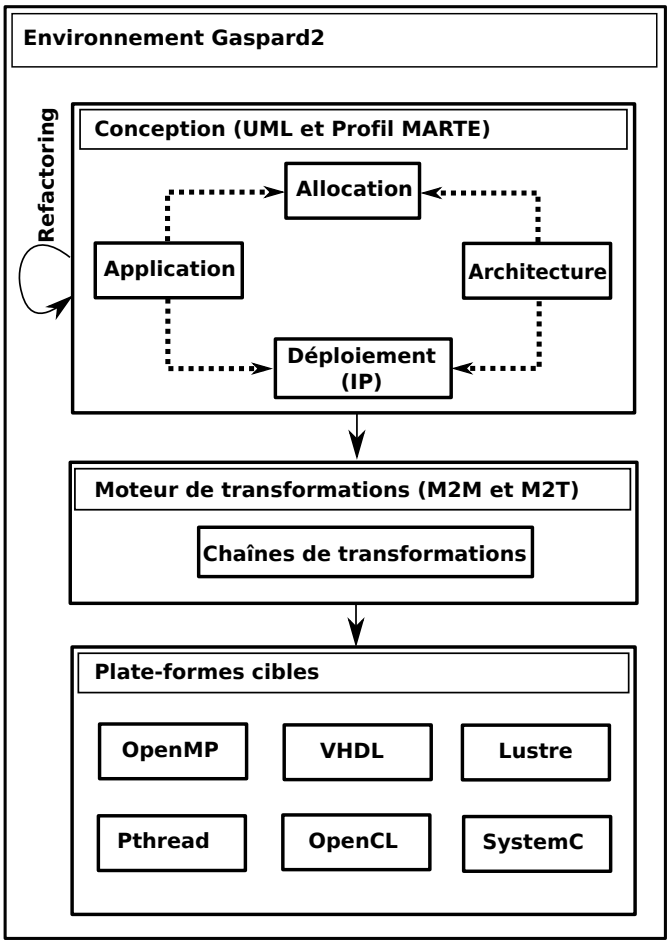


FIGURE 2: Architecture de l'environnement GASPARD2

1.4.3 *L'environnement* GASPARD2

L'environnement GASPARD2(Gaspard Array SPecification for PArallel and Distributed computing) [53, 30] est un environnement de développement intégré dédié au co-design de systèmes embarqués, développé par l'équipe DaRT de l'INRIA et du LIFL (Laboratoire d'Informatique Fondamentale de Lille). L'environnement GASPARD2 repose sur une technologie IDM pour fournir un moyen de modéliser, simuler, analyser et générer du code pour les applications exécutées sur des architectures de type SoC (System on Chip). Afin d'illustrer nos propos, la figure 2 représente l'architecture interne de l'environnement GASPARD2. Deux phases distincts sont à considérer lorsque l'on travaille dans l'environnement GASPARD2 : une phase de modélisation et une phase de transformation.

La modélisation du système repose sur quatre modèles :

- un modèle d'*application* exprimant les fonctions réalisées par le système,
- un modèle d'*architecture* décrivant l'architecture matérielle accueillant l'application,
- un modèle d'*association* liant les tâches et les différentes routines du modèle d'application sur les unités de calculs décrites dans le modèle d'architecture,

– un modèle de *déploiement* permettant d'associer à chaque composant d'application (ou d'architecture) une IP [11]. Ces IP correspondent à des fragments de code implémentant une action, une fonction, un calcul particulier ou décrivant un composant matériel.

11. INTELLECTUAL PROPERTY

Les trois premiers modèles utilisent UML enrichi du profil MARTE alors que le quatrième modèle utilise un profil de *déploiement* défini au sein de l'équipe.

À partir de cette spécification du système, une série de plusieurs transformations qui, suivant les versions [12], peuvent être localisées ou non, permet, en passant par différent niveaux d'abstraction, la génération de ce système pour différentes cibles, utilisés pour effectuer différents traitement :

12. Dans cette thèse, nous utilisons la dernière version de GASPARD2.

– *validation* avec des langages synchrones tel que LUSTRE [141] ;
– *exécution* avec PTHREAD [53], OPENMP (C ou FORTRAN) [119] ou encore OPENCL [94] ;
– *simulation* au niveau TLM (Transaction Level Modeling) [23] en SYSTEMC [14, 108] ;
– *synthèse* FPGA en passant par du code VHDL [81, 109].

Exemple. Pour la branche OPENCL, le modèle UML profilé MARTE est transformé dans une représentation équivalente conforme au métamodèle MARTE. Ensuite, les différentes répétitions distribuées de l'application modélisée sont analysées pour en déduire les dépendances existant entre elles, en fonction de leur placement sur l'architecture CPU ou GPU. Ces dépendances sont ensuite ordonnancées statiquement. Par la suite, le code OPENCL de l'application, respectant la spécification modélisée [13] est généré à partir du dernier modèle produit par la chaîne de transformations et peut être utilisé pour tester l'exécution de l'application modélisée.

13. Par exemple, les dépendances de données ou encore le placement.

1.4.4 *Modéliser avec* GASPARD2

Comme nous l'avons indiqué précédemment, les modèles GASPARD2 sont composés de quatre modèles : un modèle d'*architecture*, un modèle d'*allocation*, un modèle d'*application* et un modèle de *déploiement*. Dans les sous-sections suivantes, nous allons présenter les quatre modèles constituant un modèle GASPARD2 sur l'exemple d'une application qui incrémente les différentes coordonnées d'un, ou plusieurs, vecteurs, puis les afficher. Cette application est placée sur une architecture composée de deux processeurs dont l'un est un *dual core*.

Modèle d'application

Le modèle d'application, illustré en figure 3, est représenté par un diagramme composite UML. La définition est l'utilisation des différents composants sont séparés. Ainsi, pour notre exemple d'incrémentation de vecteur, trois composants représentent les fonctionnalités élémentaires de l'application :

– *GenVector* pour la génération du vecteur,
– *Increment* pour l'incrémentation des éléments du vecteur,
– *PrintVector* pour l'affichage du vecteur.

Chacun de ces composants possède des ports. Par exemple, le composant *GenVector* possède un port *outGen* avec une *shape* de 27. Le composant *PrintVector* possède un port *inPrint* avec une *shape* de 27. Pour illustrer le parallélisme potentiel des applications GASPARD2, les

coordonnées du vecteur ne sont pas incrémentées toutes en même temps, mais trois par trois. Les tailles des deux ports *inIncr* et *outIncr* sont donc de 3. Ce composant sera répété 9 fois. Le composant *incrVect* permet de représenter cette répétition. Les *Tilers* expriment comment les sous-ensembles de 3 coordonnées sont sélectionnés parmi les 27 coordonnées du vecteur. Ainsi, à chaque *Tiler* est implicitement associé un algorithme permettant d'aller chercher des données selon un motif définit [19]. Ce motif est défini et paramétré par les *tagged values* : *origin*, *paving* et *fitting* du *Tiler*. Les *Tilers* définis ici précisent que le vecteur d'entrée est « consommé » par groupe de 3 éléments et que les éléments incrémentés sont rangés par groupe de 3 dans le vecteur de sortie. Le composant *MainAppli* représentant l'application générale contient une instance du composant *GenVector*, une instance du composant *incrVect* et une instance du composant *PrintVector*.

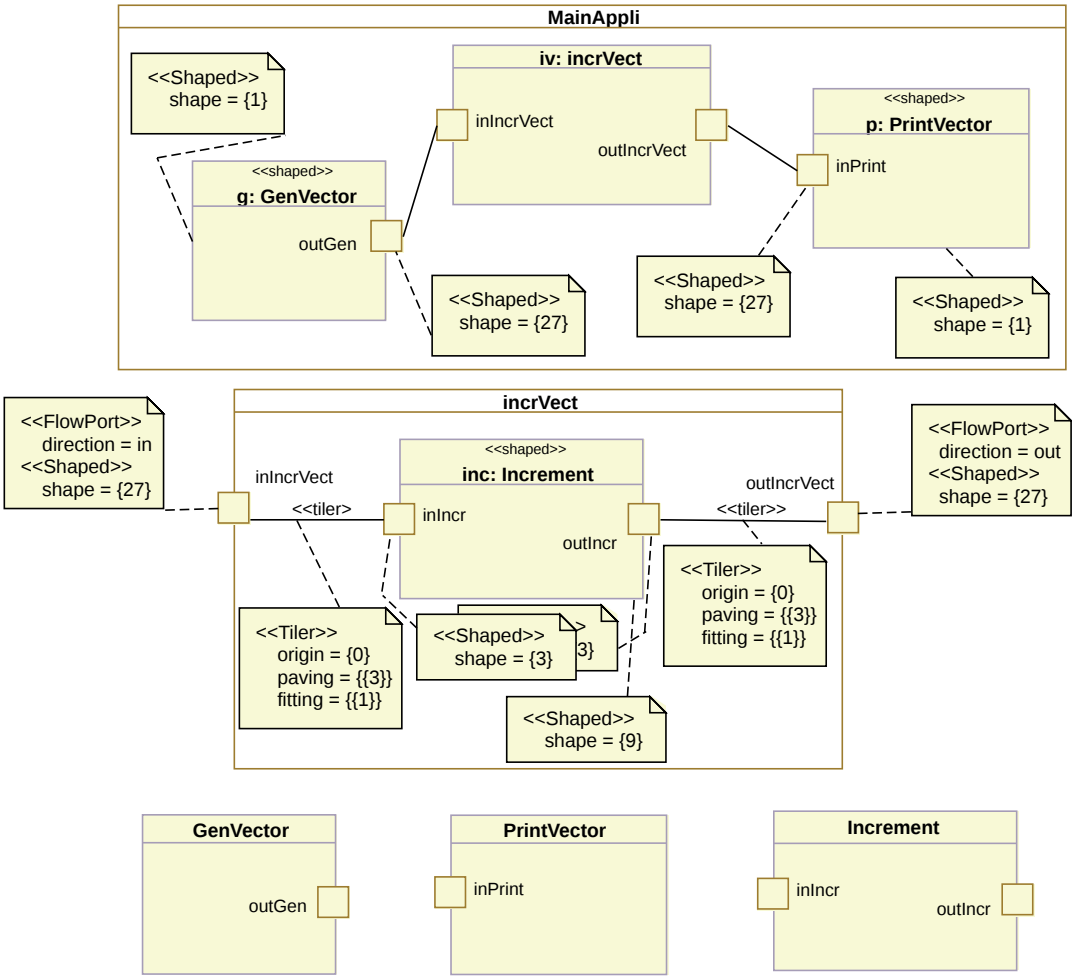


FIGURE 3: Modèle d'application

Modèle de déploiement

Le modèle de déploiement se concentre sur l'implémentation des différentes fonctions mathématiques et algorithmes élémentaires requis par l'application. La figure 4 présente les trois composants élémentaires de l'application : *GenVector*, *PrintVector* et *Increment*. Ces composants élémentaires sont implémentés par des fonctions existantes grâce aux

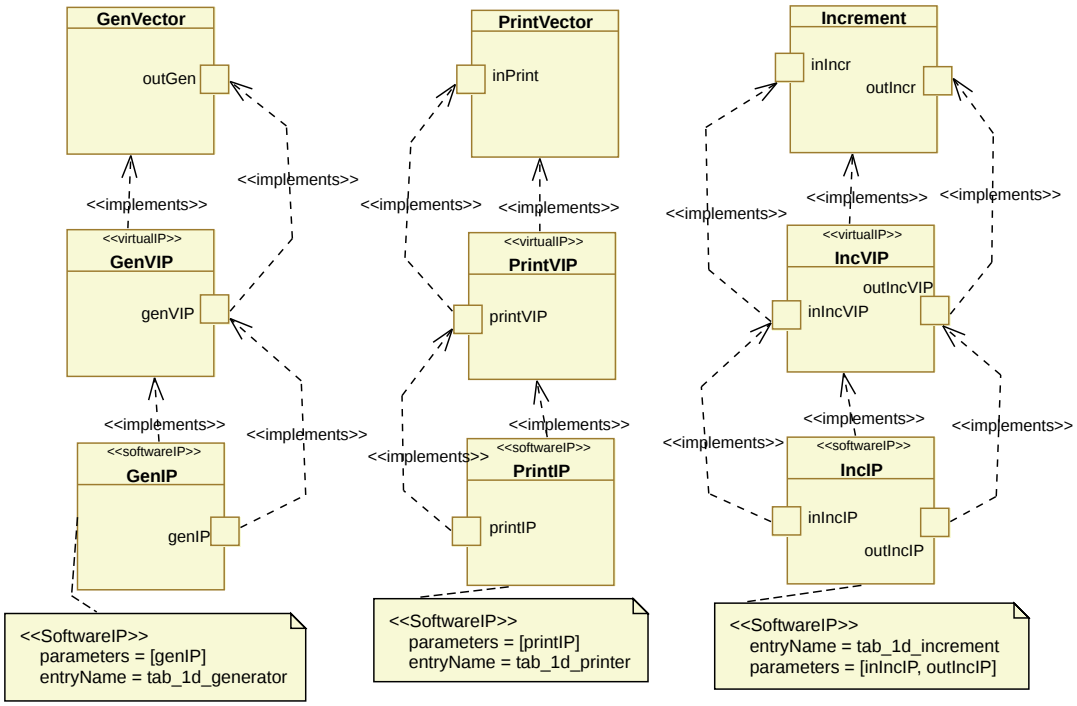


FIGURE 4: Modèle de déploiement

virtualIP [14] et *softwareIP* qui leur sont associés. Par exemple, le composant élémentaire *GenVector*, permettant de générer un vecteur, est implémenté par la fonction *tab_1d_generator* et possède un paramètre : *genIP*. Ainsi, lorsque un composant de type *GenVector* est créé dans le modèle d'application, un appel à la fonction *tab_1d_generator* est généré lors de l'étape de génération du code source. Il en est de même pour les fonctions d'affichage d'un vecteur d'une certaine taille (*PrintVector*) et l'incrémentation des coordonnées d'un vecteur d'une certaine taille (*Increment*).

14. Les *virtualIPs* permettent d'avoir une même interface pour plusieurs implémentations (*softwareIP*) d'une même fonction. Ces implémentations pouvant correspondre à des langages différents, ou à des version différentes du même algorithme.

Modèle d'architecture

L'architecture sur laquelle l'application d'incrémentation de vecteur est exécutée est décrite en figure 5. Cette architecture est composée de 2 processeurs : *proc1* et *proc2* de type *PaProcessor* [15]. Le processeur *proc1* est un simple processeur alors que le processeur *proc2* est en fait composé de 2 processeurs. Le processeur *proc1* est associé à un cache *cache1* et le processeur *proc2* ne possède pas de cache. Tous deux communiquent avec la mémoire *mem* grâce à un bus *router* permettant de router les informations demandées/enregistrées par les processeurs.

15. Ce type n'est pas défini dans le modèle, mais dans une bibliothèque annexe fournit avec GASPARD2.

Modèle d'allocation

Finalement, le modèle d'*allocation* permet de préciser sur quelles parties de l'architecture modélisée l'application modélisée va être exécutée [16]. Ce modèle, représentée à la figure 6, reprend les éléments des modèles d'application et d'architecture en précisant les différentes liaisons qui existent entre les deux modèles.

Ce modèle précise que le composant *g* générant le vecteur d'entrée est placé sur le premier processeur *proc1*. Le port de sortie de ce composant *outGen* est, lui, placé en mémoire *mem*. Quant au composant *p*,

16. Par exemple, sur quel processeur une tâche va être exécutée.

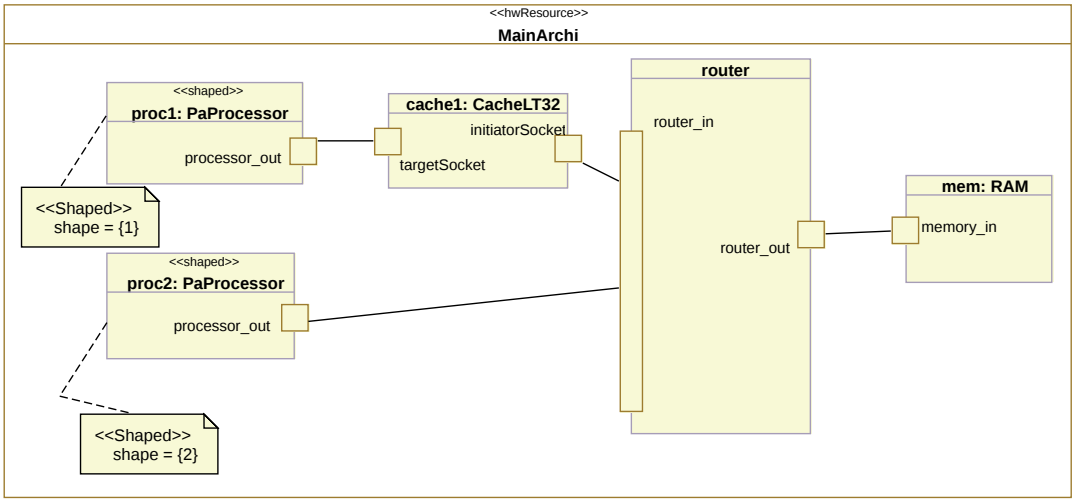


FIGURE 5: Modèle d'architecture

il est placé sur le second processeur *proc2* alors que son port d'entrée *inPrint* est placé dans la mémoire RAM : *mem*. Le composant *iv* n'est pas placé explicitement, mais c'est son sous composant *inc* qui est placé explicitement sur l'architecture modélisée. Sur le modèle, il est précisé que le sous composant *inc* est placé sur le premier processeur *proc1* et que ses ports d'entrée et de sortie sont placés sur la mémoire *mem*.

Ces quatre modèles ensemble permettent de générer automatiquement un code source complet qu'il est possible de compiler sans avoir à le modifier.

1.4.5 *Problématiques, contraintes et piste d'étude*

Dans le contexte de l'environnement GASPARD2 et des chaînes de compilation IDM, deux types d'acteurs interviennent : le concepteur de la chaîne de compilation IDM [17] et le concepteur de modèles [18]. L'existence de deux types d'acteurs implique des problématiques différentes selon leur point de vue.

17. le développeur

18. l'utilisateur terminal de la chaîne de compilation IDM

Concepteur de modèles

L'utilisateur de la chaîne de compilation se concentre sur la conception de modèles. Dans un premier temps, les modèles qu'il produit doivent être correctement modélisés, c'est-à-dire que la « syntaxe » du modèle doit être correcte. Cependant, un modèle correcte syntaxiquement n'assure pas que le comportement de l'application générée est celui attendu ou qu'il possède des performances adéquates. Comment alors corriger ou améliorer les performances des modèles produits ?

Les seuls informations disponibles sont relatives à l'exécution du code généré. Il est donc nécessaire d'en recueillir un maximum et de les reconnecter au modèle de conception pour informer l'utilisateur de la chaîne de compilation IDM. De cette façon, il serait possible de déterminer si le comportement ou les performances obtenues sont correctes.

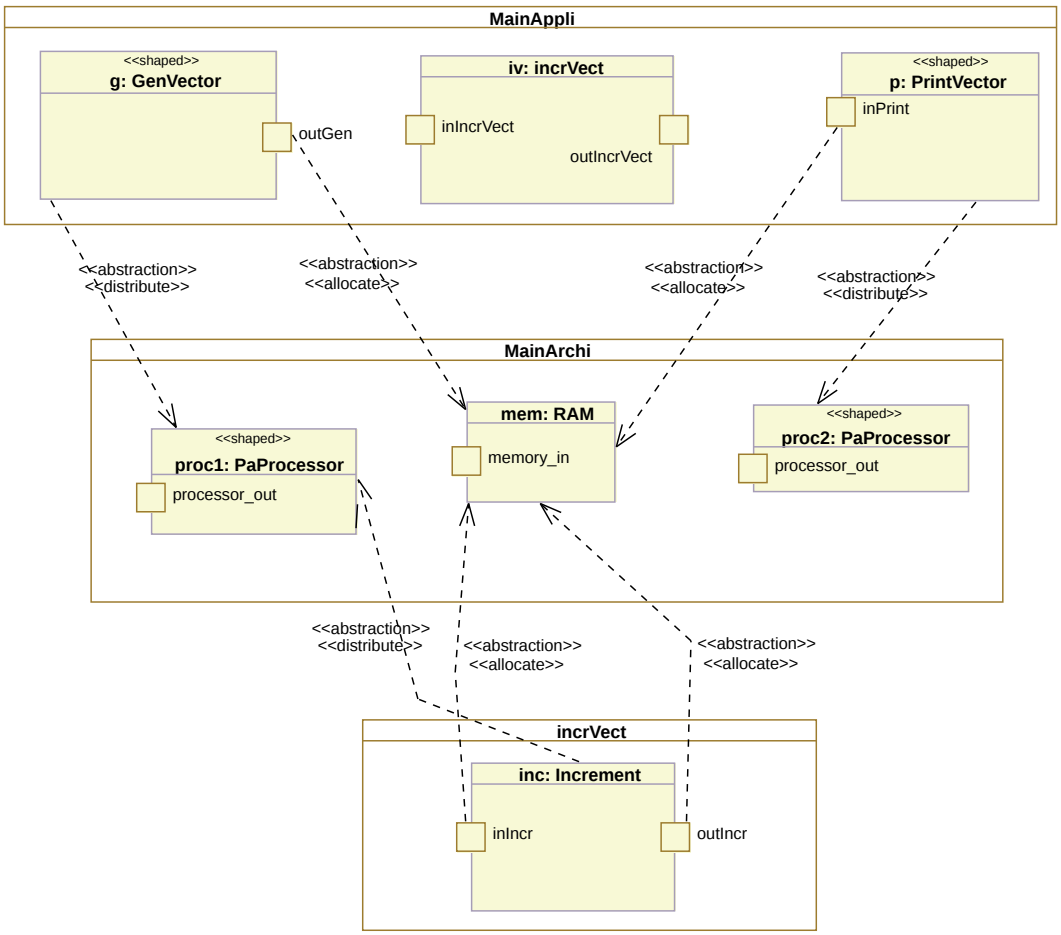


FIGURE 6: Modèle d'association

L'utilisateur peut se poser la question de la correction et des performances de ses modèles si la chaîne de compilation IDM qu'il utilise est digne de confiance.

Concepteur de la chaîne de transformation

L'objectif du développeur de la chaîne de compilation est différent de celui de l'utilisateur. Il doit assurer que la chaîne de compilation IDM est sans erreur et, par conséquent, que les transformations de modèles à modèles qu'il produit sont sans erreur. En effet, une erreur introduite dans le compilateur peut ce répercuter sur les applications générées même si le modèle proposé par l'utilisateur est sans erreurs. Comment alors fournir un moyen aux développeurs de corriger leurs transformations de modèles lorsqu'ils construisent une chaîne de compilation IDM ?

Cette fois, dans ce cadre, il est nécessaire de récupérer de l'information sur l'exécution des transformations de modèles pour pouvoir aider le développeur dans sa tâche de correction des transformations de modèles.

Contraintes sur les chaînes de compilation IDM *dans* GASPARD2

L'environnement GASPARD2 repose en très grande partie sur la notion de chaînes de compilation IDM en posant un certain nombre de

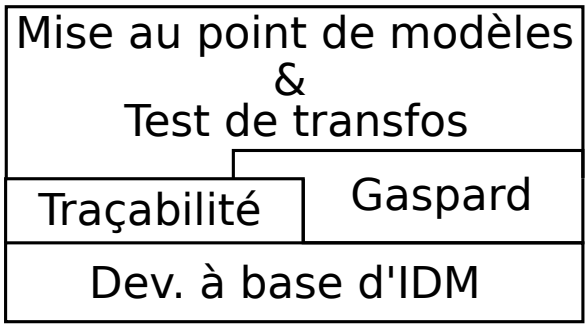


FIGURE 7: Architecture de notre proposition

conditions sur la chaîne de compilation en elle-même et l'utilisation de l'environnement. Ces contraintes sont les suivantes :

- les modèles intermédiaires générés pendant l'étape de compilation ne sont pas manipulables par le concepteur,
- les différentes étapes de la chaîne de compilation sont écrites avec des langages de transformations de modèles à base de règles,
- le code source généré ne nécessite pas de modification de la part du concepteur.

Il faut donc considérer ces contraintes comme des « pré-requis » à nos contributions.

La traçabilité

Pour les problématiques exposées précédemment, il est nécessaire de pouvoir :

- récupérer de l'information sur l'exécution des transformations de modèles exécutées dans une chaîne de compilation IDM,
- être capable de lier de l'information obtenue à différents niveau d'une chaîne de compilation IDM sur le(s) modèle(s) d'entrée du compilateur.

La traçabilité et plus précisément, la trace de transformations de modèles permet de retrouver ces informations. C'est donc sur la trace de transformations de modèles que nos contributions et nos solutions reposent. La figure 7 illustre comment nos contributions se positionnent par rapport aux chaînes de compilation IDM [19]. La traçabilité, et plus particulièrement la trace de transformations de modèles, est donc fortement liée aux transformations de modèles. Comme nous l'avons présenté précédemment, l'environnement GASPARD2 repose sur l'utilisation de chaînes de compilation IDM, mais aussi sur la traçabilité, car il intègre les contributions que nous proposons dans cette thèse. Finalement, les contributions que nous proposons dans cette thèse reposent sur la trace de manière générale, s'intègrent dans l'environnement GASPARD2 et prennent en compote les contraintes de cet environnement comme celles sur les transformations. Ces contributions sont réutilisable dans le cadre d'autres chaînes de compilation IDM pour peu qu'elles respectent les contraintes que nous avons énoncées précédemment.

19. Noté *dev. à base d'*IDM sur la figure.

1.5 CONCLUSIONS

Dans ce chapitre nous avons vu les concepts importants manipulés tout au long de cette thèse. Ces concepts sont parties intégrantes de l'IDM qui propose de travailler à un niveau d'abstraction élevé via

des modèles pour faciliter la conception des systèmes complexes. Les modèles manipulés sont construits grâce à des méta-modèles décrivant un ensemble de concepts spécifiques à un domaine.

Nous avons ensuite présenté les transformations de modèles qui fournissent un moyen efficace pour manipuler automatiquement les modèles et passer d'un formalisme à un autre. Dans ce contexte, nous avons identifié deux types de transformations : les transformations de modèles à modèles, manipulant et produisant des modèles, et les transformations de modèles vers texte, manipulant des modèles, mais produisant du texte en sortie.

Pour automatiser des actions complexes sur les modèles sans affecter la complexité d'une transformation, il est possible de les chaîner pour former une chaîne de transformations de modèles. Lorsque ces transformations sont chaînées afin d'obtenir, au final, le code source du système modélisé, on parle de chaîne de compilation IDM. Dans ce cadre, nous avons brièvement présenté une nouvelle façon de produire des chaînes de compilation en utilisant des transformations localisées.

Finalement, nous avons rapidement présenté GASPARD2, un environnement de co-design de systèmes embarqués, ainsi que son architecture générale reposant entièrement sur le concept de chaîne de compilation IDM. C'est dans le cadre de GASPARD2 et plus généralement des chaînes de transformations que se place cette thèse. Dans le prochain chapitre, nous allons présenter le concept de *trace* pour les transformations de modèles à modèles ainsi que les travaux existants à travers la littérature.

2

TRAÇABILITÉ DANS LES TRANSFORMATIONS DE MODÈLES

Dès les débuts du génie logiciel, comprendre et assurer le lien entre les différentes phases du développement a toujours été une priorité. Les techniques de *cross-referencing* [134] qui ont pris la forme de matrices, de bases de données ou encore de liens hyper-textes marquent les premiers efforts mis en oeuvre pour garder des liens entre les différents artefacts [67, 73]. Les informations concernant ces liens devaient être générées et maintenues, en grande partie, manuellement. Évidemment, ce mécanisme est vite devenu obsolète et a évolué, mais ces techniques sont néanmoins toujours utilisées pour créer rapidement des dépendances entre les artefacts pendant le développement.

Dans l'ingénierie dirigée par les modèles (IDM), la volonté de garder ces liens tout au long des phases de développement reste une priorité. Dès lors qu'une transformation de modèles a lieu, des éléments sont consommés pour en produire de nouveau. Comprendre dès le premier regard quels sont les éléments qui en ont produit d'autres n'est pas une chose aisée. Pour garder ces informations, il est important de conserver des liens entre les différents éléments, appelés lien de traçabilités. En effet, selon [65] : « *La traçabilité permet d'établir les degrés de parentés entre les produits d'un processus de développement, notamment les produits liés par une relation de prédécesseur-successeur ou de maître-subordonné.* » En IDM, ces informations de « prédécesseur-successeur » sont conservées en utilisant les liens de traçabilité sous forme de modèle. Évidemment, dans le monde de l'IDM, plusieurs relations de traçabilités existent et selon les domaines, ces relations peuvent prendre plusieurs sens, même s'ils lient toujours des artefacts entre eux. Par exemple, dans [38] et [58], les auteurs proposent une trace sous forme de modèles pour l'ingénierie des besoins. Les liens de traçabilité exprimés dans ces travaux établissent des relations entre les besoins et les artefacts créés pour répondre à ses besoins. En revanche, dans [71] où une trace de

transformations de modèles est définie, les liens de traçabilité exprimés permettent de déterminer les artefacts à l'origine de la création d'autres dans une chaîne de transformations.

Dans cette thèse, nous nous concentrons uniquement sur les chaînes de transformations de modèles et tout particulièrement les chaînes de compilation IDM. Dans ce cas, ce sont les traces de transformations de modèles qui sont à l'honneur et, par conséquent, nous ne nous attardons pas sur les différentes traces que l'on peut retrouver dans l'ingénierie des besoins. Pour toutes informations supplémentaires sur la trace dans l'ingénierie des besoins, nous vous renvoyons à un excellent état de l'art proposé dans [137]. À travers les sections suivantes, nous nous attachons à présenter les traces pour les transformations de modèles à modèles et de modèles vers texte, les procédés mis en oeuvre pour leurs générations et leurs sauvegardes ainsi que des exemples de leur utilisation à travers la littérature. Nous présentons ensuite un ensemble d'exigence qu'un mécanisme de trace doit satisfaire pour être à la fois flexible, réutilisable, facile à maintenir et à analyser. Nous clôturons le chapitre en montrant les moyens de gestion de la trace dans les chaînes de transformations.

2.1 TRAÇABILITÉ DE MODÈLES À MODÈLES

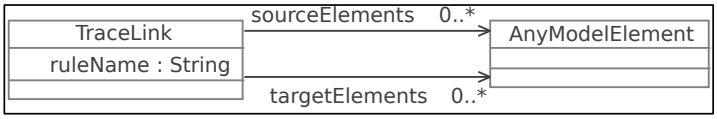
Comme nous l'avons précisé, lors d'une transformation, des éléments sont consommés alors que d'autres sont produits ou modifiés. De cette façon, la trace créée par une transformation de modèles relie des éléments des modèles d'entrée vers des éléments des modèles de sorties. Il faut cependant différencier les transformations qui produisent de nouveaux modèles différents de ceux d'entrées (que nous avons appelées transformations *in* vers *out* au chapitre 1), des transformations qui modifient des modèles d'entrée (appelées transformations *sur place*). En effet, même si les traces produites dans les deux cas possèdent le même sens, dans le cas des transformations *sur place*, la façon de gérer la trace est différente. Dans un premier temps nous allons présenter les traces dans les transformations de modèles *in* vers *out* avant de décrire problèmes engendré par la gestion de la trace dans les transformations *sur place*.

2.1.1 *Trace des transformations de modèles in vers out*

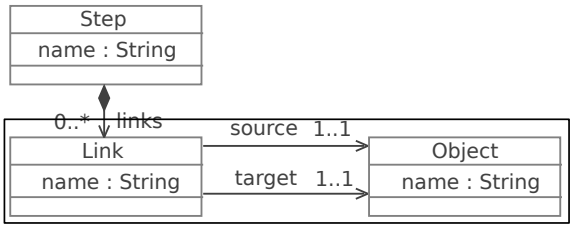
Plusieurs méta-modèles pour gérer la traçabilité des transformations de modèles ont été proposés à travers la littérature. Due à l'intention commune, quelque soit le domaine [1], les méta-modèles de trace dans les transformations de modèles possèdent un noyau structurel assez semblable. Comme exemple, trois méta-modèles de trace sont présentés en figure 8. Le premier, figure 8a, correspond au méta-modèle défini dans [71] par Jouault pour l'ATLAS TRANSFORMATION LANGUAGE (ATL). Le second, figure 8b, correspond au méta-modèle présenté dans [42], proposé pour le langage KERMETA. Finalement, le dernier, figure 8c est une partie du méta-modèle de trace de l'EPSILON MERGING LANGUAGE (EML) [76]. Néanmoins, les différents méta-modèle proposés ont tous leurs propres caractéristiques.

1. On cherche à lier des artefacts entre eux.

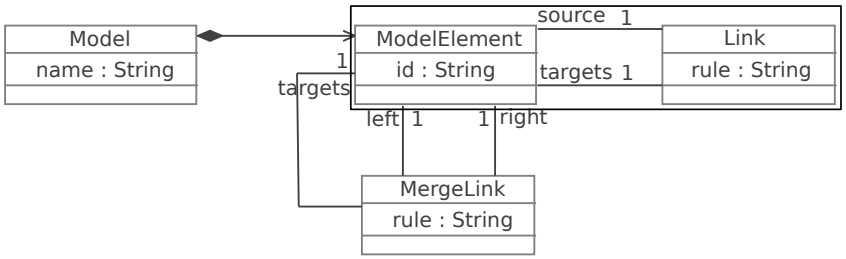
Le noyau commun aux trois méta-modèles est composé de deux méta-classes et de deux références. Il correspond à la partie encadrée dans les figures 8a, 8b et 8c. Dans chacun d'eux, on retrouve un lien de



(a) Méta-modèle de trace ATL [71]



(b) Méta-modèle de trace KERMETA [42]



(c) Extrait du méta-modèle de trace EML [76]

FIGURE 8: Analogie entre les méta-modèles de trace

trace (noté *TraceLink* en figure 8a et *Link* en figure 8b et figure 8c) qui peut être associé à un ou plusieurs (selon le méta-modèle) éléments de modèles. Bien entendu, nous reviendrons plus en détail sur chacun des méta-modèles dans la suite du document.

Comme nous venons de le voir sur cet exemple, les noyaux des méta-modèles de traces que l'on peut trouver à travers la littérature sont assez semblables. En revanche, ces méta-modèles de traces diffèrent, d'une part, par le sens et le type de lien qu'ils tissent entre les modèles manipulés par la transformation et, d'autre part, par la manière dont ils représentent l'information. La manière dont l'information de trace est produite est aussi propre à chaque mécanisme de trace. Ainsi, deux approches sont répertoriées pour les mécanismes de traçabilité : l'approche purement méta-modèle (*pure metamodel approach* et l'approche par marquage de trace (*trace tagging approach*). On peut retrouver cette distinction dans [128].

Approche de traçabilité purement méta-modèle

Par purement méta-modèle, il faut entendre « avec des liens de traçabilité spécifique » pour chaque type d'artefact différent. Selon cette approche, il n'existe donc pas un méta-modèle de trace unique permettant de tracer tous les types de transformations, mais un méta-modèle de trace qui est construit pour un ensemble de méta-modèles d'entrée et de sortie connus à l'avance. Les méta-modèles de trace utilisés dans ces approches décrivent de manière explicite les liens qui vont être tracés entre les éléments des différents méta-modèles. Pour définir un méta-modèle de trace selon cette approche, les éléments représentant les « liens » sont donc directement tissés dans le méta-

modèle de trace entre plusieurs éléments des méta-modèles d'entrée et de sortie.

La figure 9 montre un exemple d'un méta-modèle de trace d'une approche purement méta-modèle présenté par Paige et Al. [36]. Trois méta-modèles y sont représentés : *ClassMetamodel*, *ComponentMetamodel* et *ComponentClassTraceMetamodel*. Le méta-modèle de trace proposé (*ComponentClassTraceMetamodel*) est totalement dédié aux méta-modèles *ComponentMetamodel* et *ClassMetamodel* (qui correspondent, respectivement, aux méta-modèles d'entrée et de sortie de la transformation). Il comprend exactement deux liens de trace spécifiques : *ComponentPackageTraceLink* et *ServiceMethodTraceLink* et ne s'intéresse pas aux autres concepts manipulés des méta-modèles d'entrée et de sortie. Le premier lien de trace *ComponentPackageTraceLink* est défini exclusivement entre les concepts *Package* et *Component*, alors que le second lien *ServiceMethodTraceLink* est défini exclusivement entre les concepts *Method* et *Service*.

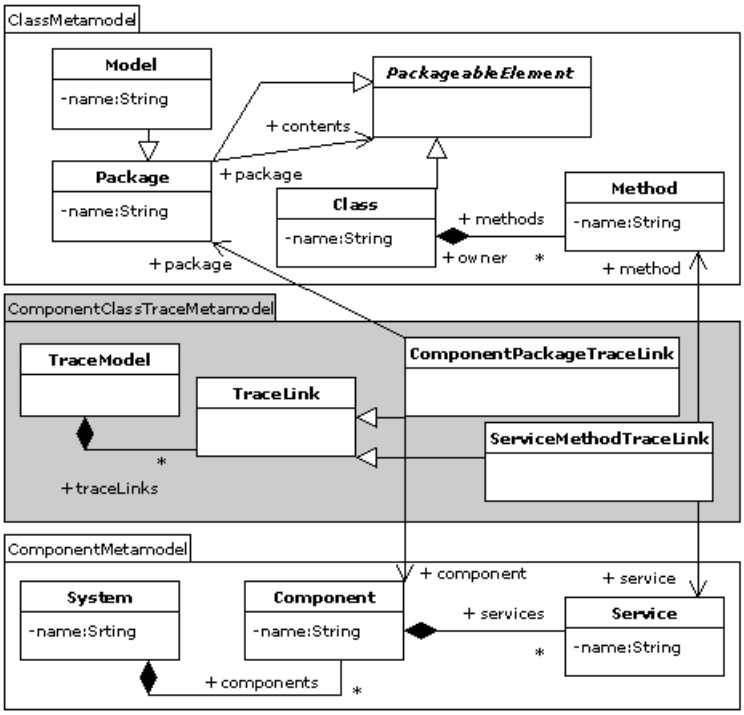


FIGURE 9: Exemple de méta-modèle de trace reposant sur l'approche purement méta-modèle [36]

Ainsi, lorsqu'une trace suivant l'approche purement méta-modèle est produite, un type de lien différent est tissé entre les instances de *Package* et de *Component* et les instances de *Method* et *Service*. De cette façon, les liens de trace tissés entre les éléments sont fortement typés et possèdent une sémantique forte. De plus, l'utilisation d'un tel type de trace permet de facilement maintenir et analyser les liens de traçabilité tissés.

Un tel méta-modèle de trace se construit grâce à un langage de méta-modélisation : *Traceability Metamodelling Language* (TML) qui est présenté en figure 10. Les traces produites à partir de TML contiennent des liens de traces (*TraceLink*) qui peuvent être liés à plusieurs éléments via des *TraceLinkEnd*. Ces liens de traçabilités ont la possibilité de

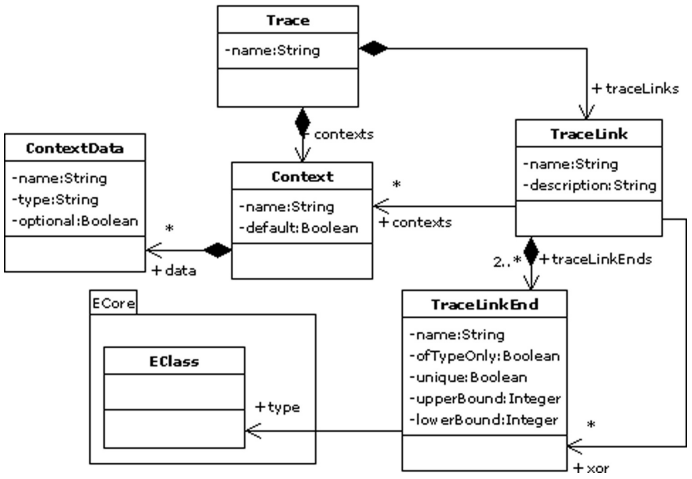


FIGURE 10: Méta-modèle TML [36]

contenir diverses informations liées au contexte de la transformation (représenté par les méta-classes *Context* et *ContextData*).

Il est certain que l'utilisation d'une telle trace facilite grandement son analyse et sa maintenance. Néanmoins, ce type de trace nécessite obligatoirement de connaître à l'avance les méta-modèles d'entrée et de sortie de la transformation. De plus, le méta-modèle de trace est produit pour une transformation en particulier. Pour que ce méta-modèle de trace soit réutilisable, il faut que la transformation soit définie en fonction des mêmes méta-modèles et pour les mêmes éléments. De plus, si le méta-modèle d'entrée ou de sortie évolue ainsi que la transformation, il faut redéfinir le méta-modèle de trace pour prendre en compte la liaisons de nouveaux éléments.

Approche de traçabilité par marquage de trace

Afin de construire un unique méta-modèle de trace facilement réutilisable mais possédant des liens avec une sémantique plus faible, il existe des méta-modèles de trace utilisant une approche par marquage de trace (*trace tagging*) qui fournissent une plus grande flexibilité quant aux captures des liens de traçabilités.

L'approche par marquage de trace est l'approche la plus souvent utilisée dès lors que l'on parle de transformation de modèles à modèles. Dans cette section, nous allons montrer quelques exemples de méta-modèle de trace faisant référence aux langages de transformation de modèles les plus utilisés dans la communauté IDM.

LA TRAÇABILITÉ DANS ATL Parmi les différents méta-modèles de trace de transformations de modèles proposés à travers la littérature, celui imaginé par Jouault est minimal. Ce méta-modèle a été implémenté pour le langage de transformation ATL [71] et fournit une « base » à la définition de nouveaux méta-modèles de trace pour les transformations de modèles de par sa simplicité.

La figure 11 présente à nouveau le méta-modèle de trace proposé pour ATL. Il est composé de deux méta-classes. La première représente le lien entre les éléments (*TraceLink*) tandis que la seconde représente un élément quelconque du modèle d'entrée ou de sortie (*AnyModelElement*).

Les relations *sourceElements* et *targetElements* indiquent que les liens de traces sont tissés entre plusieurs éléments des modèles d'entrée et de sortie. Les liens de traces portent un attribut *ruleName* permettant de préciser quelle est la règle qui a consommé les éléments pointés par *sourceElements* et créé ceux pointés par *targetElements*. Ainsi, un lien de trace indique que un ou plusieurs éléments des modèles d'entrée ont créé un ou plusieurs éléments des modèles de sortie par l'exécution d'une règle.

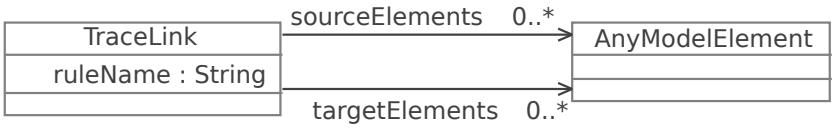


FIGURE 11: Méta-modèle de trace pour ATL [71]

Il est à noter ici que la méta-classe *AnyModelElement* permet d'indiquer que les liens de traces peuvent lier n'importe quels éléments instance des méta-modèle d'entrée et de sortie. Ce point est important car il différencie les traces basées sur le méta-modèle de Jouault des traces produites dans [36]. En effet, contrairement à la trace suivant l'approche purement méta-modèle proposée par Paige et Al., le lien de trace entre deux éléments possède la même sémantique quelque soit les éléments qu'il lie. Il n'est donc pas nécessaire de produire un méta-modèle spécialement dédié à une transformation.

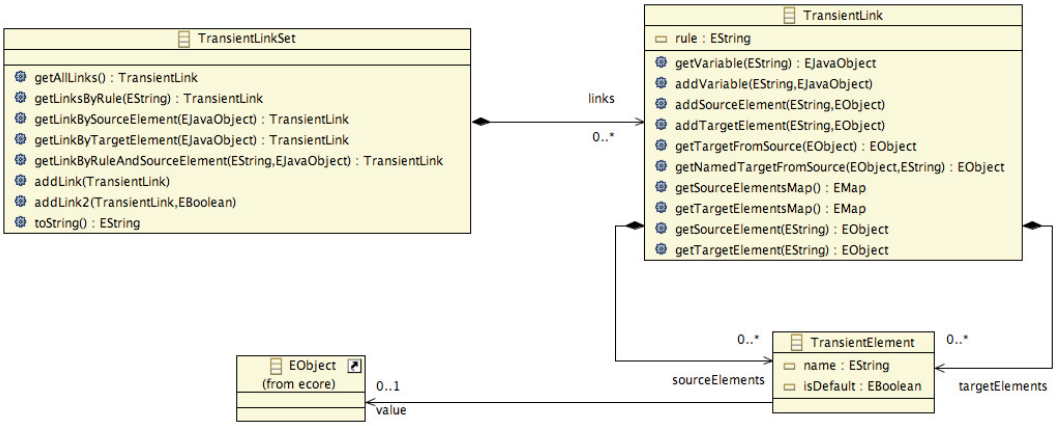


FIGURE 12: Méta-modèle de trace avancé pour ATL [140]

Plus récemment, un autre méta-modèle de trace à été proposé pour ATL dans [140]. Ce méta-modèle est inspiré par le méta-modèle de Jouault comme base et présente la particularité de posséder plusieurs méthodes facilitant son interrogation et sa maintenance. Le méta-modèle est présenté en figure 12. On remarque que la méta-classe *TraceLink* a été renommée en *TransientLink* et que des méthodes lui ont été ajoutées pour récupérer et ajouter facilement des informations dans la trace. Pour stocker l'ensemble des liens tissés lors d'une transformation, une méta-classe *TransientLinkSet* a été ajoutée. Elle aussi possède des méthodes qui lui permettent de gérer au mieux les liens qu'elle contient (pour effectuer des recherches ou des ajouts). L'autre différence majeure entre le méta-modèle proposé par Jouault et celui présenté en figure 12 réside dans l'utilisation d'une méta-classe *TransientElement*. Cette méta-classe correspond à une représentation interne d'ATL par

laquelle le moteur de transformation considère les instances manipulées par la transformation à un moment donné. Ces *TransientElements* font donc office d'une représentation d'un élément des modèles d'entrée ou de sortie de la transformation.

En résumé, le coeur de la trace proposée par Yie est similaire à la trace proposée par Jouault. Les grosses différences se situent dans les méthodes ajoutées aux méta-classes ainsi que dans l'utilisation d'une méta-classe comme représentation des éléments manipulés par la transformation. Nous verrons en section 2.1.3 que le mécanisme de trace proposée par Yie se différencie sur d'autres points, plus relatif à la génération de la trace qu'à sa structure.

LA TRAÇABILITÉ DANS KERMETA La traçabilité proposée pour le langage KERMETA [42] amène une façon très singulière de traiter les liens de trace. La figure 13 représente le méta-modèle de trace du langage KERMETA, déjà montré en figure 8b.

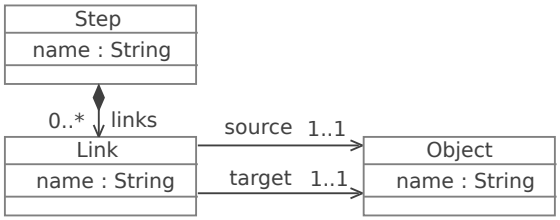


FIGURE 13: Méta-modèle de trace KERMETA [42]

En effet, si l'on regarde en détail la cardinalité des références *source* et *target* partant du lien de trace *Link* vers les éléments tracés, on se rend compte qu'un lien de trace associe seulement un élément source à un élément destination. Pour représenter la création par la transformation d'un élément par plusieurs éléments, un lien est créé par éléments. La figure 14 montre sur un exemple simple comment les différents type de relations sont gérés. Les éléments *e1*, *e2*, *e3* et *e4* représente des éléments du modèle d'entrée, alors que les éléments *s1*, *s2*, *s3* et *s4* représentent des éléments du modèle de sorties. Les éléments sont transformés par 3 règles : *r1*, *r2* et *r3*.

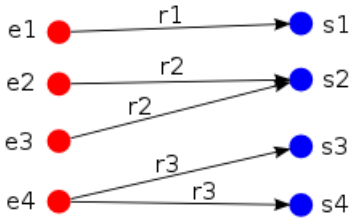


FIGURE 14: Exemple des liens gérés par la trace KERMETA [42]

En suivant les liens de trace présentés en figure 14, on en déduit que l'élément s1 a été créé à partir de l'élément e1 via la règle r1. De même, l'élément *s2* a été créé à partir des éléments *e2* et *e3* via la règle r3 et l'élément *e4* a produit deux éléments : s3 et s4 via r3.

Cependant, cette façon de gérer la trace pose tout de même quelques problèmes. Si un élément du modèle d'entrée est réutilisé plusieurs fois

par une même règle de transformation, il est difficile de distinguer les éléments qui sont créés par l'une ou l'autre exécution de la règle. Dans l'exemple figure 14, il est difficile de savoir si s3 et s4 on été créés par la même exécution d'une règle ou par deux exécutions séparées de la même règle sur le même élément. De la même façon, pour e2 et e3, il est difficile de déterminer : si e2 a créé s2 par la règle r2, puis e3 a aussi mené à la création de s2 par e3, écrasant cette dernière puisque s2 avait déjà été créé. Ou bien s'il faut obligatoirement e2 et e3 pour produire s2 en utilisant la règle r2. Par conséquent, il est assez complexe de maintenir une telle trace.

REMARQUE. Le méta-modèle de trace pour kermeta comprend aussi une méta-classe *step*. Nous reviendrons sur celle-ci en section 2.3.

LA TRAÇABILITÉ DANS EML Le méta-modèle de trace EML, représenté en figure 15, est un méta-modèle de trace contenant des concepts en plus du noyau structurel présenté en section 2.1.1.

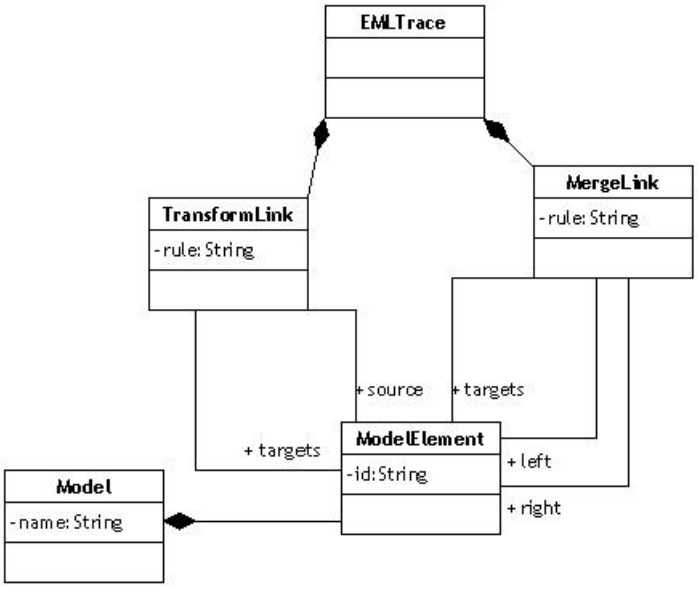


FIGURE 15: Méta-modèle de trace EML [76]

Le méta-modèle de trace EML ne compte que trois méta-classes : *EMLTrace*, *TransformLink* et *MergeLink*. Les deux autres méta-classes : *Model* et *ModelElement* sont des concepts importés du méta-modèle EML, représentant un modèle et un élément de modèle (un peu à l'image du concept *AnyModelElement* dans le méta-modèle de trace ATL et du concept *Object* dans le méta-modèle de trace KERMETA). Par conséquent, ce ne sont pas des concepts exclusifs de la trace. La méta-classe *EMLTrace* représente la racine d'un modèle de trace EML. Celle-ci peut contenir différents liens possédant une sémantique différente. Le premier, *TransformLink*, correspond à un lien de trace entre un élément *source* et plusieurs éléments destination *targets* produits par une règle de transformation. Le second lien, *MergeLink*, correspond à un lien de trace produit par une règle de composition entre deux éléments *left* et *right* vers plusieurs éléments destination *targets*. Ce méta-modèle possède donc un concept en plus du concept *TransformLink* (concept équivalent

à *TraceLink* du méta-modèle ATL et à *Link* du méta-modèle KERMETA), un concept *MergeLink* dédié à la trace des règle de composition. Grâce à ce lien dédié, il est possible de différencier les deux opérandes (*left* et *right*) de l'opération de composition.

Il est à noter, sur le méta-modèle de trace EML, que les références *source* et *targets* sont navigables dans les deux sens, alors que les références équivalentes des deux autres méta-modèles précédemment présentés ne le sont pas. La méta-classe *ModelElement*, importée du méta-modèle de EML (figure 15) a donc été modifiée pour pouvoir proposer cette navigation de retour. Même si cet ajout possède un réel avantage, il limite la réutilisation du méta-modèle pour un autre langage de transformations. En effet, puisque la méta-classe importée du méta-modèle EML a été modifiée pour ajouter une référence vers la trace, cela signifie que le méta-modèle de trace est dépendant du méta-modèle EML et d'éventuels algorithmes écrits pour ce méta-modèle de trace ne peuvent pas être réutilisés avec un autre langage de transformation.

LA TRAÇABILITÉ DANS QVT Le langage de transformation de modèles QVT est le standard proposé par l'OMG. Comme nous l'avons évoqué au chapitre 1, QVT est composé de trois sous langages : le langage RELATION, le OPERATIONAL MAPPING et le CORE. Les traces produites pour chacun de ces sous langages conservent néanmoins, plus ou moins, les mêmes informations [98]. Étant donné que le standard QVT ne propose pas de structure pour la trace (pas de méta-modèle de trace), celle-ci est laissée libre à l'implémentation. Ainsi, on trouve un méta-modèle de trace différent pour chaque implémentation du standard. Chacun des deux implémentations du sous langage OPERATIONAL MAPPING (QVTo et SMARTQVT) propose un méta-modèle de trace différent, mais l'utilisation qui en est faite par le moteur reste la même.

La trace est utilisée dans QVT au cours des transformations. Elle sert notamment de mémoire pour pouvoir identifier les éléments sur lesquels une règle a déjà été exécutée et résoudre ainsi des références.

Dans cette section, nous montrons une implémentation proposée pour le méta-modèle de trace QVTo en figure 16. Le méta-modèle de trace proposé pour QVTo est plutôt complexe. Il contient 11 méta-classes et importe 2 méta-classes de méta-modèle annexes (*MappingOperation* et *EObject*). La méta-classe *Trace* est la racine de la trace produite par une transformation QVTo. Elle peut contenir plusieurs enregistrements de traces *TraceRecord*. Un enregistrement de trace correspond à l'exécution d'une règle de MAPPING du langage OPERATIONAL MAPPING. Il lie donc une règle de MAPPING (*EMappingOperation*) à ses paramètres (*EMappingParameter*), à son résultat (*EMappingResult*) et à son contexte (*EMappingContext*). Les *EMappingParameters*, *EMappingResults* et *EMappingContexts* sont liés à leur valeurs *VarParameterValue*. Á partir de cette méta-classe, il est possible de récupérer l'élément transformé en naviguant à travers les méta-classes *ETuplePartValue* ou *EValue* qui font références à un *EObject*. Dans ce méta-modèle, les références aux éléments sources et destinations des méta-modèles manipulés par la transformation se font par l'intermédiaire des *EMappingParameters* et des *EMappingResults*. Les méta-classes *ObjectToTraceRecordMapEntry* et *MappingOperationToTraceRecordMapEntry* quant à elles servent d'index dans la trace. Elles permettent de mettre en correspondance et

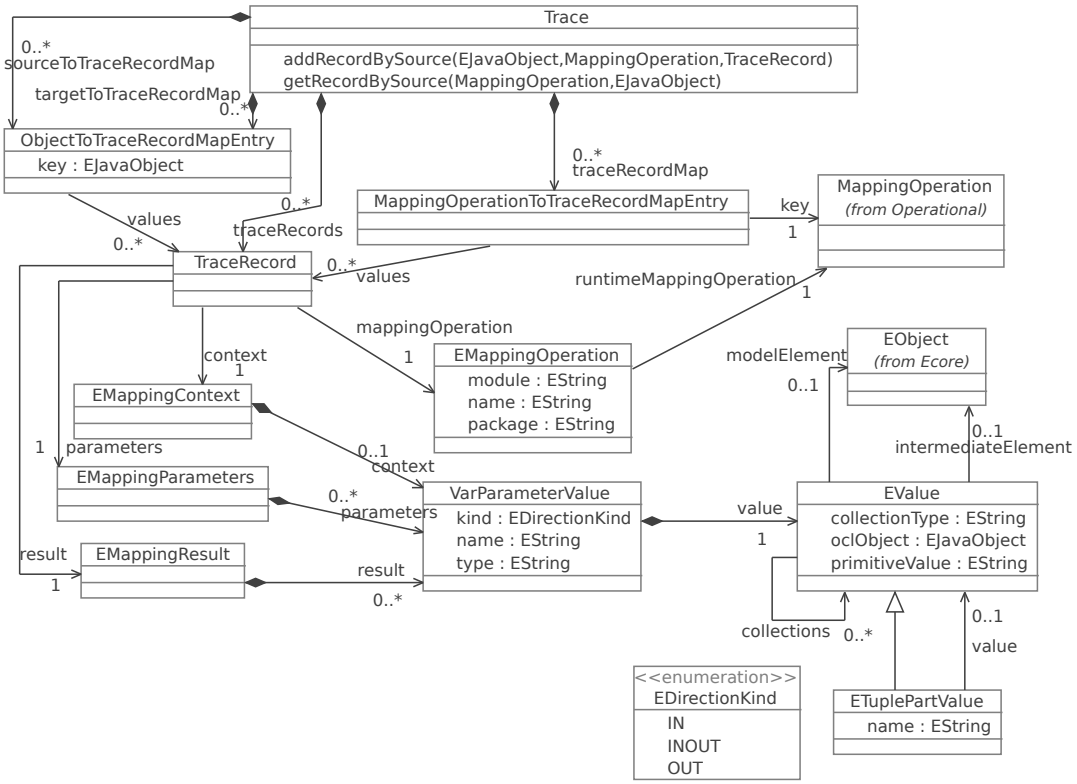


FIGURE 16: Méta-modèle de trace implémenté dans QVTo

de retrouver facilement tous les éléments produits par une opération de mapping (via *MappingOperationToTraceRecordMapEntry*) et tous les éléments sources et destinations créés au cours de la transformation (via *ObjectToTraceRecordMapEntry*).

ETRACE Jusqu'à présent les méta-modèles que nous avons vus étaient dédiés à un langage de transformation donné. Dans [3], Amar et al. proposent un méta-modèle de trace utilisé pour tracer les transformations à caractères impératifs. Ce méta-modèle est représenté en figure 17. Nous pouvons voir que le concept de lien a été fortement affiné. En effet, un pattern composite a été utilisé (méta-classes *AbstractLink*, *Link* et *CompositeLink*) pour pouvoir représenter une succession hiérarchique de liens. Les liens, sont, à l'image du méta-modèle de trace pour KERMETA, liés à un seul élément source et destination, ce qui pourrait impliquer les mêmes problèmes de maintenance, mais l'ajout de la méta-classe *LinkType* peut aider à lever les ambiguïtés en utilisant, par exemple, un numéro représentant l'exécution d'une règle. Ainsi, si plusieurs liens possède le même numéro, cela signifie que c'est la même exécution de la règle qui a mené à la création des éléments qu'ils lient. Cette méta-classe embarque plusieurs attributs (*description*, *purpose*, *uses*, *example*, *name*, *sourceOrigin* et *targetOrigin*) qui vont permettre de spécifier un peu plus le type de lien qui est tissé entre deux éléments.

Dans cette section, nous avons montré les différents méta-modèle de trace existant les plus manipulés ou possédant des caractéristiques particulières.

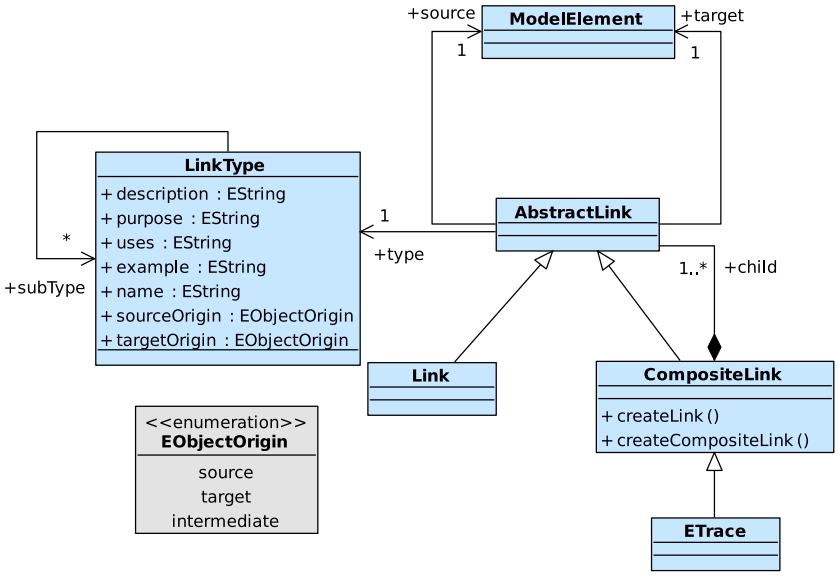


FIGURE 17: Méta-modèle de trace ETRACE [3]

2.1.2 *Trace des transformations sur place*

Parler d'une trace différente pour les transformations *sur place* serait maladroit. Cela supposerait que les méta-modèles de trace présentés précédemment ne sont plus fonctionnels dès lors que l'on parle de transformations *sur place*, ce qui n'est pas le cas. Dans cette section, nous essayons surtout d'attirer l'attention sur les problèmes de maintenance et de recherche d'informations induits par la trace lorsque l'on traite avec des transformations *sur place*.

L'un des soucis majeurs des traces produites pour les transformations *sur place* est sa représentation. Pour mémoire, une transformation *sur place* modifie un modèle existant et n'en crée pas un nouveau. Le lien de trace qui est donc tissé s'effectue non pas entre des éléments des modèles d'entrée et de sortie distincts, mais entre des éléments du même modèle. Lorsqu'un élément est modifié dans le modèle d'entrée, le lien de trace désigne le même élément comme source et destination du lien. Dans ce cadre et notamment pour les modification effectuées sur des attributs, il est difficile de percevoir quel était la valeur de l'attribut avant la transformation.

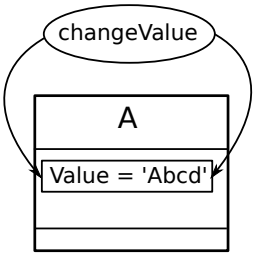


FIGURE 18: Problème d'interprétation des traces *inout*

Ce problème est illustré en figure 18. Il est visuellement représenté un simple élément *A* et un lien de trace nommé *changeValue*. Le lien de trace tissé nous indique que l'élément *A* a été modifié, plus précisément

que son attribut *Value* a été changé via la règle de transformation *changeValue*. Cependant, il est impossible de savoir quelle modification a été faite sur l'élément si l'état du modèle avant la transformation et l'état du modèle après la transformation ne sont pas conservés. Dès lors que l'on travaille avec des transformations *sur place*, il devient alors complexe d'interpréter la trace produite puisqu'il est impossible de déterminer les modifications exactes qui ont été effectuées sur le modèle.

En utilisant la trace produite par [3], il est possible de donner une première réponse à ce problème. En effet, grâce aux informations additionnelles retenues par le concept de *LinkType* [2], les auteurs peuvent conserver les valeurs des attributs modifiés.

2. *c.f*, figure 17.

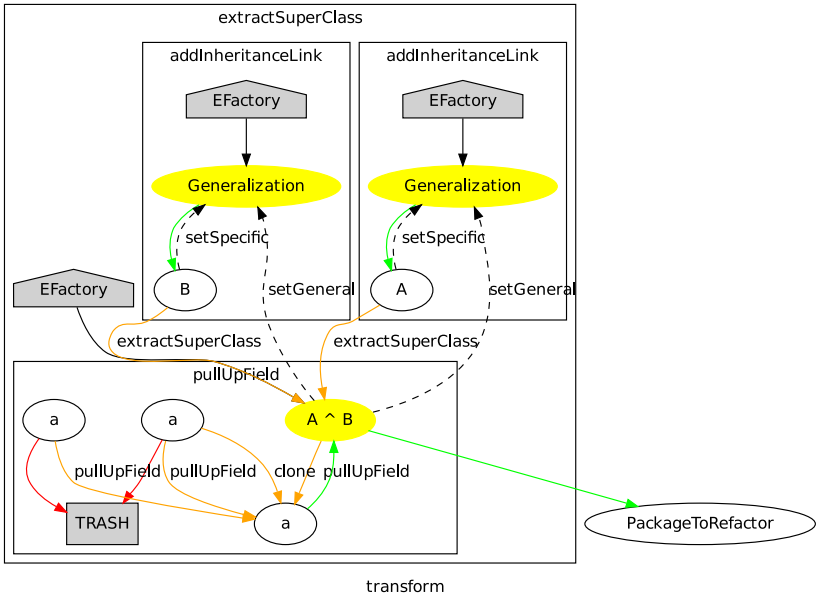


FIGURE 19: Exemple de ETRACE produite pour une transformation de *refactoring* [3]

Un exemple de leur trace pour une transformation de *sur place* est représenté en figure 19. La trace présentée est récupérée après exécution d'une transformation *sur place* consistant à détecter les paires de classes contenant des attributs identiques puis à créer une super classe commune où les attributs identiques sont déplacés [3]. En regardant la trace, il est possible de retrouver l'exécution de la transformation. Les concepts grisés *TRASH* et *EFactory* représentent des concepts spécifiques qui n'appartiennent pas au modèle modifié. Ils représentent respectivement la suppression et la création d'un concept. Ainsi, tout lien rentrant dans *TRASH* indique une suppression et tout lien sortant de *EFactory* indique une création. On peut déjà observer que deux éléments *a* [4], sont supprimés et ont servi à la création, via la règle *pullUpFields*, d'un nouvel attribut *a*. On peut aussi voir que la règle *extractSuperClass* s'est servie des éléments *A* et *B* pour créer l'élément *A^B*. Finalement, on voit que deux *Generalization* [5] sont créées et qu'elles relient les éléments *A* et *B* avec la super classe créée *A^B*. Ce sont les règles *setSpecific* et *setGeneral* qui permettent d'établir ces liaisons entre les *Generalizations* et les éléments *A* et *B*.

3. Proposé par Fowler et al. dans [49]

4. Correspondant aux attributs identiques.

5. Les *Generalization* représente la relation d'héritage.

Cependant, la lecture d'une telle trace est vraiment complexe sur un exemple simple. Lors d'un passage à l'échelle, si aucun algorithme pour faciliter la lecture de la trace n'est présent, cela limite grandement son utilisation. La gestion et la récupération des informations dans une transformation *sur place* pose réellement un problème.

Finalement, à travers la littérature, seul le méta-modèle de trace proposé par [3] propose une solution à la gestion de la trace dans les transformations *sur place*, mais cette solution reste difficile à exploiter.

2.1.3 *La capture des liens de traçabilité*

Le méta-modèle de trace utilisé est le point central d'un mécanisme de traçabilité, mais il est également important de s'intéresser à la façon dont la trace sera produite durant l'exécution de la transformation. Dans cette section, nous présentons et référençons les différentes techniques permettant d'obtenir des informations de traçabilité. Ces techniques reposent sur l'utilisation des liens de traçabilité explicite et des liens traçabilité implicite d'une transformation.

Liens de traçabilité explicite

On parle de lien de traçabilité explicite lorsque l'on spécifie explicitement les liens de traçabilité qui seront construits pendant la transformation. Les règles consommant et créant les éléments dont on veut garder la trace sont donc modifiées pour gérer la trace au moment de leurs exécutions. Cette technique est notamment mise en œuvre dans [71] et [42].

Le listing 2.1 présente un exemple de règle de transformation ATL dans laquelle la production d'un lien de traçabilité a été insérée (ligne 11 à 18). La ligne 2 montre que le méta-modèle de trace[6] est explicitement passée comme paramètre de sortie de la transformation. La règle *A2BPlusTrace* (ligne 4 à 19) produit un élément *t* de type *B* (ligne 8) à partir d'un élément *s* de type *A* (ligne 6). L'élément *t* porte le même nom que l'élément *s* (ligne 9). Dans cette règle de transformation simple, les instructions permettant la construction d'un lien de trace *traceLink* ont été insérées aux lignes 11 à 14 et aux lignes 16 à 17. Les lignes 11 à 14 construisent un lien de trace de type *TraceLink* en précisant le nom de la règle tracée (*A2BPlusTrace*) grâce à son attribut *ruleName* (ligne 12) et en pointant *t* comme élément de destination (ligne 13). Les lignes 16 et 17, quant à elles, pointent l'élément *s* comme élément source.

6. Méta-modèle de trace ATL, *c.f,* figure 11.

```
1  module Src2DstPlusTrace;
2  create OUT : Dst, trace : Trace from IN : Src;
3
4  rule A2BPlusTrace {
5   from
6    s : Src!A
7   to
8    t : Dst!B (
9     name <- s.name
10   ),
11   traceLink : Trace!TraceLink (
12    ruleName <- 'A2BPlusTrace',
13    targetElements <- Sequence {t}
14   )
15  do {
```

```
16        traceLink.refSetValue('sourceElements',
17             Sequence {s});
18    }
19  }
```

Listing 2.1: Production d'un lien de trace ATL [71]

Ces morceaux de code insérés dans les transformations sont ajoutés manuellement, ce qui rend l'opération fastidieuse et limite la maintenance et l'évolution de la transformation. Aussi, une technique visant à automatiser l'insertion de ces fragments de code a été proposée par Jouault dans [71]. Elle repose sur l'utilisation de transformation du premier ordre, les *High Order Transformation* (*HOTs*) [121]. Ces transformations considèrent le langage de transformation de modèle comme un simple modèle (conforme au méta-modèle du langage de transformation) qu'il est possible de manipuler. Ainsi, il est aisé d'ajouter, supprimer ou modifier les concepts d'une transformation en utilisant une transformation de modèle.

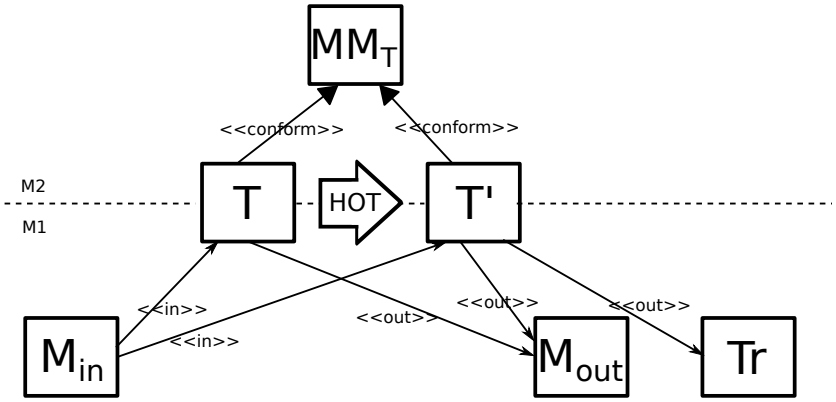


FIGURE 20: Insertions des liens de trace par *HOT*

La figure 20 illustre l'approche avec une transformation T transformée via une *HOT* en T′. Deux niveaux d'abstractions *M1* et *M2* sont représentés[7] La transformation initiale T est conforme à un méta-modèle de transformation MM_T. La transformation T′, embarquant les fragments de code concernant la trace, est produite grâce à une *HOT* et, est conforme au même méta-modèle MM_T. Alors que la transformation initiale prend un modèle d'entrée M_{in} et crée un modèle de sortie M_{out}, pour le même modèle d'entrée M_{in}, la transformation T′ produite par la *HOT* crée deux modèles de sorties : M_{out}, comme la transformation T, mais aussi un modèle de trace Tr. L'insertion des fragments de code concernant la trace est effectuée par une *HOT*, considérée comme une transformation *sur place*, sur la transformation à modifier[8]. Même si l'insertion des fragments de code est automatisée, il n'en reste pas moins le problème de l'évolution de la transformation. Si celle-ci est modifiée, il faut de nouveau appliquer la *HOT* pour ajouter la construction des liens de trace automatiquement. De plus, pour pouvoir appliquer une telle technique, il faut pouvoir avoir accès au modèle de la transformation et profiter d'une synchronisation entre ce modèle de la transformation et sa syntaxe textuelle afin de pouvoir jouer la transformation une fois modifiée. Si aucune synchronisation entre le modèle de la transformation et sa forme textuelle n'existe, il faut au

7. Le niveau modèle et méta-modèle.

8. Nous n'avons pas représenté la *HOT* comme une transformation *sur place* sur la figure 20 pour des raisons de lisibilité.

moins pouvoir exécuter la transformation en utilisant directement son modèle.

L'ajout de la construction de la trace directement dans le code des transformations possède l'avantage de fonctionner aussi bien pour la traçabilité orientée purement méta-modèle que la traçabilité orientée marquage de traces. En effet, il est possible d'ajouter les fragments de code uniquement pour certaines règles de transformation en créant un type de lien particulier et ainsi orienter la production de la trace. Cependant, cette technique est particulièrement longue à mettre en œuvre manuellement et requiert une bonne connaissance de la transformation à tracer.

Traçabilité implicite

Par opposition à la traçabilité explicite où la production de la trace est directement ajoutée dans la transformation à tracer, la traçabilité implicite propose de tisser les liens de trace sans intrusion dans la transformation. Ainsi, la transformation ne requiert pas de modification particulière pour pouvoir bénéficier du support de la trace, celle-ci étant produite directement par le moteur de transformation.

Dans la section précédente, nous présentions comment la trace était produite pour le langage de transformation ATL. La production était basée sur une traçabilité explicite. Plus récemment, toujours pour ATL, Yie a proposé dans [140], une manière alternative pour produire la trace durant l'exécution d'une transformation. Cette fois, ce sont les liens de traçabilités implicite qui sont visés. Les informations nécessaires à la construction de la trace sont récupérées directement à partir du moteur de transformation à partir du bytecode ATL produit. En effet, selon les séquences d'instructions présentes dans le bytecode ATL, il est possible d'identifier les informations nécessaires à la trace tel que le nom de la règle utilisée et les éléments manipulés en entrée et sortie de la règle. De cette façon, les liens de traçabilités sont directement et automatiquement capturés pendant l'exécution de la transformation. En revanche, ce procédé de capture par scrutation du bytecode à l'exécution de la transformation se retrouve dans peu de langages de transformation. En effet, peu d'entre eux sont traduit vers un langage type bytecode.

Une autre approche proposée en [3] utilise la programmation orientée aspect pour la capture des traces implicites d'une transformation. La programmation orientée aspect permet de séparer les préoccupations transverses en plaçant les services comme des « aspects » du programme principal qu'il faut lui intégrer [74]. Ce paradigme de programmation peut s'adapter à tout langage, sous réserve de posséder un tisseur d'aspect permettant d'intégrer automatiquement les aspects au programme principal. Ce tissage d'aspect s'effectue en fonction de *pointcuts* précisant à quels endroits du programme le service en question doit être intégré. La figure 21 illustre l'architecture générale de l'outil *ETraceTool* reposant sur la programmation par aspects proposé par [3].

L'environnement impératif de programmation se compose d'une transformation écrite en JAVA utilisant le framework EMF. Les différents évènements de transformation, comme la création d'éléments, sont capturés par l'*Aspect Tracer* qui construit un modèle de trace conforme au méta-modèle *Nested Trace Metamodel*. La trace produite peut être ensuite visualisée grâce à un graphe annoté et/ou sauvegardée dans un modèle XMI.

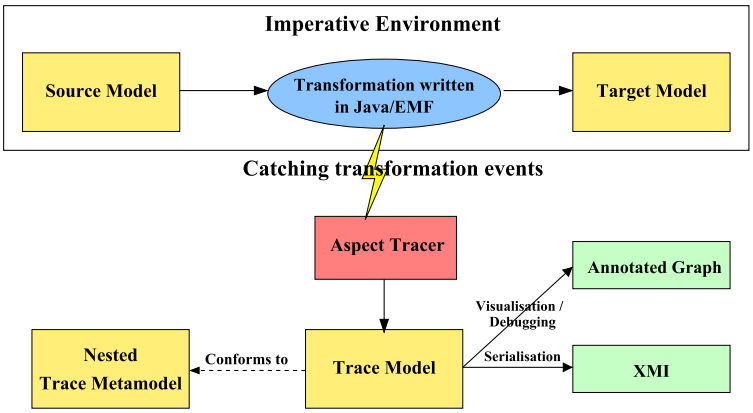


FIGURE 21: Architecture générale de ETraceTool [3]

En utilisant ce paradigme de programmation, la trace est vue comme un service additionnel qui est ajouté au programme principal. L'idée proposée dans [3] est assez semblable à celle proposée dans la partie relatant des liens de trace explicite et de l'utilisation des *HOTs* à l'exception de l'étape de tissage qui s'effectue ici sur le code JAVA compilé. Cependant, cette technique requiert l'utilisation d'un tisseur d'aspect qui n'existe pas, pour le moment, pour les langages de transformation de modèles dédiés (comme ATL ou QVT). Les auteurs reposent alors sur une transformation en Java pour pouvoir bénéficier d'outils existants.

2.1.4 *Sauver une trace*

La sauvegarde des liens de traçabilités obtenus lors d'une transformation est primordiale. C'est cette étape qui permet la réutilisation de la trace par un algorithme *ad-hoc*. Deux approches se distinguent pour la conservation du modèle de trace produit.

La première propose d'*annoter* les modèles d'entrée et/ou de sorties pour profiter des informations de trace directement sur les artefacts manipulés par la transformation. Par exemple, pour représenter un raffinement dans un modèle UML, une association stéréotypée «refines» lie les deux éléments concernés [61]. Cette approche est fréquente dans les outils de modélisation car elle est facilement compréhensible par les humains et car elle représente les liens de traçabilité par des éléments de modèle visuels. Cependant, cette approche peut uniquement représenter des liens de traçabilité intra-modèles. En effet, il est difficile de tisser un lien entre deux éléments de modèles conformes à des méta-modèles différents en embarquant le lien de trace dans l'un ou l'autre des modèles. De plus, l'ajout systématique de liens de traçabilité à l'intérieur d'un seul modèle entraîne, indubitablement, sa « pollution » [34], ce qui peut nuire à sa lisibilité et à sa compréhension.

La seconde approche se concentre sur l'*externalisation* du modèle de trace. Ainsi, lorsqu'un modèle de trace est produit, celui-ci est sauvegardé dans un modèle à part référençant les éléments des modèles d'entrée et de sortie. De plus, la sauvegarde dans un modèle externe permet à une application tierce de facilement réutiliser la trace produite. Cette façon de procéder, très répandue, se retrouve dans les mécanismes de traces tel que ceux présentés dans les sections précé-

dentes. Cependant, sans outils dédiés permettant une interrogation ou une visualisation de la trace, celle-ci est assez peu compréhensible par un humain. Cette façon de procéder a pour avantage d'être plus simple à gérer tout en laissant les modèles sources et destination de la transformation inchangés.

Dans [76], les auteurs proposent d'utiliser les deux approches ci-dessus pour en récupérer leurs avantages. Les liens de traçabilités sont donc conservés dans un modèle externe, mais peuvent être composés à la demande avec les modèles d'entrée et de sortie de la transformation. Pratiquement, les modèles d'entrée et de sortie sont composés pour donner un nouveau modèle, puis des annotations comprenant les informations de trace lui sont ajoutées.

2.1.5 *Utilisations classiques des traces*

Obtenir et conserver une trace est une étape, certes, indispensable, mais qui possède peu d'utilité si celle-ci n'est pas manipulée et réutilisée. Nous présentons ici les applications pour lesquelles les traces de transformations de modèles présentées sont utilisées.

Aide au debug de transformations

Il arrive qu'une transformation contienne des erreurs introduites lors de son écriture. Dans ce cas, il faut chercher l'erreur dans la transformation, ce qui peut vite devenir un cauchemar lorsque les transformations sont complexes. Les traces de transformations de données peuvent être vues comme un historique de l'exécution de la transformation. Ainsi, il est possible de s'en servir comme une première aide afin de saisir les actions effectuées par la transformation et, de plus simplement comprendre les actions effectuée par une transformations de modèles. Ce sont les buts premiers que l'on trouve pour les mécanismes de trace présentés en [71], [42] et [76]. Bien évidemment, les autres traces présentées peuvent très bien s'acquitter de cette tâche, surtout celle présentée en [3] qui revendique son affinement du concept lien comme un avantage pour pouvoir effectuer des actions semblables à celles d'un *debugger*[9].

9. Entre autre, le pas-à-pas.

Aide à la composition de vues de modèles

La trace présentée en [140] a été utilisée comme aide à la composition de différentes vues de modèles dans un environnement de modélisation multi-vues[10] [139]. Pour déterminer les concepts communs aux différents formalismes, un modèle de correspondance entre les différentes vues est maintenu. Dans ce contexte, si les différents modèles de vues sont exprimés dans des formalismes différents, il est alors difficile de les composer. Une solution proposée par les auteurs pour éviter une composition entre formalismes différents est de chercher un formalisme commun dans lequel les modèles sont transformés. Ainsi, effectuer la composition au niveau du formalisme commun revient à effectuer une composition homogène[11]. Une fois le formalisme commun déterminé, le challenge est donc de produire automatiquement le nouveau modèle de correspondance, indiquant les éléments communs aux différentes vues exprimées dans ce nouveau formalisme. C'est la trace qui est utilisée pour effectuer cette tâche. Elle permet aux auteurs de générer une transformation permettant de créer, à partir du modèle

10. Permet de décrire un système en développement sous différents points de vues, par exemple, structurel et comportemental, en utilisant des formalisations différentes.

11. Avec les deux modèles à composer dans le même formalisme.

de correspondance existant, le nouveau modèle de correspondance. Le modèle de correspondance trouvé et les différentes vues à composer étant représentées dans le même formalisme, la composition s'effectue donc dans un environnement homogène et le problème est levé.

Analyse d'impact

L'analyse d'impact n'est pas directement discuté dans les différentes publications concernant les mécanismes de trace présentés dans ce chapitre. Néanmoins, l'analyse d'impact est une des utilisations de la trace de transformations de modèles la plus classique. L'analyse d'impact est une technique utilisée pour identifier les parties d'un programme qui doivent être modifiée si un concept est modifié ou supprimé [124]. En effet, la moindre modification dans la spécification ou dans les besoins d'un projet informatique existant peut impacter différentes parties du programme déjà produit. Certaines peuvent devenir obsolète et d'autres peuvent présenter des problèmes de fonctionnement. Ces parties doivent donc être identifiées pour pouvoir modifier le programme avec le minimum d'efforts. Dans ce contexte, la trace peut être vue comme un ensemble de fils liant les différents éléments de modèles les uns aux autres. Il devient alors plutôt aisé de regarder quels sont les éléments qui vont être impactés par une modification. En d'autres termes, si un élément est modifié, on regarde quels fils sont tirés.

2.1.6 *Récapitulatif*

Afin de comparer les différents mécanismes de trace présentés dans les sections précédentes, nous les avons regroupés dans la table 1. Nous avons différencié les mécanismes de trace proposés par Jouault et Yie pour ATL à cause de leur gestion différente des traces de transformations et de leur structure de méta-modèle. Il faut aussi noter que le langage de méta-modélisation TML est présent dans le tableau, mais ce sont les méta-modèles de trace qu'il est susceptible de décrire qui sont jugés.

Nous avons référencé 9 critères pour comparer et différencier les mécanismes de traces.

Une approche à suivre pour un mécanisme de trace

Comme nous l'avons vu auparavant, les mécanismes de trace suivent deux grandes approches : l'approche orientée purement méta-modèle (P) et l'approche par marquage de trace (T). Si l'on inspecte chacun des mécanismes présentés, on remarque que la plupart d'entre eux suivent une approche par marquage de trace, car cette approche propose plus d'avantages que l'approche orientée purement méta-modèle. En effet, même si l'approche purement méta-modèle, proposée dans [36], introduit une facilité de maintenance et d'analyse par le biais des liens de trace spécifique, la complexité et les restrictions qu'elle entraîne rendent le mécanisme très peu flexible et non réutilisable. À l'inverse, les autres mécanismes, suivants l'approche par marquage de trace, ne possèdent pas cette notion de lien de trace spécifique, mais possède l'avantage d'être beaucoup plus flexibles et facilement réutilisables quel que soit la transformation utilisée.

CRITÈRE	ATL J. [71]	ATL Y. [140]	KERMETA [42]	EML [76]	ETRACE [3]	QVT(O,R) [98]	CORE [98]	TML [36]
Approche	T	T	T	T	T	T	T	P
Liens de trace $1-1$	+	+	+	+	+	+	+	+
Liens de trace $1-n$	+	+	-	+	-	+	+	+
Liens de trace $m-n$	+	+	-	-	-	+	+	+
Capture règle	+	+	+	+	+	+	+	~
Trace des att.	-	-	+	-	+	-	-	-
Indépendance	+	-	+	-	~	-	-	+
Capture liens	E	I	E	I	I	I	E	E
Sauvegarde	EX	EX	EX	EX/IN	EX	EX	EX	EX

Dues aux similarités entre la gestion de la trace pour les langages OPERATIONAL MAPPING et RELATION de QVT, nous les avons concentrés dans une seule colonne (noté QVT(O,R) dans le tableau).
Les valeurs P et T représentent les approches orientée *Pure Metamodel* et *Trace Tagging* respectivement.
Les valeurs E et I sont placés dans le tableau pour représenter les captures de liens EXPLICITE et IMPLICITE.
Les valeurs EX et IN sont placés dans le tableau pour représenter la sauvegarde de la trace dans un modèle EXTERNE ou INTERNE.

TABLE 1: Récapitulatifs des mécanismes de traces présentés dans ce chapitre

Associations d'éléments entre eux

Dans une transformation, il est possible qu'une seule règle consomme plusieurs éléments et produise plusieurs éléments. Afin de représenter au mieux cette consommation et création d'éléments, il faut pouvoir associer, dans la trace, plusieurs éléments consommés à plusieurs éléments produits par une transformation. Dans ce contexte, l'utilisation de liens de trace $m-n$ [12], pouvant lier m éléments consommés à n éléments produits semble la représentation la plus adaptée pour associer des éléments entre eux. Il est à noter que les mécanismes de trace ETRACE [3] et KERMETA [42], considèrent uniquement des liens de trace $1-1$, c'est-à-dire, pouvant associer seulement 1 élément d'un modèle de sortie à 1 élément d'un modèle d'entrée. Avec cette manière de procéder, les informations d'une règle consommant m éléments et produisant n éléments sont représentés par un ensemble de plusieurs liens de trace $1-1$. Il est alors difficile de déterminer avec exactitude quels sont les éléments consommés et produits par une même règle et rend la trace complexe à maintenir et à analyser.

12. Comme pour les mécanismes de trace ATL J. [71], ATL Y. [140], QVT [98] et TML [36].

Conservation des règles

La capture des règles est un des critères universelles de la trace. Tous les mécanismes présentés précédemment proposent de conserver le nom des règles exécutées pendant une transformation. Lors de l'analyse de la trace, cette information permet de savoir quels sont les éléments consommés et produits par une règle en particulier.

Trace des attributs

Dans une transformation, il n'est pas rare de trouver des attributs d'une classe créant, ou participant à la création, de nouveaux concepts. Pour pouvoir conserver l'ensemble des informations relatives à une transformation, il est donc important de pouvoir tracer les attributs pour déterminer quelles sont les créations ou modification qu'ils ont

engendrées dans les modèles de sortie. Ne pas les tracer pourrait introduire des manques dans les transformations et rendre l'analyse des traces plus complexes, voir impossible lorsque les attributs sont beaucoup manipulés. Les seuls mécanismes de trace qui proposent de tracer les attributs sont les mécanismes proposés pour KERMETA [42] et pour *ETraceTool* [3]. Cela suppose une granularité générale des autres mécanismes de trace plutôt grossière et qui peut poser problème lors de certaines opérations réclamant un maximum d'informations dans la trace.

Indépendance de tout langage de transformation

Pouvoir réutiliser au maximum le méta-modèle de trace quel que soit le langage de transformation utilisé permet de pouvoir réutiliser les algorithmes utilisant la trace. En effet, si le méta-modèle ou la façon de capturer la trace est propre à un langage ou un type de transformation, la réutilisation est réduite. Les mécanismes de trace effectivement indépendant de tout langage de transformation sont ceux présentés pour ATL J. [71], KERMETA [42] et les traces proposées par TML [36]. Les autres mécanismes sont complètement dédié à un langage de transformation particulier. Nous pouvons remarquer que la trace ETRACE [3] n'est pas dédiée à un langage en particulier, mais à un type de langage : les langages de transformations impératifs.

Capture des liens de traçabilité

La façon dont les liens de traçabilité sont généré au cours d'une transformation reposent sur l'utilisation de liens de trace implicites ou explicites. Selon les deux types de liens utilisés, les méthodes utilisées pour les récupérés sont différents et entraîne une complexité plus ou moins évidente à gérer selon les langages de transformations utilisés. Le choix de l'exploitation des liens de trace implicite ou explicite dépend du langage utilisé. Si l'on considère un mécanisme de trace comme indépendant de tout langage de transformation, il est alors maladroit de choisir l'exploitation d'un type de lien plutôt qu'un autre sans connaître le langage de transformation à tracer.

Conservation de la trace

La récupération des liens de trace est une étape qui a peu d'importance si on ne détermine pas un moyen pour sauvegarder et accéder à la trace. Tous les travaux présentés proposent de sauvegarder les éléments de trace dans un modèle externe. Ce type de sauvegarde possède l'avantage de ne pas restreindre le mécanisme à la trace de transformations endogènes. En effet, lorsque les transformations sont exogènes, les concepts présents dans les deux méta-modèles ne peuvent pas se retrouver dans un même modèle. Il devient alors impossible de lier les deux concepts sur un même modèle.

2.2 TRAÇABILITÉ DE MODÈLES VERS TEXTE

Les traces produites par les transformations de modèles vers texte possèdent une structure particulière que nous allons présenter ici. Ces traces permettent de tisser un lien direct entre des modèles et une forme textuelle [13]. Les méta-modèles de trace qui sont utilisés pour les

13. Cela peut être du code source ou encore de la documentation.

transformations de modèles à modèles ne fonctionnent plus. Il faut donc utiliser un formalisme dédié pour pouvoir gérer ces liens de traces.

2.2.1 *Méta-modèles de trace pour la génération de texte*

Une transformation de modèles vers texte consiste à associer à un concept d'un méta-modèle d'entrée un bloc de lignes de codes, une seule ligne ou même un mot. C'est cette vision que les méta-modèles de trace de modèles vers texte proposent. À titre d'exemple, la figure 22 présente le méta-modèle de trace de modèles vers texte proposé dans [106].

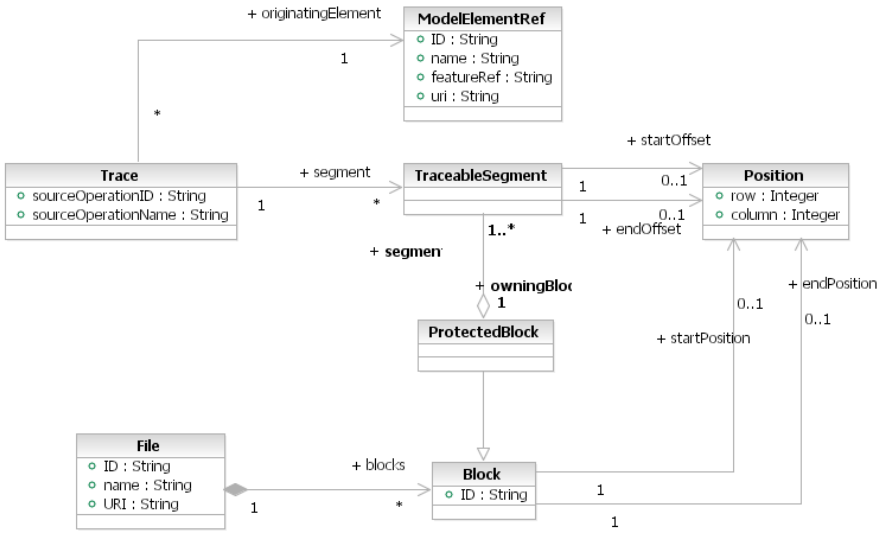


FIGURE 22: Méta-modèle de trace de modèles vers texte [106]

Dans ce méta-modèle, les deux mondes [14] sont clairement distingués. La méta-classe *ModelElementRef* représente une référence à un élément de modèle, alors que la méta-classe *File* représente une référence à un fichier contenant le texte généré. Par ailleurs, chaque *File* est caractérisé par un chemin de stockage *URI*, un nom *name* et un identifiant *ID*. Les fichiers sont considérés comme étant composés de plusieurs *Blocks* de texte commençant et terminant à des *Positions* particulières spécifiées par leur rangée (*row*) et leur colonne (*collumn*). Le coeur [15] de ce méta-modèle de trace relie donc des éléments de modèles et des blocs de texte entre eux. Ainsi, un lien de trace *Trace* relie à un élément de modèle *ModelElementRef* un segment de texte *TraceableSegment* tout en conservant le nom de la règle de transformation exécutée. De cette façon, un *TraceableSegment* peut être le morceau d'un *Block*. Les blocs de lignes peuvent être typés comme bloc protégé, *ProtectedBlock*, signifiant que certains segments tracés sont protégés des changements éventuels et ne peuvent pas être modifiés par l'utilisateur.

Ce méta-modèle originellement présenté dans [106] a été légèrement modifié dans [107] pour en faciliter la manipulation. Ce nouveau méta-modèle est présenté en figure 23.

14. Le monde des modèles et le monde textuel.

15. La structure centrale définissant les liens de trace.

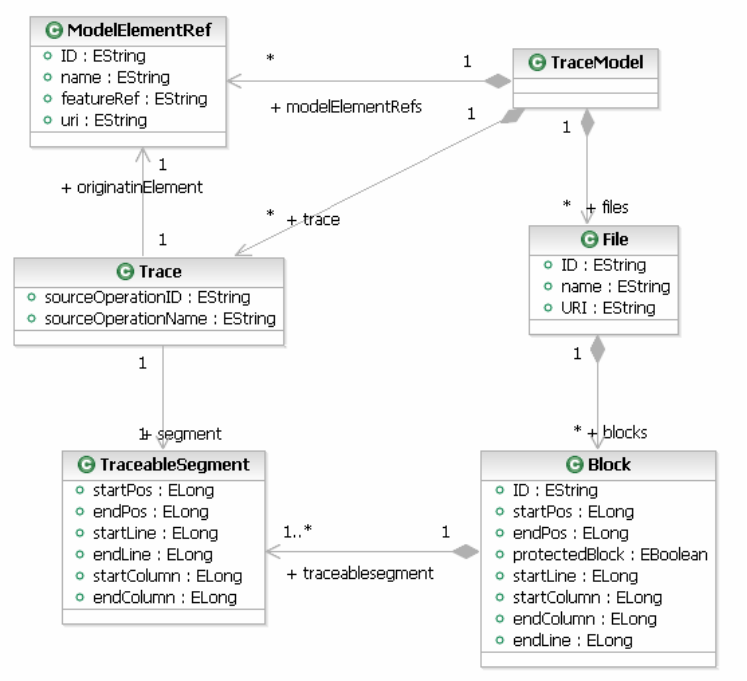


FIGURE 23: Méta-modèle de trace de modèles vers texte [107]

On retrouve le même cœur que dans le méta-modèle de la figure 22 avec des liens de *Trace* liant des éléments de modèles *ModelElementRef* avec des segments de texte *TraceableSegment*. Il en est de même pour la représentation des fichiers. Cependant, on observe la disparition de la méta-classe *ProtectedBlock* au profit d'un attribut dans la méta-classe *Block* et la disparition de la méta-classe *Position* pour des attributs dans les méta-classes *Block* et *TraceableSegment*.

Finalement, il est évident que compte tenu de leur forte liaison à la structure générale des fichiers texte, la gestion des traces de transformations de modèles vers texte est difficile.

2.2.2 *Générer une trace de modèles vers texte*

La plupart des techniques appliquées aux transformations de modèles à modèles ne sont pas utilisables pour générer une trace de modèles vers texte.

- **Utilisation de HOT** : l'utilisation de HOT dans ce contexte est tentante, mais inapplicable. Dans le contexte de transformations de modèles à modèles, les HOTs permettaient d'insérer automatiquement des lignes pour générer le modèle de trace à l'exécution de la transformation. Cependant, dans le contexte de transformations de modèles vers texte, les langages de transformation ne permettent pas de générer des modèles, ce qui empêche l'utilisation de cette technique.
- **Utilisation d'aspects** : actuellement, aucun tisseur d'aspect n'existe pour les langages de transformation de modèles vers texte, il est donc impossible de porter la solution pour les transformations de modèles à modèles proposée dans [3] aux transformations de modèles vers texte.

– **Utilisation des données du moteur de transformation** : cette technique propose de récupérer directement dans le moteur de transformation les informations nécessaires. Cette technique permet à la fois de récupérer les informations de trace, mais aussi de créer un modèle de trace externe.

La méthode employée pour la construction de méta-modèles de trace consiste à récupérer directement dans le moteur de transformation les informations dont la trace à besoin. Cependant, une telle méthode n'est pas sans risque pour la maintenance et la réutilisation de la trace par un algorithme *ad-hoc*. En effet, la trace dépend entièrement de la manière dont la transformation a été écrite. Si celle-ci n'a pas été correctement ou assez « découpée » en règles [16], on peut se retrouver, dans des cas extrêmes, avec une trace ne comportant qu'un seul lien de traçabilité entre tous les éléments des modèles et tous (voir un seul !) les blocs d'un fichier. Il devient alors impossible de maintenir ou d'utiliser la trace produite pour une quelconque opération. Ceci impose, lorsqu'une transformation de modèles vers texte est écrite, d'observer certaines bonnes pratiques pour rendre la transformation et la trace plus simple à maintenir et à réutiliser [132].

16. Il est souvent possible de générer plusieurs lignes de code relatives à plusieurs concepts des modèles d'entrée grâce à une seule règle.

2.2.3 *Utilisation des traces de modèles vers texte*

Dans [107], plusieurs exemples d'utilisation de la trace de modèles vers texte sont donnés. On y retrouve entre autre les analyses suivantes :

– l'analyse de couverture,
– l'analyse d'impact,
– l'analyse d'orphelins,
– la documentation de traces,
– l'évolution de modèles de traçabilités.

Une autre utilisation de la trace de modèles vers texte souvent mise en avant est la synchronisation entre les modèles et le texte généré par la transformation. La synchronisation entre modèles et texte propose de modifier automatiquement un modèle en fonction des modifications effectuées sur le code généré ou de modifier un texte généré à partir des modifications effectuées sur le modèle. La trace est donc utilisée pour identifier, soit les éléments relatifs aux blocs de code modifiés, soit les blocs de code relatifs à l'élément modifié. La synchronisation automatique entre modèles et texte est complexe à mettre en œuvre, mais est possible lorsque le texte correspond à un code source et que la synchronisation est implémentée pour un langage en particulier. À titre d'exemple, nous pouvons citer l'outil commercial Traceability [97] proposé par Obeo qui assure, grâce à la trace de modèles vers texte, une synchronisation entre modèles et code JAVA.

2.3 TRAÇABILITÉ DANS LES CHAÎNES DE TRANSFORMATIONS

Parmi les méta-modèles de traces que nous avons présentés précédemment, seul le méta-modèle de trace proposé pour le langage KERMETA [42] est capable de travailler avec des chaînes de transformations.

Celui-ci est représenté en figure 24 [17]. Une méta-classe *Trace* a été ajoutée pour permettre le stockage de plusieurs *Steps*. Un *Step*, quant à lui, permet de conserver les liens créés pour une transformation. Ainsi, une *Trace* permet de stocker plusieurs transformations.

17. *c.f* aussi, figure 8b en page 29 et figure 13 en page 33.

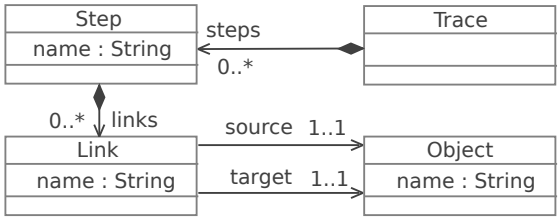


FIGURE 24: Méta-modèle de trace KERMETA pour des chaînes de transformations [42]

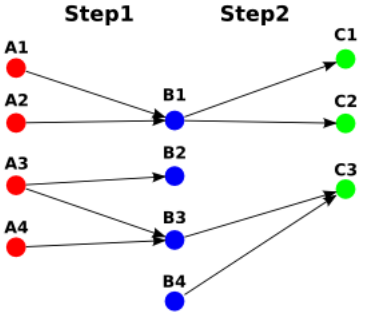


FIGURE 25: Exemple de trace KERMETA pour une chaîne de transformations [42]

Afin d'illustrer les nouveaux concepts introduits dans la trace KERMETA, la figure 25 présente un exemple de trace pour une chaîne composée de deux transformations. Les deux transformations sont représentées par *Step1* et *Step2*. La première transformation prend les éléments A_i en entrée et produit les éléments dans le modèle de sortie, les éléments B_j. Ces éléments, ainsi qu'un élément B4 déjà présent dans le modèle sont directement consommés par la seconde transformation pour produire les éléments C_k. La trace ainsi produite garde tous les liens tissés durant les transformations et permet de naviguer du premier modèle au dernier. Ainsi, il est possible de savoir, par exemple, que l'élément C3 du dernier modèle produit a été créé à partir des éléments A3 et A4 dans le premier modèle. Le méta-modèle de trace proposé pour KERMETA est donc capable de travailler avec des chaînes de transformations de modèles à modèles uniquement. Pour des chaînes de transformations orientées compilation[18], la dernière étape de génération de code ne pourrait pas être sauvegardé, ni tracé avec ce formalisme.

La trace KERMETA permet donc de gérer la trace dans des chaînes de transformation KERMETA. Cependant, elle ne permet pas de gérer la trace pour des chaînes utilisant plusieurs langages de transformations de modèles à modèles différents.

18. *c.f* Chaînes de compilations IDM au chapitre 1.

LA TRAÇABILITÉ DANS GMM La gestion de la trace dans GMM[19] effectue une distinction entre une traçabilité dans le *small* et dans le *large* [10]. La traçabilité dans le *small* fait référence à la traçabilité pour une transformation alors que la traçabilité dans le *large* fait référence à la chaîne de transformations complète. Pour référencer les différents modèles de traces, les auteurs utilisent des méga-modèles[20]. En utilisant les méga-modèles, il est possible de naviguer entre les différentes traces et surtout entre un modèle et les traces qui lui sont attachées.

19. GLOBAL MODEL MANAGEMENT [2].

20. Un modèle dont ses éléments représentent des modèles ou des méta-modèles [16, 43].

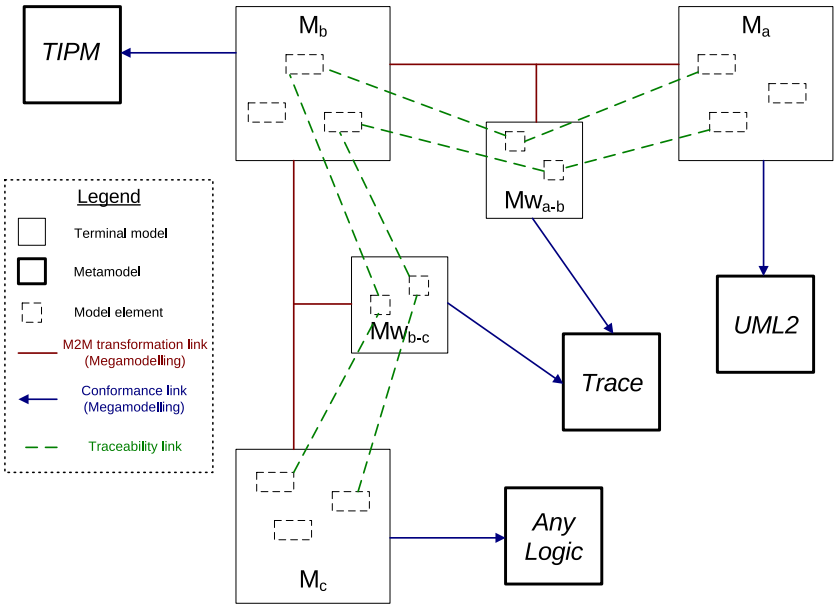


FIGURE 26: Exemple de gestion d'un chaîne de ormations en utilisant les méga-modèles [21]

La figure 26 présente les différents liens introduits par méga-modelling dans une simple chaîne de transformations. La chaîne esquissée est composée de deux transformations : une transformant le modèle UML M_a en modèle TIPM M_b et une autre transformant le modèle M_b en modèle ANYLOGIC M_c [21]. Nous ne nous attardons pas sur les méta-modèles auxquels chaque modèle est conforme. Durant les transformations, des liens de traces ont été tissés (liens en pointillés) et sont stockés dans deux modèles de traces : MW_{a-b} et MW_{b-c}. Les liens de transformation introduits par méga-modelling sont les liens représentés entre chaque modèle et sont notés *M2M transformation link* dans la légende. Il s'agit d'une relation ternaire reliant les modèles d'entrée et de sortie d'une transformation et la trace produite par la transformation. De cette façon, il est possible de naviguer entre les différents modèles et traces en utilisant les liens introduits par méga-modelling.

LA TRAÇABILITÉ DANS UNITI UNITI[21] est une infrastructure pour la spécification et l'implémentation de chaînes de transformations [125]. Cette infrastructure propose d'implémenter les transformations complexes par une succession de plusieurs transformations plus simples à écrire. Le but principal de cette approche est de faciliter les compositions transparentes et d'augmenter la réutilisation des transformations écrites dans différents langages de transformations. Pour atteindre cet objectif, cette infrastructure génère et gère des liens de traces entre chaque transformation et composition [130].

21. **Uni**FIED TRANSFORMATION INFRASTRUCTURE

La figure 27 présente la façon dont les traces sont gérées par UNITI. Cette figure montre le méta-modèle UNITI permettant la spécification et l'exécution des chaînes de transformations décrites. Les parties relatives à la gestion de la trace sont celles encadrés. Le méta-modèle de base de UNITI a été étendu avec les méta-classes *TraceFormalParameter* (situées dans les encadrés haut et bas de la figure). Elles représentent les modèles de traces qui peuvent être produits par une transformation :

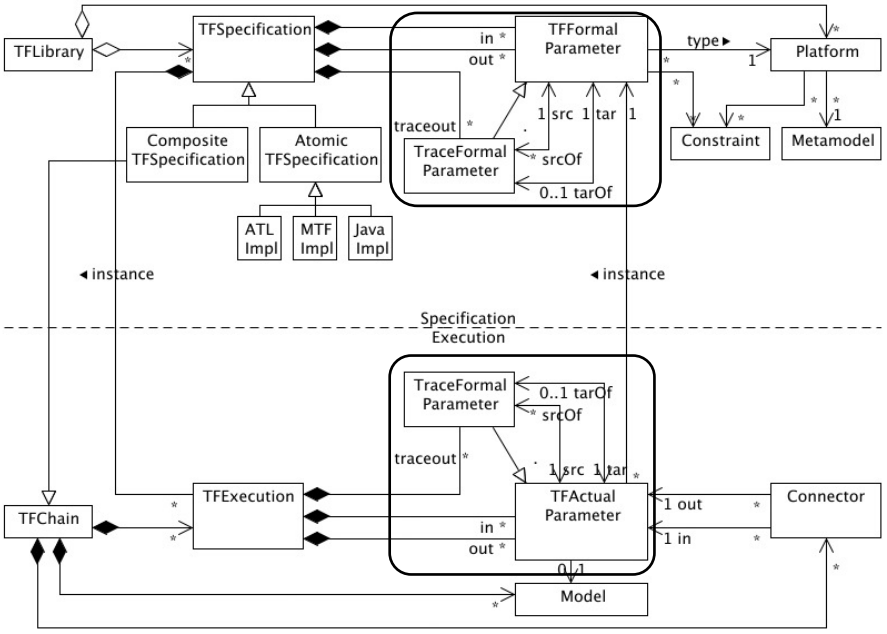


FIGURE 27: Intégration de la trace dans le méta-modèle de UNITI [123]

TFSpecification (au niveau spécification) et *TFExecution* (au niveau exécution). Ces modèles de traces sont reliés à des *TFformalParameter* et des *TFActualParameter* représentant les paramètres d'une transformation [22]. De cette façon, pendant la construction ou l'exécution d'une chaîne de transformations, les liens entre les traces et les modèles consommés et produits sont conservés.

22. Les modèles d'entrée et de sortie d'une transformation.

Gérer au mieux les traces produites dans une chaîne de transformations n'est pas une chose facile. Lorsque l'on travaille avec des chaînes de compilation IDM, les transformations de modèles à modèles sont chaînées les unes aux autres. Ainsi, il faut aussi bénéficier d'un mécanisme pour pouvoir conserver la séquence de traces produites à l'instar de ceux présentés dans cette section 2.3. Le mécanisme proposé par KERMETA est séduisant, mais mélange les préoccupations en embarquant la conservation des séquences de traces de modèles et les traces de modèles dans le même méta-modèle, ce qui va à l'encontre du principe de séparation des préoccupations préconisé par l'IDM. Afin de séparer au mieux les préoccupations, deux types de traçabilités sont différenciés : la traçabilité dans le *small* et dans le *large*. Les approches présentées dans [10, 125] tiennent compte de cette sépration. L'approche proposée dans [10] utilise des méga-modèles pour pouvoir lier les différents modèles entre eux. Cependant, l'utilisation de méga-modèles n'est pas dédié spécifiquement pour un mécanisme de trace. Il est évident que la séquence entre les traces de modèles sont bien conservées, mais proposer des algorithmes manipulant uniquement les traces devient plus complexe à écrire. En effet, il faudrait pouvoir déterminer précisément les types des modèles pointés dans le méga-modèle. Un mécanisme plus dédié à la trace serait donc préférable. Dans [123], le mécanisme proposé est entièrement dédié à la trace, mais il est dédié à la plateforme UNITI. Ainsi, la séquence des traces peut être conservée, mais

uniquement lorsque l'on traite avec le méta-modèle complet UniTi. Les concepts utiles à la trace n'ont plus de sens et ne sont plus utilisables lorsqu'ils sont extraits du méta-modèle UniTi.

2.4 DES EXIGENCES POUR UN MÉCANISME DE TRACE

L'étude de l'état de l'art montre qu'il existe une forte dichotomie entre les approches gérant la trace dans les transformations de modèles à modèles et celles la gérant dans les transformations modèles vers texte. En effet, les concepts manipulés par les deux types de transformations sont vraiment différents. En revanche, il est possible dans la même approche de gérer la trace pour une simple transformations de modèles à modèles et pour une chaîne de transformations. De plus, l'analyse des différentes approches existantes à permis de mettre en évidence 6 exigence relatives à la trace de transformations de modèles à modèles et 3 exigences relatives à la trace dans une chaîne :

1. approche orientée marquage de trace,
2. capture des règles de transformation,
3. liens de trace $m - n$,
4. trace des attributs,
5. indépendance de tout langage de transformation,
6. sauvegarde de la trace dans un modèle externe,
7. séparation des préoccupations,
8. approche dédiée uniquement à la trace,
9. indépendance de toute plate-forme.

Parmi les mécanismes de trace présentés précédemment, aucun ne prend en compte la totalité des exigences que doit satisfaire un mécanisme de trace. Pour pouvoir prendre en compte l'intégralité des exigences, il faut donc définir un nouveau mécanisme.

2.5 CONCLUSIONS

Dans ce chapitre nous avons vu comment les transformations de modèles à modèles et de modèles vers texte sont tracés. Nous avons présenté quelques méta-modèles de trace utilisés par les langages de transformations les plus utilisés dans le monde de l'IDM, à savoir, ATL, Kermeta, EML et QVT. Nous avons ensuite discuté des différentes techniques utilisées pour capturer les liens de traçabilité lors de l'exécution d'une transformation avant de montrer quelques utilisations classiques de tel types de traces. Finalement, nous avons montré comment les traces sont gérées lorsque des transformations sont chaînées les unes aux autres.

Les avancées effectuées dans le monde de la traçabilité depuis les premiers efforts mis en œuvre pour lier deux artefacts entre eux sont importantes. Cependant, alors que les traces sont extrêmement utilisées dans l'ingénierie des besoins, les traces de transformations de modèles sont encore peu utilisées par des applications ou algorithmes tierces. Les différents mécanismes existant dans la littérature possèdent des avantages quant à leur façon de stocker les liens de traces tissés ou à leur façon de les capturer, mais chacun d'eux présente un manque ou une particularité qui les lie trop à un langage donné et limite leur

application à un autre. De ces remarques, nous avons tiré une liste d'exigence qu'un mécanisme de trace se doit de satisfaire pour gérer la trace dans les transformations de modèles à modèles et le chaînes de transformations et nous avons vu qu'aucun des mécanismes actuels ne permet de satisfaire l'ensemble de ces exigences.

Dans le prochain chapitre, nous présentons notre propre mécanisme de trace qui prend compte les remarques tirées de ce chapitre et nous montrons que celui-ci satisfait les exigences énoncées dans ce chapitre.

Deuxième partie

MÉCANISME DE TRAÇABILITÉ À GRANULARITÉ FINE

3

MÉCANISME DE TRACES ADAPTÉ AUX TRANSFORMATIONS ET AUX CHAÎNES

Dans le chapitre précédent, nous avons donné un certains nombre d'exigences qu'un mécanisme de trace se doit de satisfaire. Parmi les mécanismes de trace existants, nous avons vu qu'aucun d'eux n'était en mesure de satisfaire l'ensemble de ces exigences et qu'il était, par conséquent, nécessaire de proposer un nouveau formalisme.

Afin de répondre aussi bien aux exigences relatives à la trace dans une unique transformation de modèles vers modèles que celles traitant de la trace dans les chaînes, nous proposons un mécanisme de traçabilité qui repose sur l'utilisation de deux méta-modèles spécifiques.

Dans un premier temps, nous présentons en détails de notre approche dans les sections 3.1 et 3.2 avant de montrer en section 3.3 un moyen efficace de gagner en performances lors d'un passage à l'échelle. Par la suite, en section 3.4, nous présentons comment notre mécanisme a été adapté à QVT et comment il permet d'outrepasser des limitations de la trace « native » QVT. Finalement, nous revenons sur les transformations localisées et les transformations *sur place* en section 3.5 avant de conclure en section 3.6.

3.1 TRACE LOCALE

Pour pouvoir tracer les transformations de modèles tout en respectant les exigences que nous avons énoncées précédemment, nous avons proposé dans [56] le méta-modèle de trace locale. Le méta-modèle de trace

locale est présenté entièrement en figure 28. Il est composé de 9 méta-classes et d'une référence au méta-modèle ECORE [1], implémentation du EMOF.

1. Par la méta-classe *EObject*.

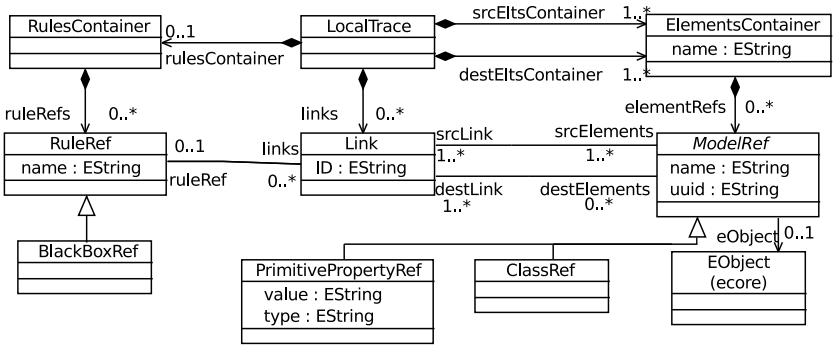


FIGURE 28: Méta-modèle de trace locale

Pour plus de clarté, nous avons décomposé la description du méta-modèle en trois parties.

3.1.1 *Le lien entre éléments : le cœur du méta-modèle de trace locale*

Dans un premier temps, nous présentons le « cœur » du méta-modèle, c'est-à-dire les concepts qui permettent la création de liens de trace et leurs associations à des éléments des modèles.

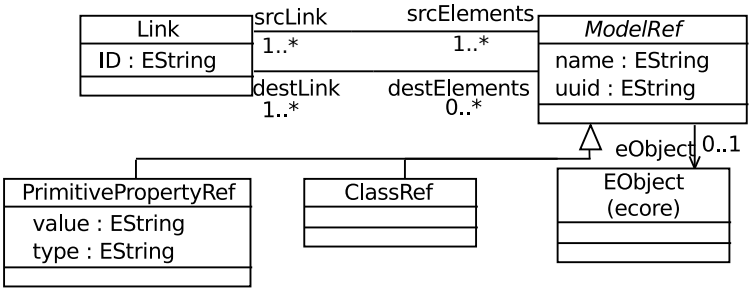


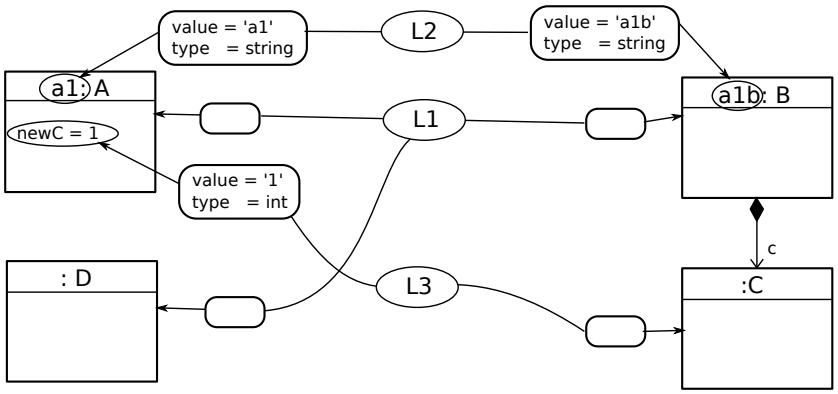
FIGURE 29: Extrait du méta-modèle de trace locale : le cœur de la trace

Le méta-modèle de Jouault a été pris comme base pour la construction de notre méta-modèle. Ainsi, on retrouve le même cœur structurel que pour les méta-modèles de trace présentés précédemment avec les méta-classes *Link* et *ModelRef*. La méta-classe *ModelRef* est ici utilisée comme représentation d'un élément [2]. Le concept représentant le lien *Link* embarque un attribut *ID* qui permet de différencier les différents liens tissés lors de la construction de la trace. Le lien peut associer un ou plusieurs *ModelRef* d'entrée à plusieurs *ModelRef* de sortie. Le concept *ModelRef* possède une référence à un *EObject* qui pointe vers un élément manipulé par la transformation au cours de son exécution. De plus, la méta-classe *ModelRef* contient deux attributs *name* et *uuid* [3]. Le premier correspond au nom de l'élément tracé, si celui-ci en porte un et le second correspond à un numéro d'identification unique, automatiquement généré par le *framework* utilisé pour l'implémentation du méta-modèle, porté par l'élément de modèle. Afin de pouvoir bénéficier d'une granularité fine et ainsi pouvoir tracer les attributs, la méta-classe *ModelRef* est abstraite et peut être instanciée en tant que

2. Au même titre que la méta-classe *TransientElement* proposée par Yie et al. *c.f* figure 12 page 32.

3. *Universal Unique IDentifier*

PrimitivePropertyRef ou *ClassRef*. La figure 30 montre un exemple de trace locale tenant compte uniquement de ces 4 méta-classes.



Afin de ne pas surcharger l'illustration, nous avons volontairement enlevé les attributs *name* et *uuid* des *ModelRef*, ce qui explique les éléments « vide ».

FIGURE 30: Exemple de trace locale

Sur l'exemple, les classes *a1 :A* et *:D* appartiennent au modèle d'entrée alors que les classes *a1b :B* et *:C* appartiennent au modèle de sortie.On trouve 7 éléments de type *ModelRef* (carrés arrondis) dont 3 *PrimitivePropertyRef*[4] et 4 *ClassRef*. Les liens de traces sont eux représentés par les éléments notés *L1*, *L2* et *L3*. En regardant les informations contenues dans la trace locale, il est possible d'affirmer à partir du lien *L1* que les éléments *a1 :A* et *:D* du modèle d'entrée ont créé l'élément *a1b :B* du modèle de sortie. De même, le lien *L2* permet de dire que l'attribut *a1* a servi à créer l'attribut *a1b*. Finalement, le dernier lien nous montre que l'attribut *newC* a permis de produire la classe *:C*.

L'utilisation d'une référence vers l'élément *EObject* permet de tracer n'importe quel élément de modèle sans en préciser les types ce qui place nettement notre méta-modèle dans une approche de trace orientée marquage de trace. De plus, comme nous l'avons vu en figure 29 et montré sur l'exemple de la figure 30, les attributs sont tracés et les liens de trace peuvent être $m - n$.

4. Les *ModelRef* de type *PrimitivePropertyRef* sont les carrés arrondis contenant des valeurs pour les attributs *value* et *type*.

3.1.2 *Capturer les règles de transformations*

Dans les méta-modèles présentés au chapitre 2, c'est le concept de lien qui capture, via un attribut, le nom de la règle le créant. Dans la trace locale, nous proposons de séparer les préoccupations en désolidarisant les règles, des liens qu'elles ont créés.

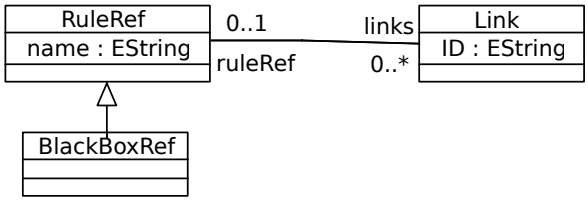


FIGURE 31: Extrait du méta-modèle de trace locale : capturer les règles

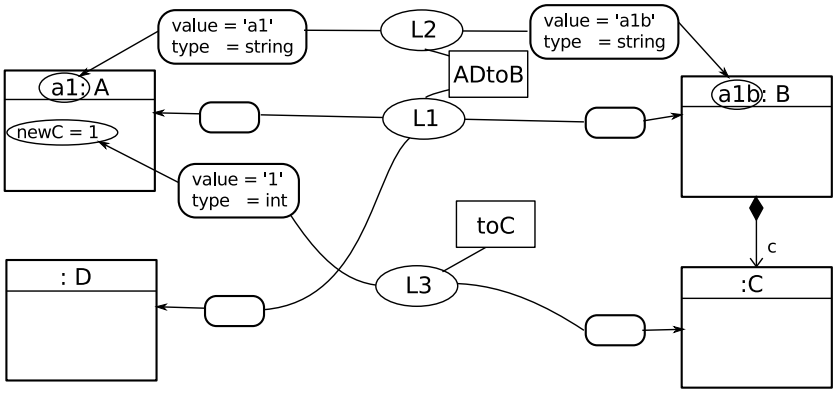
La figure 31 montre un autre extrait du méta-modèle de trace locale contenant les concepts relatifs à la capture des règles de transformations. Sur cet extrait, on retrouve le concept *Link* précédemment introduit[5]

5. *c.f*, figure 29.

ainsi que les nouveaux concepts : *RuleRef* et *BlackBoxRef*. Le concept *RuleRef* est utilisé pour capturer les règles de transformation. Son attribut *name* permet de conserver le nom de la règle qui crée un élément. La référence liant les deux concepts indique qu'une règle de transformation peut être associée à plusieurs *Links*. Cette même référence indique aussi que le concept *RuleRef* n'est pas obligatoire et qu'un *Link* peut éventuellement n'être relié à aucune règle [6]. Finalement, le concept *RuleRef* est raffiné en *BlackBoxRef* pour pouvoir représenter d'éventuelles règles de transformation qui seraient boite noire [7]. La figure 32 représente la figure 30 avec les nouveaux concepts introduits dans l'extrait de méta-modèle présenté à la figure 31.

6. Nous verrons par la suite un contexte dans lequel cette référence peut ne pas exister.

7. Dont les détails d'implémentations ne sont pas connues.



Afin de ne pas surcharger l'illustration, nous avons volontairement enlevé les attributs *name* et *uuid* des *ModelRef*, ce qui explique les éléments « vide ».

FIGURE 32: Exemple de trace locale

Sur cet exemple enrichi des nouveaux concepts, nous retrouvons deux *RuleRef* : *ADtoB* et *toC*. La lecture de la trace représentée permet de voir que la consomation de l'élément *a1 :A* et *:D* par la règle *ADtoB* a mené à la création de l'élément *B*. De même, l'attribut *a1* à été consommé par la règle *ADtoB* qui a produit l'attribut *a1b*. D'autre part, il est possible de dire que l'attribut *newC* a été utilisé par la règle *toC* pour créer *:C*.

Grâce au concept *RuleRef*, le méta-modèle de trace locale permet de capturer les rèles de transformation.

3.1.3 *Organiser la trace*

Il est judicieux de pouvoir organiser la trace produite pour pouvoir facilement atteindre certains concepts. En effet, dans les extraits de méta-modèle de trace locale présenté actuellement, il est difficile de voir, par exemple, quels sont les *ModelRef* pointant vers des éléments des modèles d'entrée ou de sortie de la transformation. Afin de mieux organiser et aussi faciliter la lecture de la trace produite, les concepts *LocalTrace*, *RuleContainer* et *ElementsContainer* ont été ajoutés.

Sur l'extrait de méta-modèle de la figure 33, on retrouve également les concepts *Link*, *ModelRef* et *RuleRef* présentés précédemment [8]. Le concept, *LocalTrace* est la racine de toute trace locale. Elle contient directement les différents liens de trace tissés lors de l'exécution de la transformation. En revanche, elle ne contient pas directement les *RuleRef* et les *ModelRef*, mais deux conteneurs d'éléments particuliers *RuleContainer* et *ElementsContainer*. Le premier container, *RuleContainer* est utilisé pour regrouper l'ensemble des *RuleRef* créés alors que le

8. *c.f*, figure 29 et figure 31.

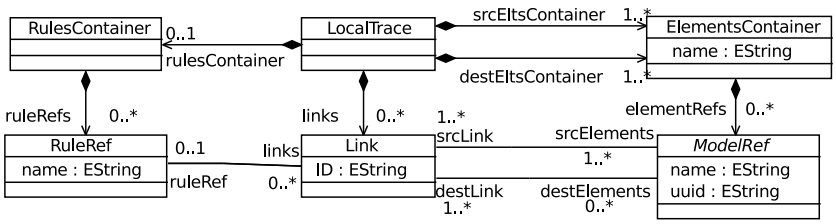
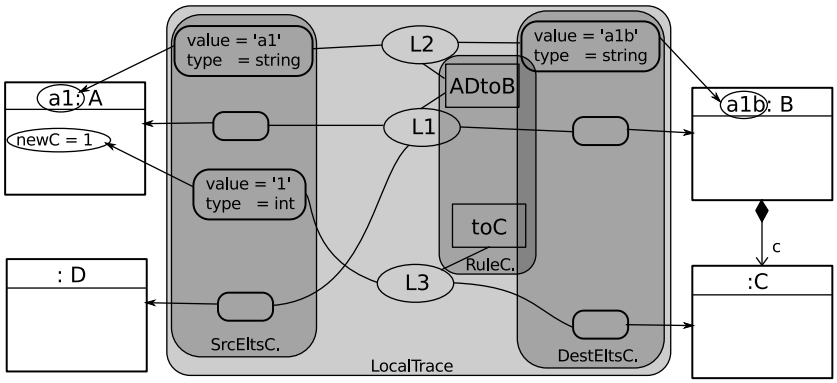


FIGURE 33: Extrait du méta-modèle de trace locale : l'organisation de la trace

concept *ElementsContainer* est utilisé pour regrouper les *ModelRef* selon leur appartenance à un modèle d'entrée ou de sortie de la transformation. C'est grâce aux références *srcEltsContainer* et *destEltsContainer*, que les *ElementsContainers* représentant les éléments des modèles sources manipulés et ceux des modèles destination de la transformation sont différenciés. Ainsi, en considérant un *ElementsContainer* par modèle d'entrée et de sortie, il est précisé qu'une trace locale peut lier plusieurs modèles d'entrée à plusieurs modèles de sortie d'une transformation. La figure 34 illustre, sur la base de la figure 32, les concepts ajoutés par l'extrait de méta-modèle présenté en digure 33.



Afin de ne pas surcharger l'illustration, nous avons volontairement enlevé les attributs *name* et *uuid* des *ModelRef*, ce qui explique les éléments « vide ».
Les noms *RuleC.*, *SrcEltsC.* et *DestEltsC.* sont des abréviations pour *RuleContainer*, *SrcEltsContainer* et *DestEltsContainer*.

FIGURE 34: Exemple de trace locale

Les divers éléments de la trace locale sont regroupés et organisés dans différents conteneurs. Ainsi, on retrouve la racine du modèle de trace *LocalTrace* regroupant les liens de trace *L1*, *L2* et *L3*, les conteneurs d'éléments *SrcEltsC.* et *DestEltsC.* ainsi que le conteneur de règle *RuleC.*. Les conteneurs permettent de mieux représenter la trace locale et d'accéder rapidement à des ensembles d'éléments facilement sans avoir à effectuer de recherches complexes. La figure 35 présente une capture d'écran de la trace locale illustrée en figure 34 sous forme arborescente. On y distingue les *ElementsContainers* et les *ModelRefs* qu'ils contiennent, le *RulesContainer* ainsi que les différents *Links* reliant les éléments manipulés par la transformation entre eux. Si l'on regarde les propriétés du *Link* mis en surbrillance, on retrouve les informations présentes dans la figure 34, à savoir que les éléments *a1 : A* et *: D* ont mené à la création de l'élément *a1b : B* par l'exécution de la règle *ADtoB*.

Pour la définition de ce méta-modèle de trace locale, pas une seule fois nous ne nous sommes intéressés aux capacités ou aux spécificités

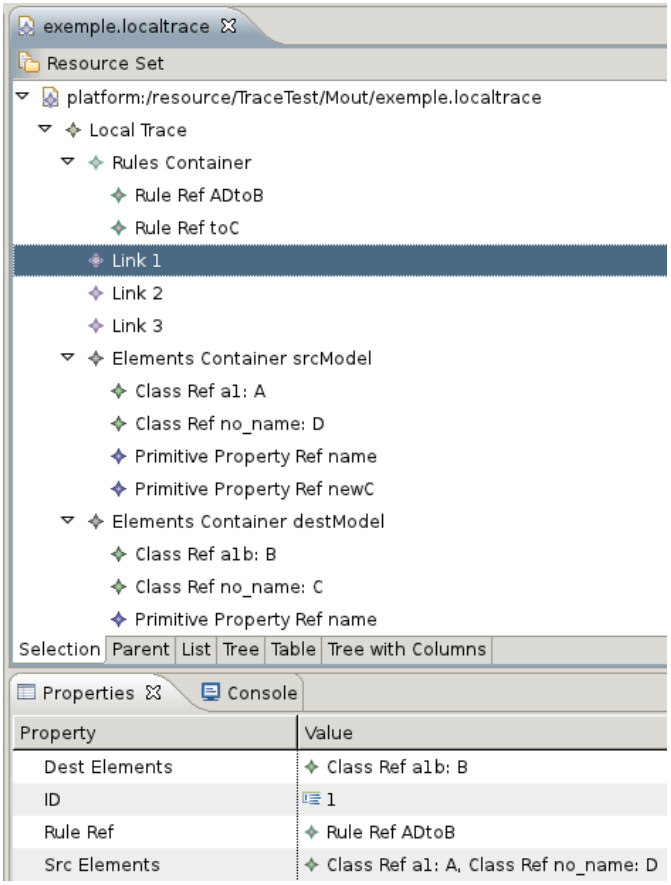


FIGURE 35: Exemple de trace locale sous forme arborescente

proposées par un langage de transformation. Le méta-modèle de trace locale a été défini indépendamment de tout langage de transformation

Le méta-modèle de trace locale est capable de tisser des liens de trace $m - n$, de tracer les attributs, de capturer les règles de transformation et il est indépendant de tout langage de transformation.

3.2 TRACE GLOBALE

Le méta-modèle de trace locale précédemment présenté permet de produire un ensemble de liens de traçabilité pour des transformations de modèles à modèles. Cependant, lorsqu'une chaîne de transformations est utilisée, le méta-modèle de trace locale n'est pas capable de garder la séquence des traces produites. Pour effectuer une telle tâche, nous avons aussi proposé dans [56] le méta-modèle de trace globale.

Le méta-modèle de trace globale est présenté en figure 36. La racine du modèle de trace est représentée par le concept *GlobalTrace*. La trace globale peut contenir des *TraceModels* et des *LocalModels*. Le concept *TraceModel* représente un modèle de trace locale auquel il est lié par la référence *localTrace*, référençant la racine du méta-modèle de trace locale. Le second concept *LocalModel* représente les modèles d'entrée et de sortie de la transformation tracée. Le méta-modèle indique aussi que chaque *TraceModel* est lié à plusieurs *LocalModel*. Les modèles de sortie sont accessibles grâce à la référence *destModels* et les modèles d'entrée

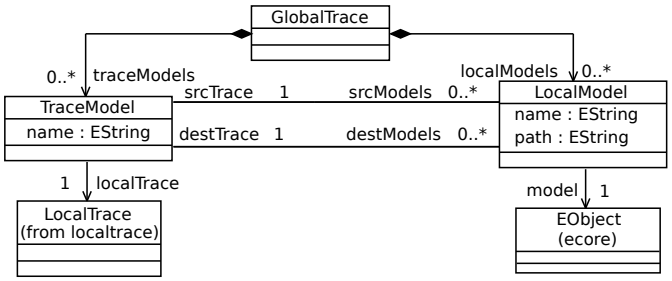


FIGURE 36: Méta-modèle de trace globale

grâce à la référence *srcModels*. Ainsi, deux transformations T_1 et T_2 se suivent si le modèle source de T_2 est dans l'ensemble *srcModels* de la trace associée à T_2 et dans l'ensemble *destModels* de la trace associé à T_1. La trace globale permet donc de naviguer à travers les modèles et les traces de modèles créées tout le long d'une chaîne de transformations.

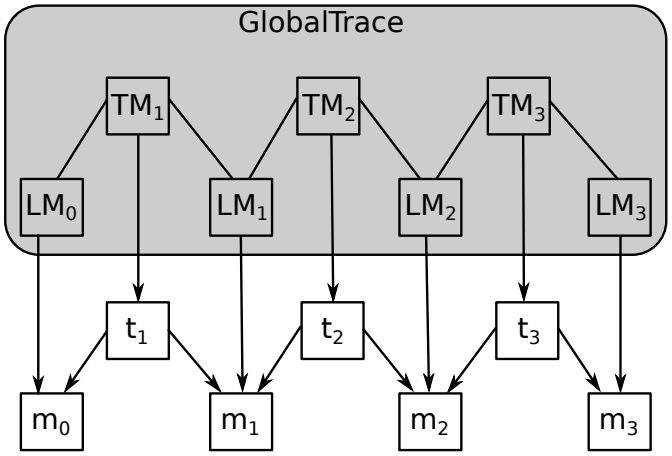


FIGURE 37: Exemple de trace globale pour une chaîne de 3 transformations

La figure 37 montre un exemple de trace globale sur une chaîne composée de 3 transformations. Le modèle m_0 est transformé en modèle m_1 et produit un modèle de trace t_1. Ce modèle m_1 est ensuite transformé en m_2, puis le modèle m_2 est transformé en m_3 tout en produisant les modèles de trace t_2 et t_3. La trace globale *GlobalTrace* contient 4 *LocalModel* notés LM_0, LM_1, LM_2 et LM_3. Ces concepts pointent sur les modèles en entrée et en sortie des traces. Ils sont aussi reliés à des *TraceModel* (notés TM_1, TM_2 et TM_3). Grâce à ces références, il est possible de naviguer de *LocalModel* en *TraceModel* et ainsi passer d'une trace locale à une autre et/ou d'un modèle transformé à un autre.

Ce méta-modèle de trace globale, uniquement dédié à la trace locale, permet de séparer les préoccupations liées à la gestion des traces locales de la trace des transformations. Il permet de gérer les différentes liaisons entre les traces locales et les modèles manipulés dans une chaîne de transformations. De plus, le méta-modèle de trace locale a été défini indépendemment de toute plate-forme et, est par conséquent, réutilisable.

3.3 TRACE RÉDUITE

L'utilisation conjointe de la trace locale et de la trace globale permet de garder les liens tissés dans une transformation de modèle et de naviguer à travers les traces locales produites pour une chaîne de transformations. Comme nous l'avons vu avec la figure 32, lorsqu'une trace locale est produite, elle contient un grand nombre d'éléments. Sur ce simple exemple, 12 concepts de trace ont été créés et gardés dans la trace locale. Il est évident que lorsque l'on passe à l'échelle, le nombre d'éléments augmente et, par conséquent, la taille de la trace locale aussi. De manière générale, plus les modèles d'entrée d'une transformation possèdent d'éléments, plus il y a de concepts manipulés et créés par la transformation et, en conséquence, plus le modèle de trace locale possède d'éléments.

Ce problème de taille reste relativement gérable lorsque l'on travaille avec une seule trace locale, mais devient complexe à gérer lorsque plusieurs traces locales coexistent, comme lors de l'utilisation d'une chaîne de transformations. En conséquence, le passage à l'échelle couplé à l'utilisation de plusieurs traces dans les chaînes de transformations peut engendrer des pertes de performances, notamment des algorithmes qui les manipulent régulièrement. Pour réduire les temps d'exécution et accroître les performances de ces algorithmes, il est nécessaire de réduire au maximum les traces locales sans perdre d'informations.

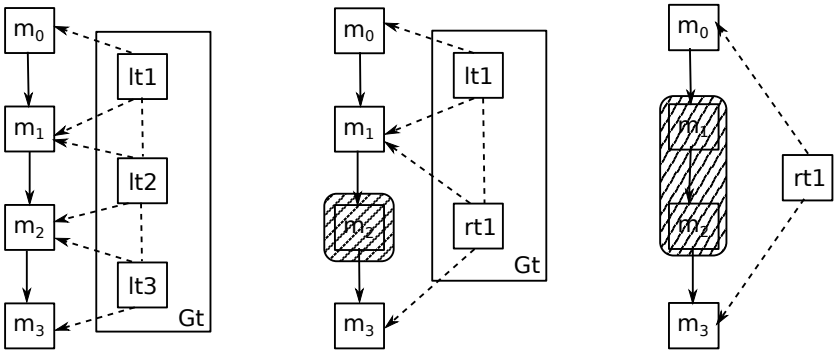
Dans une chaîne de transformations, toutes les traces sont rarement scrutées par l'utilisateur. Par exemple, pour une chaîne de compilation IDM, l'utilisateur de la chaîne peut traiter avec les modèles d'entrée et le code produit uniquement. L'ensemble des modèles et des traces intermédiaires ne lui sont pas directement utiles pour l'analyse. Les traces intermédiaires permettent de reconstituer les liens entre les modèles d'entrée de la chaîne et les dernier modèles. Or, seuls ces liens sont utiles. Donc en construisant ces liens à partir des traces locales, cela pourrait réduire la taille des traces manipulées et améliorerait les performances des algorithmes utilisant les traces locales. D'un point de vue de la trace, cela revient à considérer la chaîne de transformations comme une unique transformation.

L'opération que nous venons de décrire, effectuée sur l'ensemble d'une chaîne de transformations, pourrait aussi être opérée localement sur un morceau de la chaîne. Cette opération de réduction masque une partie de la chaîne de transformations en considérant la séquence des transformations masquées comme une unique transformation.

Ce principe de réduction, que nous appelons « trace réduite », est présenté en figure 39 sur une chaîne faite de 3 transformations. La chaîne transforme un modèle initial m_0 en un modèle m_3 tout en produisant 3 traces locales lt_1 à lt_3 tel que montré sur la figure 38a. Une première réduction rt_1 de la trace est montrée en figure 38b. Cette réduction masque la transformation produisant le modèle m_2 dans la chaîne de transformations. Ainsi, il est possible de naviguer directement du modèle m_1 au modèle m_3 par l'intermédiaire de rt_1. Cette trace réduite est dite trace réduite $1,3$ puisqu'elle s'effectue entre les étapes 1 et 3 de la chaîne de transformations, produisant les modèles m_1 et m_3. De la même manière, la figure 38c montre une trace réduite $0,3$[9]. Cette fois, ce sont les transformations produisant les modèles m_1 et m_2 qui sont masqués par la réduction. La chaîne de transformations est donc considérée comme une unique transformation prenant m_0 en entrée et

9. Donc entre le modèle m_0 et m_3.

produisant m_3 en sortie. Dans ce cas, la trace globale *Gt* utilisée pour pouvoir naviguer d'une trace locale à l'autre n'est plus nécessaire.



(a) Traces non réduites (b) Trace réduite 1, 3 (c) Trace réduite 0, 3

Pour ne pas surcharger les illustrations, nous n'avons pas représenté les concepts de *LocalModel* et *TraceModel*. Ainsi, les liens en pointillés entre deux traces locales représentent directement la navigation d'une trace locale à une autre.

FIGURE 38: Principe de réduction des traces

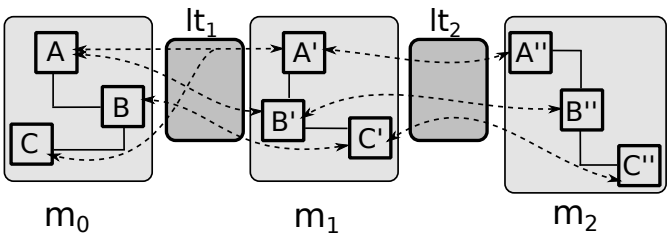
Finalement, en supposant une chaîne de n transformations ($2 \leqslant n$), créant un modèle m_n à partir d'un modèle m_0 et générant n traces locales lt_1 à lt_n, il est possible de produire une trace réduite i, j ($0 \leqslant i < j \leqslant n$). Bien évidemment, une trace réduite i, j avec $j = i + 1$ est équivalent à une trace locale t_j. Par exemple, sur la chaîne de transformations présentée en figure 38a, produire la trace réduite $0, 1$ reviendrait à essayer de réduire uniquement la trace locale lt_1 et ne présente pas d'intérêt.

La construction d'une trace réduite i, j s'effectue par navigations du modèle m_j vers le modèle m_i au travers des différentes traces locales. La figure 39 montre le principe de réduction de la trace sur une chaîne de deux transformations prenant en entrée un modèle m_0 et produisant un modèle m_2 en sortie. Deux traces locales sont générées pour l'occasion : lt_1 et lt_2 [10]. Les traces locales permettent de voir que l'élément *A* du modèle m_0 a été transformé en *A'* dans m_1 grâce à l'élément *C* et que ce même élément *A* a aussi mené à la création de *B'* dans le modèle m_1. Puis, *A'* a été transformé en *A''* dans le modèle m_2 et *B'* en *B''* dans le modèle m_2. Finalement, le dernier élément *C''* du modèle m_2 a été créé à partir de *C'* du modèle m_1, lui-même créé par l'élément *B* du premier modèle m_0.

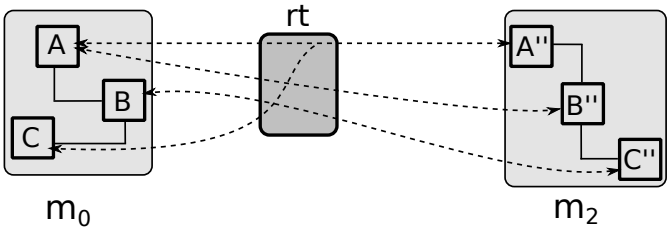
10. Figure 39a

La figure 39b nous montre la trace réduite $0, 2$ pour la chaîne de transformations présentée en figure 39a. Pour construire cette trace réduite *rt*, il faut pouvoir trouver tous les éléments de m_0 utiles à la création d'un élément de m_2. Par exemple, pour trouver les éléments du modèle m_0 qui ont conduit à la création de *A''* dans m_2, il faut, dans un premier temps, naviguer de m_2 vers m_1 grâce à lt_2. L'élément *A'* est retrouvé. Finalement, la navigation depuis m_1 vers m_0 donne les éléments *A* et *C*. Ainsi, *A* et *C* ont conduit à la création de *A''*. De la même manière, en procédant par navigation, on obtient que l'élément qui a mené à la création de *C''* est *B* et que *B''* a été créé à partir de *A*.

Sur cet exemple, sans réduction de la trace, les deux traces locales contiennent au total 6 liens de trace auxquels s'ajoutent, implicitement,



(a) Traces non réduite



(b) Trace réduite 0, 2

FIGURE 39: Principe de réduction des traces

les représentations des éléments qui sont présents dans la trace locale, à savoir les *ModelRef*. Donc, 6 *ModelRef* pour lt_1 et 6 de plus pour lt_2. Les deux traces réduites contiennent donc au total 18 éléments de trace. Après réduction de la trace, il ne reste plus que 3 liens de trace et 6 *ModelRef*, soit 9 éléments de trace.

Pour construire la trace réduite, nous utilisons une simple trace locale. En effet, comme la chaîne des transformations masquées est considérée comme une seule transformation, une simple trace locale est suffisante. En revanche, le concept de règle *RuleRef* n'est pas utilisé. Même si la chaîne de transformations masquée est considérée comme une simple transformation, c'est la séquence de plusieurs règles dans les transformations qui a produit les éléments des modèles de sorties.

3.4 ADAPTATION DE LA TRACE LOCALE POUR QVT

11. Chapitre 1.1

Comme nous l'avons présenté plus tôt[11], les transformations de modèles utilisés dans les chaînes de compilation IDM de GASPARD2 sont développées avec QVTo. Afin de pouvoir bénéficier du support de la trace locale, il faut donc l'adapter au langage QVT. Il est raisonnable de penser que la trace locale qui sera produite doit conserver les mêmes informations que celle de la trace « native » QVT. Cependant, nous avons observé dans celle-ci un ensemble d'informations manquantes, empêchant son utilisation dans un autre but que celui pour lequel elle a été créée.

Pour répondre aux attentes des algorithmes manipulant la trace, nous avons identifié dans [4] les informations manquantes dans la trace QVT et le moyen de les obtenir. Dans un premier temps, nous allons voir, sur un exemple, quelles sont les limitations impliquées par la trace QVT, puis nous verrons comment la trace locale peut être utilisée pour lever ces limitations. Nous terminerons en expliquant ensuite comment la génération de la trace locale a été implémentée à QVTo.

3.4.1 *Limitations de la trace* QVT

La trace QVT est peu documentée dans la spécification proposée par l'OMG. Il est précisé que, pour le langage Operational Mapping, un lien de trace est créé à chaque fois qu'un Mapping est exécuté. Ce lien relie les éléments en entrée et en sortie qui sont spécifiés par la définition d'un Mapping [98, 79].

La trace produite par QVT, même si elle est accessible, n'a pas pour vocation d'être utilisée par un algorithme tierce. Elle est produite dans le but de permettre à l'utilisateur et au moteur de transformation de retrouver des éléments déjà créés par une transformation en cours d'exécution. Ce mécanisme porte le nom de mécanisme de résolution. Il est utilisé automatiquement par le moteur de transformation pour éviter la duplication d'éléments dans le modèle de sortie. Par exemple, si un Mapping est appelé deux fois sur un même élément, la trace est interrogée pour récupérer l'élément déjà créé et ainsi éviter de créer une nouvelle instance. Ce mécanisme, peut aussi être appelé par l'utilisateur en utilisant des mots clefs de la famille des *resolve* [12]. De cette façon, l'utilisateur peut récupérer des éléments (créés ou évalués par le moteur) et les modifier. Compte tenu de son utilisation et des éléments qu'elle contient, la trace QVT peut sembler suffisante pour être utilisée par un algorithme tierce. Cependant, dès que l'on sort du mécanisme de résolution, la trace QVT est bien souvent inefficace. Pour adapter notre trace locale à QVT pour fournir différent supports, nous devons combler ces manques.

12. Comme, par exemple, *resolve*, *resolveOne* ou encore *invresolve* [98].

Pour appuyer nos propos, nous allons présenter à travers 4 scénarios simples de synchronisation de modèles [88] les problèmes rencontrés lorsque l'on utilise la trace QVT. Comme exemple de transformation, nous utilisons la transformation classique [13] : « UML vers MARTE ». Concrètement, le méta-modèle d'entrée de la transformation est UML, utilisé avec le profil MARTE, et le méta-modèle de sortie est le méta-modèle MARTE [102]. Cette transformation est pertinente, car elle est suffisamment complexe pour ne pas être réduite à des Mapping un-pour-un [14]. Par ailleurs, la transformation n'est pas trop compliquée à comprendre, même pour des non-experts du domaine.

13. Cette transformation n'est pas classique par les domaines qu'elle manipule, mais car elle établit un pont entre le monde UML et un formalisme dédié.

14. Prenant un seul élément en entrée et produisant un seul élément en sortie.

Extrait du méta-modèle d'entrée et de sortie

Le profil MARTE, utilisé comme entrée de la transformation, possède beaucoup de concepts. Pour la démonstration, nous nous concentrons sur un extrait du profil sans détailler tous les concepts. De plus, nous ne détaillerons pas la sémantique des concepts abordés étant donné que seule la structure de la trace produite à l'issue de la transformation nous intéresse.

Le premier extrait, présenté en figure 40, contient trois notions principales : *Allocate*, *Distribute* et *Tiler*. Les stéréotypes *Allocate* et *Distribute* étendent l'élément *Abstraction* d'UML. De la même manière, le concept de *Tiler* étend l'élément *Connector* d'UML. Les trois stéréotypes possèdent plusieurs *tagged value*, permettant d'ajouter des informations aux éléments originaux d'UML. Ces *tagged value* sont typées à l'aide des *Datatypes* présent dans le profil, comme, par exemple, *IntegerVector*.

Une fois encore, en raison de la grande quantité d'éléments contenus dans le méta-modèle MARTE [15], la figure 41 regroupe uniquement les concepts relatifs à ceux présentés à la figure 40. Le lien qu'il existe entre un méta-modèle et son profil est très fort. Il est normal de trouver des

15. Méta-modèle de sortie de la transformation.

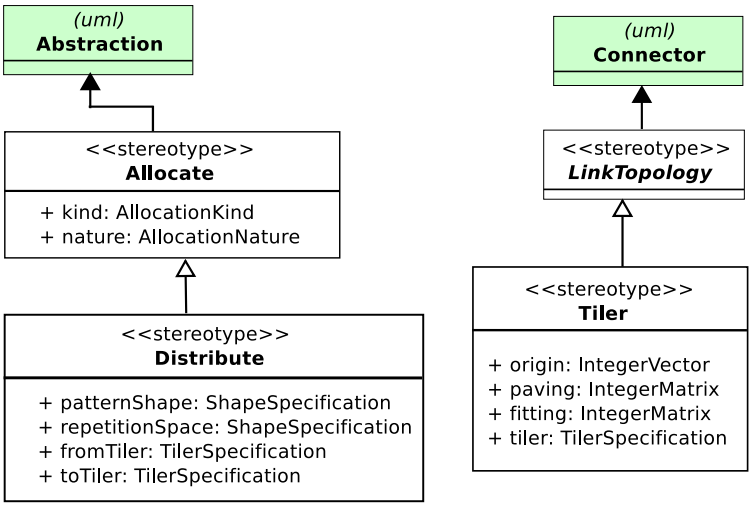


FIGURE 40: Extrait du méta-modèle d'entrée de la transformation UML vers MARTE [99]

concepts communs entre le profil MARTE et le méta-modèle MARTE. On retrouve donc les concepts *Tiler* et *Distribute* dans le méta-modèle MARTE. Toutefois, nous observons que la relation *origin* est considérée comme une relation de composition sur l'élément *IntegerVector* alors qu'*origin* est une *tagged value* dans le profil MARTE. Les autres relations dans cet extrait du méta-modèle sont définies de la même manière ; elles correspondent à une *tagged value* dans le profil MARTE.

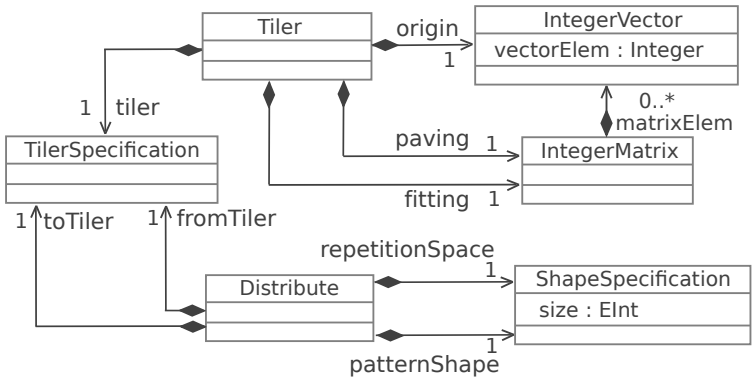


FIGURE 41: Extrait du méta-modèle de sortie de la transformation UML vers MARTE

La transformation UML *vers* MARTE *exprimée en* QVTo

La transformation UML vers MARTE a une taille conséquente : environ 1400 lignes de code QVTo regroupées en 98 règles. Encore une fois, par soucis de clarté, nous ne présentons que les règles manipulant les éléments déjà présentés ici. De plus, les règles ont été élaguées pour ne garder que les parties effectuant les appels aux différentes règles. Elles mettent en évidence, à la fois la correspondance entre les concepts UML et les concepts MARTE, la transformation des attributs et le recours à l'utilisation de HELPER.

Le listing 3.1 montre un extrait du MAPPING *DistributedMapping* manipulant les *Distributes*. La signature de la règle (ligne 54) exprime que

ce MAPPING s'applique sur les instances du concept *Abstraction* (appartenant au méta-modèle UML) et produit des instances de *Distribute* (appartenant au paquetage MARTE RSM du méta-modèle MARTE). La section *when* (ligne 55 à 59) exprime une garde qui spécifie à quelle condition la règle s'exécute. Cette garde impose ici que l'élément *Abstraction* doit porter le stéréotype *Distribute*. Enfin, le MAPPING précise que l'attribut *name* du *Distribute* créé possède la même valeur que l'attribut *name* de l'*Abstraction* UML (ligne 64).

```
54  mapping UML::Abstraction::DistributedMapping() : RSM::
        Distribute
55  when { self
56    .getAppliedStereotypes{'MARTE::MARTE_Annexes::RSM::Distribute
          '} !=
57    null } {

61    name := self.name;

64  }
```

Listing 3.1: La règle *DistributedMapping*

Le listing 3.2 présente un extrait du MAPPING *toTiler* (ligne 103 à 118) créant un *Tiler* à partir d'un *Connector* stéréotypés *Tiler*. L'attribut *origin* détenue par le *Connector* est un *IntegerVector*. Il est créé en appelant le HELPER *str2Vector* sur la *tagged value origin* contenue par le stéréotype *Tiler* (ligne 114 à 115) . Selon la signature du HELPER *str2Vector* (ligne 120), une instance *IntegerVector* est créé à partir d'une *String* [16].

16. Pour obtenir l'*IntegerVector*, le HELPER *str2Vector* effectue un parsing spécifique que nous ne décrivons pas ici.

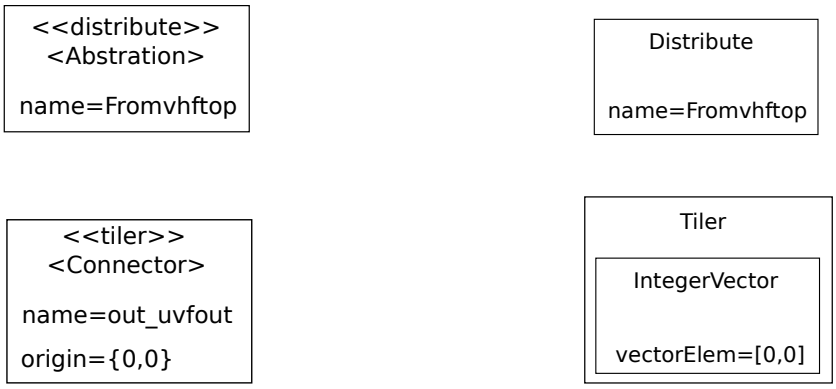
```
103 mapping UML::Connector::toTiler() : RSM::linkTopology::Tiler
        when {
104   not self .getAppliedStereotype('RSM::Tiler') .oclIsUndefined()
          } {

109   origin := self .getValue(stereotype, 'origin')
110   .oclAsType(String).str2Vector();

113 }

115 helper String::str2Vector() : Lib::IntegerVector {

123 }
```

Listing 3.2: Les règles *toTiler* et *str2Vector*

Des scénarios pour la synchronisation de modèles

Pour d'illustrer les limites de la trace QVT dans un contexte où la traçabilité est particulièrement utile, nous définissons quatre scénarios de synchronisation de modèles. Utiliser la synchronisation de modèle comme exemple est pertinent, car ce mécanisme prend la traçabilité comme entrée principale pour son algorithme [88]. De plus, les types d'informations qu'il faut chercher dans la trace pour pouvoir jouer les scénarios sont fréquemment manipulés dans d'autres contextes.

Afin de présenter et pouvoir « jouer » ces scénarios, un modèle d'entrée et un modèle de sortie ont été définis. Ils sont, encore une fois, partiellement représentés en figure 42. Seuls les instances des concepts manipulés par les règles décrites précédemment sont présentées.



(a) Modèle d'entrée UML profilé MARTE

(b) Modèle MARTE produit

FIGURE 42: Modèles d'entrée et de sortie de la transformation UML vers MARTE

17. Figure 42a.

L'extrait du modèle d'entrée [17] se concentre sur deux éléments : une *Abstraction* stéréotypée *Distribute* et un *Connector* stéréotypé *Tiler*. L'*Abstraction* porte le nom *Fromvhftop*. Le *Connector*, quant à lui, se nomme *out_uvfout* et possède sa *tagged value origin* initialisée à $\{0, 0\}$. En sortie de la transformation UML vers MARTE, lorsque le modèle présenté en figure 42a est pris en entrée, le modèle MARTE correspondant, présenté en figure 42b, est produit. Sur ce modèle, on y retrouve un concept *Distribute* possédant le même nom que l'*Abstraction* du modèle UML d'entrée. On retrouve aussi le *Tiler* contenant un *IntegerVector* initialisé à $[0, 0]$, à l'image du *Connector* dans le modèle d'entrée.

Pour les scénarios que nous allons présenter, la stratégie de synchronisation de modèle appliquée ici est simplifiée à deux règles. Nous sommes, bien évidemment, conscient qu'un mécanisme de synchronisation de modèles est, en pratique, beaucoup plus complexe et soulève plusieurs questions. Cependant, cette stratégie simplifiée est suffisante pour illustrer nos propos.

1. Si un élément est supprimé dans le modèle de sortie, alors les éléments d'entrée qui ont conduit à sa création doivent être supprimés. Si ces derniers sont impliqués dans la génération d'autres éléments dans le modèle de sortie, ils doivent également être supprimés.
2. Si un élément est modifié dans le modèle de sortie, alors les éléments qui ont conduit à sa création doivent être modifiée en conséquence. Si ces derniers sont impliqués dans la génération d'autres éléments dans le modèle de sortie, ils doivent aussi être modifiés.

Conformément à ces deux règles énoncées, comme énoncé précédemment, nous proposons quatre scénarios spécifique représentant une action de suppression ou de modification sur les éléments du modèle de sortie [18].

18. Figure 42b

Scénario 1. Suppression de l'instance de *Distribute*.

Scénario 2. Modification de l'attribut *name* porté par l'instance de *Distribute*.

Scénario 3. Suppression de l'instance *IntegerVector* portée par l'instance de *Tiler*.

Scénario 4. Modification de l'attribut *vectorElem* porté par l'instance *IntegerVector*.

Nous essayons maintenant de « jouer » ces scénarios en utilisant la trace QVT produite pendant la transformation.

La synchronisation de modèles en utilisant la trace QVT

Comme nous l'avons déjà évoqué, le but de cette section n'est pas de présenter une solution aux problèmes de synchronisation de modèle, mais de présenter, informellement, les limitations de la trace QVT même sur des exemples simples. Nous utilisons la trace QVT pour synchroniser les modèles en suivant, successivement, les quatre scénarios.

Celle-ci est, partiellement, représentée en figure 43. Seul les parties de la trace relatives aux éléments présents dans les modèles d'entrée et de sortie [19] sont dessinés. Cette trace contient deux liens. Le premier exprime que l'instance *Abstraction* nommée *Fromvhftop* conduit à la création du *Distribute Fromvhftop* en exécutant le MAPPING *DistributedMapping*. Le second lien spécifie que le MAPPING *toTiler* à utilisé le *Connector out_uvfout* pour créer une instance de *Tiler*.

19. Figure 42a et figure 42b

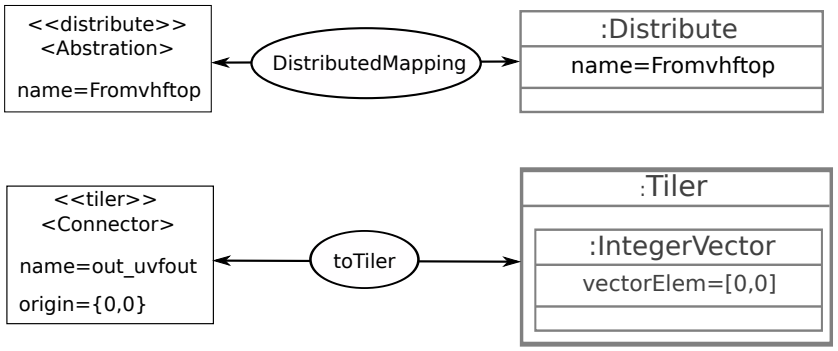


FIGURE 43: Vue d'un extrait de la trace QVT produite

Nous utilisons maintenant cette trace pour réaliser les quatre scénarios de synchronisation.

Scénario 1 : L'instance du *Distribute* est supprimée du modèle de sortie. Pour synchroniser le modèle d'entrée au nouveau modèle de sortie, les éléments du modèle d'entrée qui ont mené à la création du *Distribute* doivent être supprimées. En analysant la trace QVT, il apparaît que seul l'instance d'*Abstraction* a permis cette création grâce à l'opération *DistributedMapping*. La synchronisation est, ici, plutôt simple à réaliser : l'instance d'*Abstraction* est supprimée dans le modèle d'entrée.

Scénario 2 : La trace QVT ne fournit pas suffisamment d'informations pour déterminer les éléments qui ont mené à la création de l'attribut *name*. La synchronisation semble, ici, difficile à effectuer.

Scénario 3 : Synchroniser les modèles dans ce scénario revient à supprimer les éléments du modèle d'entrée impliqués dans la création de l'*IntegerVector*. Comme pour le scénario précédent, ces éléments ne peuvent pas être identifiés en utilisant la trace QVT. La synchronisation est donc compromise.

Scénario 4 : Une fois encore, la synchronisation est difficilement réalisable puisque la trace QVT ne fournit pas de liens de trace entre l'attribut modifié et les éléments menant à sa création.

Ces quatre scénarios illustrent le manque de liens de trace, entre les éléments d'entrée et de sortie, contenues par la trace QVT. Seul le premier scénario est réalisable avec une telle trace. Les quatre scénarios montrent que, comme les concepts, les attributs doivent être tracées. En effet, la synchronisation du modèle peut également impliquer des attributs dans les modifications à opérer [20]. Par ailleurs, le scénario 3 montre que certains concepts (par exemple IntegerVector) peuvent être créés en utilisant une règle de HELPER plutôt qu'un MAPPING. Ainsi la traçabilité pour QVT doit aussi prendre en compte les HELPER et pas seulement les MAPPING. Finalement, ces quatre scénarios montrent sur des actions très communes souvent appliquées sur un modèle [21] que la trace QVT ne possède pas une granularité assez fine.

20. Comme cela est le cas pour les scénarios 2 et 4.

21. *i.e.* modification et/ou suppression

Nous avons montré dans cette section que la trace QVT ne possédait pas une granularité assez fine pour pouvoir être utilisée par un mécanisme de synchronisation de modèles. La synchronisation de modèle n'est pas le seul exemple où l'utilisation d'une trace de transformation différente est requise. En effet, dans [136], les auteurs proposent d'effectuer du *debugging* de transformations exprimée en QVT et plus particulièrement en QVTR.

Une autre application de la trace QVT est étudiée en [78]. Les auteurs effectuent de l'analyse d'impact en utilisant leur propre formalisme de trace plutôt que celui proposé par QVT à cause de la façon dont la trace QVT est générée [22] et du manque d'informations contenue dans celle-ci.

22. *i.e.* Accessible et sauvegardée une fois que la transformation est terminée.

3.4.2 *Les manques de la trace* QVT

Le langage QVT OPERATIONAL MAPPING fournit une politique de génération de la trace reposant uniquement sur les opérations de MAPPING. Si l'on suit la spécification QVT, des instances de *trace class* sont automatiquement créées lorsque des opérations de MAPPING sont exécutées. Ces éléments tracés font référence seulement aux éléments d'entrée et de sortie exprimés dans la signature du MAPPING. L'exemple de synchronisation de modèle, que nous avons précédemment présenté, nous a montré les limitations de cette politique et suppose que d'autres éléments du langage QVT devraient être tracés.

Les sections when

Les langages OPERATIONAL MAPPING et RELATION fournissent un mécanisme de garde à travers le mot-clef *when*. Cette garde indique que la règle est exécutée seulement si la condition de la garde est satisfaite. Donc, la création d'un élément dépend, non seulement, de l'élément sur lequel la règle s'applique, mais aussi des éléments et attributs impliqués dans la garde. Ainsi, tracer tous les éléments impliqués dans la création d'un autre élément peut être très utile [23]. Le manque de la trace des sections *when* est aussi déploré par Kurtev et al. dans [78] où ils avouent que leur analyse d'impact est limitée à cause de cela.

23. Comme illustré dans l'exemple de synchronisation de modèles.

Les opérations QUERY *et* HELPER

En plus des opérations de MAPPING, le langage OPERATIONAL MAPPING propose deux autres types d'opérations : les QUERY et les HELPER.

QUERY La QUERY est une opération associée à une simple requête. Elle ne créée, ni ne modifie ou supprime d'éléments. Pour cette raison, nous ne jugeons pas utile de tracer les QUERY.

HELPER En revanche, les HELPER peuvent créer, supprimer ou modifier des éléments par effets de bords [98]. De plus, ils peuvent être appliqués plusieurs fois sur un même élément pour, potentiellement, créer plusieurs instances différentes. Comme nous l'avons vu sur l'exemple de synchronisation de modèles, laisser de côté les HELPER implique une absence d'informations concernant les éléments qu'ils ont créé, modifié ou supprimé.

Les attributs

Les attributs aussi n'apparaissent pas dans la trace QVT comme ils ne sont pas utilisés par le mécanisme de résolution d'objets proposé par QVT. Cependant, comme illustré dans l'exemple de synchronisation de modèles, garder des informations concernant la transformation des attributs peut être réellement utile.

Bien évidemment, les remarques et les observations que nous avons données ici pour le langage OPERATIONAL MAPPING sont applicables pour le langage QVTR puisque la politique employée pour tracer les éléments est relativement semblable à celle du langage OPERATIONAL MAPPING.

3.4.3 *Adaptation de la trace locale pour* QVTo *et levée de limitations*

Nous montrons dans les sous sections suivantes comment les éléments de la trace locale fournissent les informations nécessaires au moteur QVTo pour tracer les parties du langage QVT actuellement laissées de côté par la trace native de QVT.

Les opérations de MAPPING

La trace locale doit pouvoir fournir les mêmes informations que celles déjà présentes dans la trace QVT. Ainsi, les opérations de MAPPING sont tracées en utilisant le concept de *RuleRef* de la trace locale. Les éléments manipulés et produits par le MAPPING sont, eux, tracés avec le concept de *ModelRef*, donc, soit en utilisant un *ClassRef*, soit un *PrimitivePropertyRef*.

Les sections when

Les gardes d'un MAPPING sont des expressions booléennes. Elles sont définies en fonction d'éléments des modèles manipulés par la transformation. Si une partie de la garde est relative à une valeur d'attribut, le concept *PrimitivePropertyRef* de la trace locale est utilisé alors que le concept *ClassRef* sera utilisé si la garde est exprimée en fonction d'une classe du modèle.

Les opérations de HELPER

La trace des HELPER est construite de la même manière que les MAPPING. Comme nous l'avons expliqué précédemment, nous ne faisons pas de différences entre les MAPPING et les HELPER. C'est donc par le biais d'un *RuleRef* et de *ModelRef* que le HELPER sera tracé.

Les attributs

En utilisant la trace locale, il est possible de tracer les attributs en utilisant le concept de *PrimitivePropertyRef*. Les attributs sont tracés dans les corps des MAPPING et des HELPER.

Dans cette section, nous avons identifié les concepts de la trace locale qui peuvent accueillir les différentes informations de trace qu'il est nécessaire de relever lors de l'exécution d'une transformation QVT. Nous revenons sur l'exemple de synchronisation de modèles et sur la transformation UML vers MARTE pour vérifier que les précédentes limitations de la trace QVT sont bien levées en utilisant la trace locale.

Retour sur la synchronisation de modèles en utilisant la trace locale

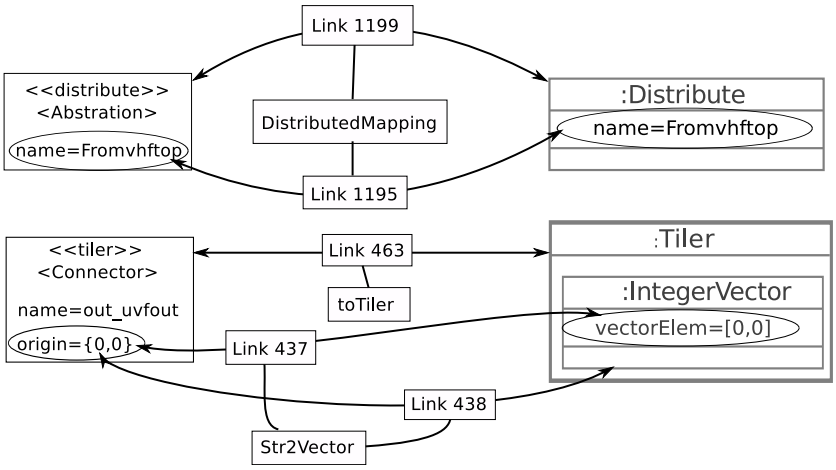


FIGURE 44: Trace locale générée pour la transformation UML vers MARTE

La figure 44 présente la trace locale générée par la transformation UML vers MARTE [24]. On retrouve les informations originalement présentes dans la trace QVT, à savoir que la règle *DistributedMapping* a utilisé l'*Abstraction* pour créer le *Distribute* [25]. De même, on retrouve bien un lien de trace *Link* 463 entre le *Connector* et le *Tiler* représentant la création effectuée par la règle *toTiler*. En plus des informations originellement utilisées dans la trace, on peut voir que le HELPER *Str2Vector* est tracé. Il référence les *Link* 437 et 438 indiquant que l'attribut *origin* a permis de créer à la fois l'élément *IntegerVector* et la valeur de son attribut *vectorElem*. Nous pouvons aussi remarquer que l'attribut *name* de l'*Abstraction* est relié par le *Link* 1195 à l'attribut *name* du *Distribute*.

En utilisant la trace locale produite pendant la transformation, nous essayons à nouveau de réaliser les synchronisations de modèles en suivant les différents scénarios.

24. Toujours en utilisant les modèles présentés en figure 42, page 70.

25. Matérialisé, ici, par le *Link* 1199.

Scénario 1 : La trace locale précise que l'instance du *Distribute* est associée à un seul lien (*Link* 1199) possédant un unique élément comme source : l'instance d'*Abstraction Fromvhftop*. Ainsi, dans ce cas, pour synchroniser le modèle et en adoptant les stratégies précédemment décrites, l'instance d'*Abstraction* est retirée du modèle d'entrée.

Scénario 2 : La trace locale et plus précisément, le *Link* 1195, permettent d'identifier l'attribut *name* de l'instance d'*Abstraction* dans le modèle d'entrée comme l'élément qui a créé l'attribut *name* de l'instance de *Distribute*. Ainsi, pour effectuer la synchronisation de modèles, l'attribut *name* de l'instance d'*Abstraction* doit être modifié.

Scénario 3 : Le *Link* 438 de la trace locale met en évidence que l'instance d'*IntegerVector* a été générée à partir de l'attribut *origin* portée par l'instance du *Tiler*. Ainsi, la suppression de l'instance d'*IntegerVector* conduit à vider l'attribut *origin* du modèle d'entrée.

Scénario 4 : La trace locale exprime, avec le *Link* 437, que création de l'attribut *vectorElem* repose sur l'attribut *origin* porté par l'instance du *Tiler*. La modification appliquée sur l'attribut *vectorElem* affecte l'attribut *origin* dans le modèle d'entrée.

Cette fois, en utilisant la trace locale, les quatre scénarios ont pu être réalisés avec succès. En effet, chaque action effectuée sur le modèle de sortie concerne des éléments [26] qui sont tracés. Il est donc simple de déterminer quels éléments du modèle d'entrée les ont créés et d'appliquer les modifications adéquates pour synchroniser les modèles.

26. *i.e.* des classes ou des attributs.

Dans cette section, nous avons identifié les concepts de la trace locale qui peuvent accueillir les différentes informations qu'il est nécessaire de relever lors de l'exécution d'une transformation QVT. De plus, nous avons aussi montré sur l'exemple de synchronisation de modèle que la trace locale permet de lever les limitations de la trace QVT. Nous allons maintenant présenter comment la trace locale est générée.

3.4.4 *Génération*

Comme nous l'avons évoqué au chapitre 2.1.3, plusieurs techniques existent pour générer les liens de traçabilité. Parmi les différentes techniques, nous utilisons la capture des liens de traçabilité implicite, d'une part, pour sa grande flexibilité et, d'autre part, pour sa non modification de la transformation à tracer [27]. Pour mettre en œuvre la capture des liens de trace, nous avons utilisé le moteur d'une implémentation du langage OPERATIONAL MAPPING : QVTo [28].

27. À l'inverse des liens de trace explicite.

28. Implémenté en JAVA en utilisant le framework EMF.

La figure 45 représente une partie du modèle l'architecture du moteur qui est utilisé pour exécuter une transformation QVTo. Afin de ne pas surcharger le modèle, l'interface graphique permettant à l'utilisateur d'exécuter une transformation est représentée par la classe *QvtLauncherUI*. Cette classe, en réalité composée de plusieurs, nous permet de mettre en évidence le lien entre l'interface graphique et le moteur. Cette classe possède une référence à la classe *InternalTransformationExecutor* servant à charger et lancer l'évaluation d'une transformation QVTo via les méthodes *LoadTransformation* et *execute*. Une instance de la classe *InternalTransformationExecutor* est créée pour une instance d'exé-

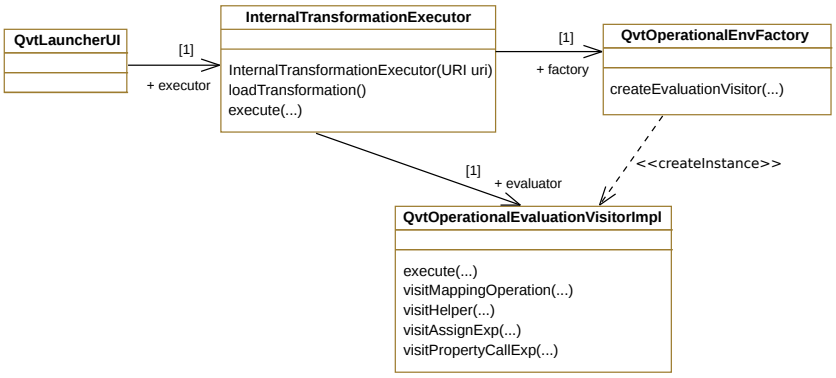


FIGURE 45: Extrait de l'architecture du moteur QVTo-3.1.0

cution d'une transformation en particulier. Son constructeur prend en paramètre le chemin de la transformation QVT à exécuter. Pour pouvoir lancer une exécution de la transformation sur des modèles d'entrée, la classe *InternalTransformationExecutor* demande à la classe *QvtOperationalEnvFactory* de lui fournir une instance capable d'évaluer la transformation. Ainsi, pour fournir l'exécution de la transformtion, une instance de la classe *QvtOperationalEvaluationVisitorImpl* est créée via la méthode *createEvaluationVisitor* de la classe *QvtOperationalEnvFactory*. Cette classe évalue la transformation en utilisant la méthode *execute*. Cette dernière appelle les méthodes :

- *visitMappingOperation*, utilisée pour évaluer un MAPPING,
- *visitHelper*, utilisée pour évaluer un HELPER,
- *visitAssignExp*, utilisée pour évaluer, entre autre, les attributs,
- *visitPropertyCall*, utilisée pour évaluer les collections et opérations spécifique sur des collections ou des attributs.

Comme ces méthodes fournisses les briques d'évaluation de la transformation, elles peuvent être utilisées pour récupérer les informations nécessaires à la trace locale.

Afin de récupérer ces informations sans pour autant modifier le moteur de transformation QVTo, nous avons étendu les classes précédemment présentées tel que cela est montré en figure 46. Toutes les classes de la figure 45 ont été étendues. Ainsi, *LocalTraceQvtLauncher* étend *QvtLauncherUI*, *TracedInternalTransformationExecutor* étend *InternalTransformationExecutor*, *QvtoLocalTrace* étend *QvtOperationalEvaluationVisitorImpl* et *QvtoLocalTraceEvaluationEnv* étend *QvtOperationalEnvFactory*. Ce sont ces classes qui vont être utilisées pour, à la fois, lancer la transformation et générer la trace locale. La classe responsable de l'exécution de la transformation : *TracedInternalTransformationExecutor* est appelée par *LocalTraceQvtLauncher*. La méthode *execute* qu'elle possède, surchargeant celle de la classe *InternalTransformationExecutor*, appelle donc la classe *QvtoLocalTraceEvaluationEnv* pour créer une instance *QvtoLocalTrace*. La création de cette instance s'effectue via la méthode surchargée *createEvaluationVisitor*. L'instance de *QvtoLocalTrace* créée exécute ensuite la transformation tout en produisant la trace locale.

La classe *QvtoLocalTrace* est la plus importante. En effet, elle surcharge les méthodes assurant l'évaluation de la transformation pour fournir le support à la trace locale sans occulter l'évaluation originale effectuée par le moteur. Le listing 3.3 montre comment est effectué l'appel à la

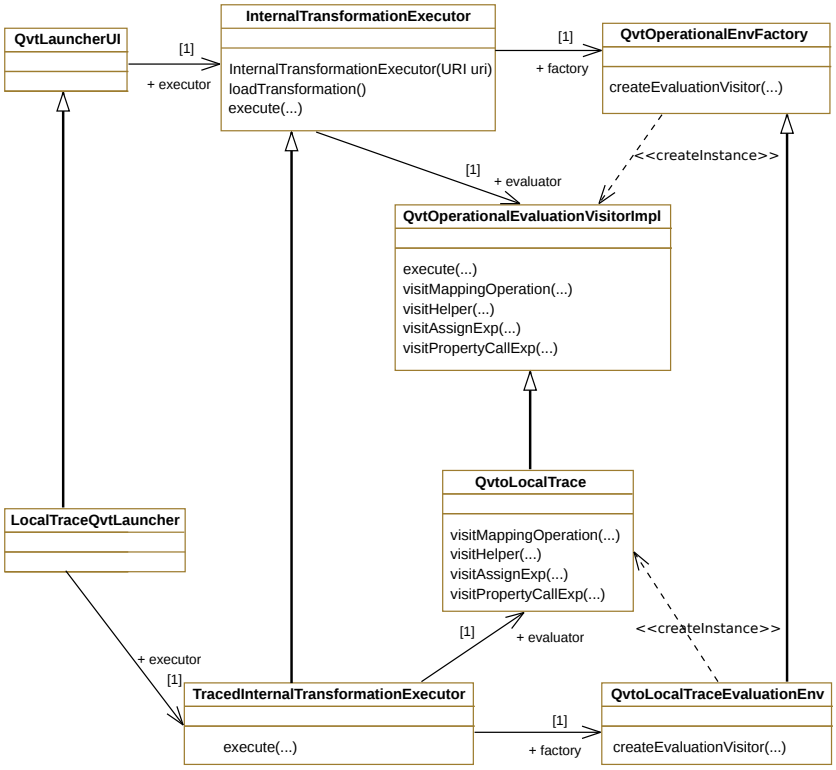


FIGURE 46: Architecture de l'architecture du moteur QVTo-3.1.0 modifiée

méthode surchargée originale et où se place le code relatif à la trace locale en prenant la méthode *visitMappingOperation* comme exemple.

```
1  public Object visitMappingOperation(MappingOperation
       mappingOperation)
2  {
3    Object res = super.visitMapping(mappingOperation);
4    // récupération de la règle utilisée et création d'un concept
       RuleRef dans la trace locale
5    // récupération des éléments sources et des éléments
       destinations produits et création des ModelRef dans la
       trace locale
6    // création d'un concept Link dans la trace locale
7    // liaisons entre le Link, les ModelRefs et le RuleRef créé
8    // sauvegarde de la trace locale
9    return res;
10 }
```

Listing 3.3: Surcharge du visiteur de MAPPING

La méthode présentée au listing 3.3 est la méthode *visitMappingOperation* surchargée de la classe *QvtoLocalTrace*. Le code des méthodes surchargées possède toujours la même structure. L'appel à la méthode originale de la super classe s'effectue à la ligne 3 avec l'utilisation du *super*. Le résultat de l'évaluation originelle est ainsi calculé et retourné en ligne 9. Finalement, le code concernant la création de la trace locale est inséré au niveau des lignes 4 à 8. Dans cette règle, la création des concepts dans la trace locale s'effectue en 5 étapes. Dans un premier temps, la règle en cours d'exécution est récupérée et un concept *RuleRef* de la trace locale et produit (ligne 4). Ensuite les éléments manipulés par la règle en cours d'exécution sont récupérés et un concept *ModelRef*

et produit pour chacun des éléments (ligne 5). Une fois les différents éléments de la transformation capturés, un *Link* est créé dans la trace locale (ligne 6) et les concepts précédemment créés lui sont attaché (ligne 7). Une fois l'ajout terminé, la trace locale mise à jour est ensuite sauvegardée (ligne 8). Nous pouvons voir qu'en utilisant le concept de surcharge, la gestion de la trace locale est ajouté à la méthode sans que celle-ci perde son comportement initial.

Dans cette section, nous avons vu comment générer la trace locale pour une transformation QVTo en nous appuyant sur le code du moteur de transformation. Étant donné l'indépendance de la trace locale vis-à-vis de tout langage de transformation, il est possible de l'adapter à d'autres langages. Durant nos expérimentations, nous avons été amené à devoir produire la trace locale pour Kermeta et MoMote, un moteur de transformations écrit en Java et utilisant le framework EMF. Dans ces deux cas, pour éviter d'avoir à composer avec le moteur du langage de transformation, la production s'est faite par l'utilisation des liens de trace explicite et des lignes de code produisant les traces ont donc été insérées dans les transformations.

3.5 LA TRACE LOCALE POUR LES TRANSFORMATIONS LOCALISÉES

Notre mécanisme de traçabilité repose sur trois principaux artefacts :
- une trace locale,
- une trace globale,
- une trace réduite.

En utilisant la trace globale, il est possible de gérer la trace sur l'ensemble d'une chaîne de transformations. Elle assure la navigation entre les différents modèles transformés et les traces locales produites. Ce mécanisme est entièrement fonctionnel pour les chaînes de transformations classiques, mais lorsque que l'on travaille avec des transformations localisées, il est nécessaire d'effectuer certains ajustements pour pouvoir continuer à naviguer entre les modèles et les traces.

3.5.1 *Les traces avec les transformations localisées*

Les transformations localisées proposent de travailler avec des transformations *sur place* pour pouvoir facilement séparer les préoccupations lors de la création de chaînes de transformations [29]. Comme pour des transformations classiques, il est possible de générer des traces pour ce type de transformation. Actuellement, le moteur de chaîne de transformations que nous utilisons dans Gaspard2 se sert de transformations localisées écrites en QVTo. Nous avons donc utilisé le même moteur de traçabilité que celui que nous avons développé pour QVTo.

29. *C.f*, chapitre 1

Cependant, dès que l'on travaille avec une succession de transformations localisées, même si la chaîne globale peut être perçue comme une chaîne de transformations classiques, l'utilisation successive de transformation *sur place* entraîne quelques soucis dans la gestion de la trace.

La figure 47 illustre, sur une chaîne de trois transformations, le problème rencontré par la trace lorsque l'on travaille avec des chaînes de transformations localisées. Lors de l'exécution de la chaîne de transformation, le premier modèle m_0 est copié en modèle m_1, puis la transformation t_1 est exécutée. Cette transformation modifie le modèle m_1 et produit la trace locale lt_1. Le modèle m_1 modifié est copié à son

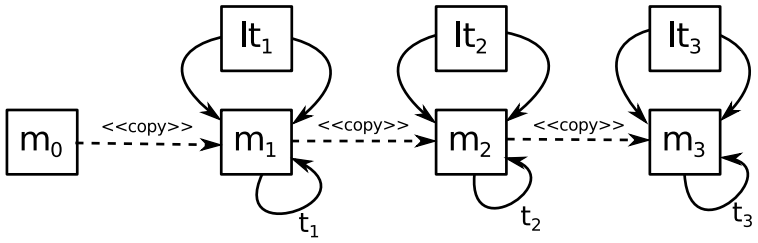
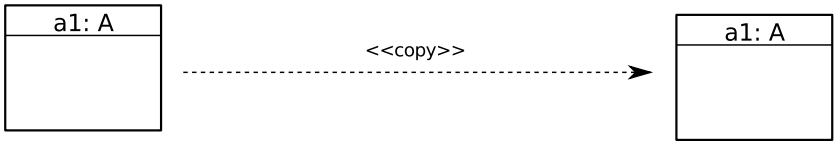
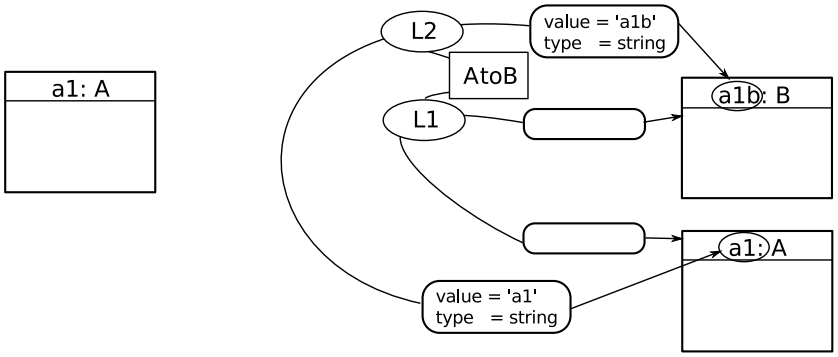


FIGURE 47: Exemple de trace locales générées pour une chaîne de 3 transformations localisées

tour en un modèle m_2 qui est directement modifié par la transformation t_2 produisant la trace locale lt_2. Finalement, le modèle modifié m_2 est dupliqué vers le modèle m_3 directement modifié par la dernière transformation t_3. Cette dernière transformation produit la trace locale lt_3. Il est possible de voir que chaque trace locale produite fait référence uniquement au modèle considéré par la transformation *sur place* à un moment donné. Ainsi, de manière générale, la trace locale lt_n produite lie uniquement des éléments du modèle m_n entre eux. Dans ces conditions, il est difficile de pouvoir naviguer entre les différents modèles d'une chaîne de transformations. Par exemple, s'il faut naviguer d'un élément du modèle m_3 vers un élément du modèle m_2, les liens de traces qui seront récupérés ne permettent pas de passer du modèle m_3 au modèle m_2. Afin de mieux comprendre pourquoi la trace locale lie uniquement des éléments du même modèle, nous détaillons une trace locale produite sur un exemple.



(a) Copie du modèle initial



(b) Exécution de la transformation et production de la trace locale

FIGURE 48: Problème lié à la construction de la trace locale pour les transformations localisées

La figure 48 illustre, sur un exemple simple, la transformation d'un modèle contenant un seul élément en utilisant une transformation localisée. Lors d'une première étape, le modèle initial est copié. Ainsi, l'élément *a1 : A* est de nouveau présent dans un nouveau modèle. Une fois que la transformation localisée est exécutée, un élément *a1b : B* a été

créé. La trace locale produite conserve un ensemble de relations sur le même modèle puisque la transformation exécutée est une transformation de *sur place*. Ainsi, la trace locale précise que l'élément *a1b : B* a été produit à partir de l'élément *a1 : A*, copie de l'élément *a1 : A* du modèle initial. Naviguer entre les modèles d'une chaîne de transformations en passant par les traces de transformations devient alors impossible.

En utilisant les transformations localisées, les traces de transformations de modèle agissent comme des transformations *sur place*. Elles modifient donc des modèles mais n'en créé pas de nouveau. On se retrouve donc avec le même problème que celui énoncé au chapitre 2 pour les traces dans les transformations *sur place*. Il devient difficile de gérer et de se servir correctement de la trace locale[30].

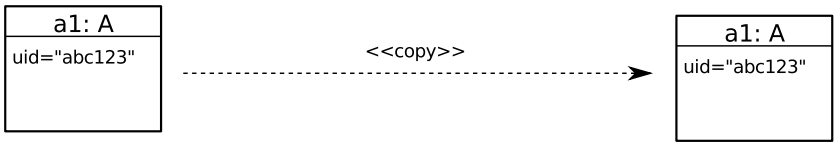
30. Ce problème n'est pas exclusif à la trace locale, mais à tout mécanisme de traçabilité.

3.5.2 *L'utilisation d'*UIDs *pour la contruction de la trace locale*

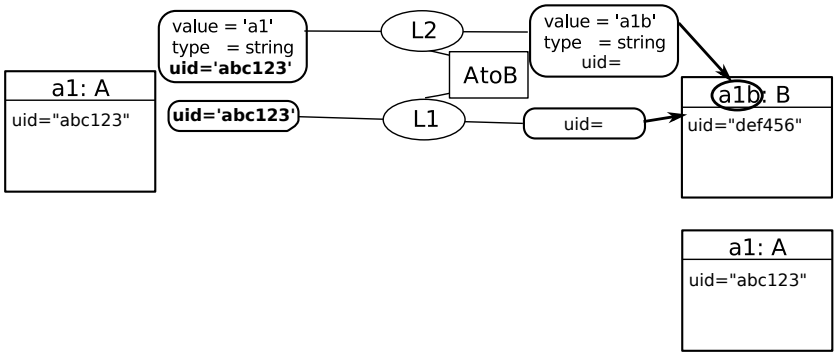
Pour manipuler au mieux la trace locale dans le contexte de transformation localisée, l'idée générale est de considérer les transformations localisées comme des transformations *in* vers *out*. Sur l'exemple présenté en figure 47, cela reviendrait à considérer la trace lt_1 entre les modèles m_0 et m_1, la trace lt_2 entre les modèles m_1 et m_2 et la trace lt_3 entre les modèles m_2 et m_3.

Pour produire une telle trace, nous proposons d'utiliser des UIDs. Ils correspondent à des identifiant uniques portés par les éléments de modèle permettant de les représenter. Ainsi, il est possible de faire référence à des éléments de modèles, même si aucune liaison n'existe. Un élément présent dans un modèle peut donc faire référence à un autre élément dans un autre modèle s'il conserve l'UID de cet élément. La trace locale est donc construite, à la fois, en fonction des UIDs des éléments d'entrée manipulés et des éléments créés et/ou modifiés.

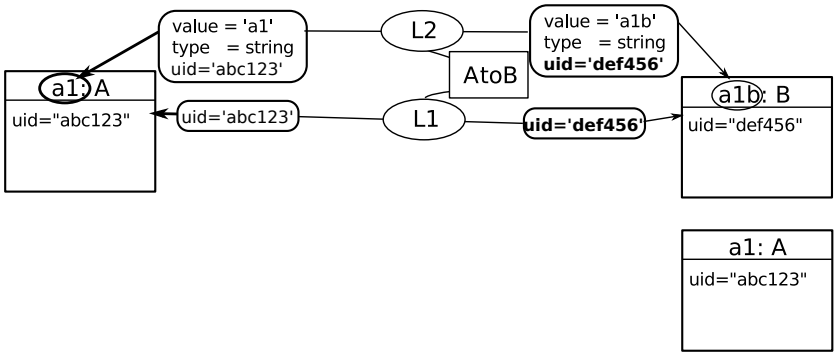
La figure 49 présente la construction de la trace locale dans le contexte de l'exemple présenté en figure 48 en utilisant les UIDs. Le modèle d'entrée de la transformation ne contient qu'un seul élément *a1 : A* possédant un UID : *abc123*. Dans un premier temps, le modèle est copié (figure 49a), puis la transformation est lancée. Le résultat de la transformation ainsi que la trace locale produite sont visibles sur la figure 49b. L'élément *a1 :A* a été transformé en *a1b : B*. La trace locale est générée asymétriquement. Sa partie gauche, conservant le lien aux éléments consommés par la transformation, possède seulement des références à des éléments en utilisant les UIDs et sa partie droite, conservant les liens aux éléments produits par la transformation, lie directement des éléments du modèle de sortie de la transformation. De cette façon, aucune référence vers l'élément *a1 : A* issue de la copie du modèle n'est gardée. Une étape finale de résolution (figure 49c) permet de lier explicitement les éléments de la trace locale au modèle initial et ainsi de considérer la trace locale comme un ensemble de relations entre deux modèles différents plutôt qu'un ensemble de relations dans le même modèle. Pour effectuer cette résolution, les éléments portant les UIDs stockés par les *ModelRef* des *srcContainer* sont cherchés dans les modèles d'entrée de la transformation, puis les *eObject* de chaque *ModelRef* des *destContainer* sont tous inspectés pour récupérer leurs UIDs.



(a) Copie du modèle initial



(b) Exécution de la transformation et production de la trace locale



(c) Résolution des références et des UIDs

FIGURE 49: Construction de la trace locale pour les transformations localisées en utilisant les UIDs

3.5.3 *Retour sur les transformations sur place*

Si l'on regarde la solution reposant sur les UIDs utilisée pour gérer la trace pour les transformations localisées, on voit que finalement, une transformation localisée, qui est en réalité une transformation *sur place*, est considérée comme une transformation *in* vers *out*. Lorsque la transformation localisée est exécutée, deux modèles distincts sont produits. Le premier correspond au modèle initial et le second à celui consommé et modifié par la transformation. La trace entre les deux modèles relatent des éléments qui ont été modifiés et créés par la transformation. Il est possible d'adapter cette façon de gérer les transformations et la trace à une simple transformation *sur place*. Ainsi, lorsqu'une transformation *sur place* est lancée, au lieu de directement exécuter la transformation sur le modèle d'entrée, une copie du modèle d'entrée est effectuer afin de gérer la traçabilité comme nous l'avons présenté dans les sections précédentes. De cette façon, cela permet de considérer la transformation *sur place* comme une transformation *in* vers *out*. De cette façon, le problème que nous avons énoncé au chapitre 2, engendré par les traces de transformations *sur place* est résolue.

3.5.4 *Recherche pour les traces de transformations localisées*

L'utilisation de transformations localisées oblige à repenser les algorithmes de recherche dans les chaînes. En effet, les éléments apparaissant dans les modèles de sortie d'une transformation localisée ne sont pas obligatoirement des éléments produit à partir d'autre. Comme les liens de traçabilité ne tisse des liens de trace que lorsque des éléments sont créés ou modifiés, il n'existe pas forcément de liens de trace entre les éléments d'un modèle d'entrée et de sortie. La figure 50 présente une chaîne simple de trois transformations localisées avec un ensemble minimal d'élément et les liens de trace locale tissés.

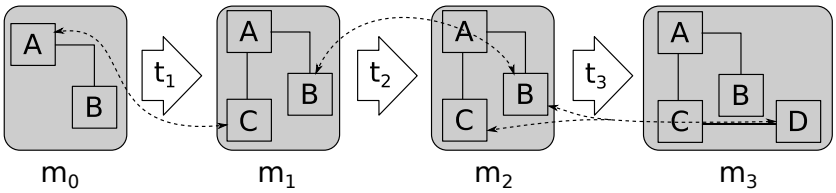


FIGURE 50: Recherche d'élément dans une chaîne de transformations

Sur cet exemple, la première transformation t_1 produit un élément *C* dans le modèle m_1 à partir de l'élément *A* du modèle m_0. Les éléments *A* et *B* du modèle m_1 sont issus de la copie du modèle m_0. Ainsi, aucun lien de trace n'apparaît entre ces éléments. La seconde transformation t_2 modifie uniquement l'élément *B*. Une fois encore, les éléments *A* et *C* apparaisse dans le modèle m_2 due à la copie du modèle m_1 en m_2. La dernière transformation t_3 se sert des éléments *B* et *C* pour produire un élément *D*.

Avec cette chaîne de transformations, lorsque l'on veut trouver, à partir du modèle m_3, par exemple, les éléments dans m_0 qui ont produit *D*, il faut commencer par suivre le lien de trace partant de *D*. Les éléments récupérés sont *B* et *C* dans le modèle m_2. Ensuite, les liens de trace partant de ces éléments vers ceux du modèle m_1 sont suivis. Aucun lien de trace vers le modèle m_1 n'est trouvé pour l'élément *C*. Pour *B*, un lien de trace menant vers l'élément *B* du modèle m_1. De cet élément, aucun lien de trace vers le modèle m_0 n'est trouvé. À partir d'un élément dans le modèle m_3, en suivant strictement les liens produits lors de l'exécution d'une chaîne de transformations localisées, il est peu probable de trouver quels sont les éléments dans le modèle m_0 qui l'ont produit.

Principe de recherche dans les transformations localisées

Sur l'exemple présenté en figure 50, en considérant la recherche des éléments du modèle m_0 ayant produit l'élément *D* dans m_3, il faut pouvoir récupérer les éléments *A* et *C*. Pour pouvoir récupérer le bon ensemble d'éléments, il faut garder au fur et à mesure tous les éléments trouvés dans les modèles intermédiaires et vérifier leur présence dans le dernier modèle accessible. En reprenant l'exemple de la figure 50, pour récupérer les éléments du modèle m_0 ayant produit *D* du modèle m_3, le lien de trace partant de *D* est navigué. Les éléments *B* et *C* sont récupérés et conservés. Ensuite, les liens de trace pointant vers *B* et *C* dans m_2 sont navigués. L'élément *B* est trouvé en remontant les liens de trace et C est conservé, car il existe aussi dans m_1. Finalement, les liens de trace de la transformation t_1 sont observés. Pour l'élément : *C*,

l'élément *A* est sélectionné dans m_0. Quant à l'élément *B*, il est cherché et trouvé dans le modèle m_0. Donc, il est aussi sélectionné. L'ensemble d'élément de m_0 ayant produit *D* dans m_3 contient donc A, B.

Algorithmes de recherche

Afin de récupérer les *descendants* et les *ancêtres* dans une chaîne utilisant des transformations localisées, nous fournissons deux algorithmes. Un algorithme de recherche en avant *frontNavigation* et un algorithme de recherche en arrière *BackNavigation*. Dans la suite de cette thèse, nous appelons les éléments produisant un élément donné des *ancêtres* et les éléments produits à partir d'un élément donné des *descendants*. Ces algorithmes conservent tout le long de la transformation les éléments visités pour vérifier s'ils appartiennent vraiment à l'ensemble des *ancêtres* ou des *descendants* de l'élément cherché. Il est à noter que ces algorithmes ne sont pas exclusifs à la recherche d'éléments dans les transformations localisées et qu'ils peuvent être utilisés pour des chaînes de transformations classique. Le premier algorithme *frontNavigation* est présenté par l'algorithme 1 et se charge donc de récupérer les descendants d'un élément donné.

Algorithme 1 Algorithme de recherche de *descendants* : *frontNavigation*

```
Require: elt : EObject, start : LocalModel, stop : LocalModel
 1: searched ← {start.srcTrace.getElementRef(elt)}
 2: children ← {}
 3: curTrace = start.srcTrace
 4: while stop ∉ curTrace.srcModels do
 5:    for each e ∈ searched do
 6:       children+ = curTrace.getElementRef(e).srcLink.destElements

 7:       searched ← searched ∪ children
 8:    end for
 9:    curTrace ← curTrace.destModels.srcTrace
10: end while
11: finalElts ← {}
12: for each e ∈ searched do
13:    if stop.isIn(e.getEObject()) then
14:       finalElts ← finalElts ∪ e.getEObject()
15:    end if
16: end for
17: return finalElts
```

Au lieu de simplement suivre les différents liens de traçabilité produits pour une chaîne de transformations, l'algorithme travaille en deux phases. Premièrement, il suit les liens de trace tissés tout en conservant les éléments visités à un moment donné (ligne 4 à 11). Ensuite, il effectue une recherche pour vérifier quels sont les éléments effectivement présents dans le modèle d'arrivée de la recherche (ligne 12 à 17). L'algorithme travaille avec trois paramètres venant de la trace globale :

- elt de type EObject, correspondant à l'élément dont on veut récupérer les descendants,
- start de type LocalModel, correspondant au modèle à partir duquel la recherche doit avoir lieux,
- stop de type LocalModel, correspondant au modèle où la recherche doit s'arrêter.

Les lignes 1 à 3 servent à l'initialisation des variables utilisées par l'algorithme. La première variable searched correspond à l'ensemble des éléments cherchés à un moment donné et correspond à l'ensemble des éléments visités tout au long de la recherche. La seconde variable children sert à conserver le résultat de la recherche des successeurs d'un élément pour une transformation. La troisième variable initialisée, récupère la trace à partir de laquelle l'algorithme va commencer à travailler, c'est-à-dire, la trace existante entre le modèle start et start + 1. La ligne 4 indique que la recherche s'interrompt lorsque le modèle final est dépassé. Ici, cela signifie, lorsque le modèle où l'on doit s'arrêter stop est la source de la trace courante, le modèle d'arrêt est dépassé. Ensuite, pour tous les éléments contenus dans searched (ligne 5), les éléments créés par la transformation sont récupérés grâce à la trace (ligne 6) et sont ensuite ajoutés à l'ensemble des éléments à chercher (ligne 7). Une fois la recherche terminée pour l'ensemble des éléments, on passe à la trace suivante (ligne 9).

Une fois que le modèle stop est dépassé, les descendants potentiels se trouvent dans searched. Il faut maintenant vérifier que ces éléments appartiennent bien au modèle de sortie. Pour chacun d'eux (ligne 12), si l'élément appartient bien au modèle terminal stop (ligne 13), il est conservé dans une variable finale finaleElts (ligne 14). Sinon, la recherche continue sur les autres éléments de searched et l'élément e n'est pas considéré comme appartenant à l'ensemble des descendants de elt. Une fois la vérification terminée, la variable est retournée par l'algorithme (ligne 17).

Cet algorithme de recherche des descendants d'un élément à partir d'un modèle de départ vers un modèle d'arrivée est aussi décliné pour une recherche d'ancêtres. L'algorithme 2 présente les opérations effectuées. Ce sont les mêmes que pour l'algorithme de recherche de descendants, à la différence du sens navigué. Mis à part le sens de navigation, les paramètres d'entrée de l'algorithme et sa structure est identique à l'algorithme de recherche des descendants.

Algorithme 2 Algorithme de recherche d'*ancêtres* : *backNavigation*

```
Require: elt : EObject, start : LocalModel, stop : LocalModel
 1: searched ← {start.destTrace.getElementRef(elt)}
 2: children ← {}
 3: curTrace = start.destTrace
 4: while stop ∉ curTrace.destModels do
 5:   for each e ∈ searched do
 6:     children+ = curTrace.getElementRef(e).destLink.srcElements
 7:     searched ← searched ∪ children
 8:   end for
 9:   curTrace ← curModels.srcModels.destTrace
10: end while
11: finalElts ← {}
12: for each e ∈ searched do
13:   if stop.isIn(e.getEObject()) then
14:     finalElts ← finalElts ∪ e.getEObject()
15:   end if
16: end for
17: return finalElts
```

3.5.5 *Construction de la trace réduite*

La construction de la trace réduite peut être débutée lorsqu'une trace globale existe, donc lorsqu'une chaîne de transformations a fini d'être exécutée. La construction de cette trace repose sur les algorithmes de recherche de descendants et d'ancêtres. Pour construire la trace réduite sur la totalité d'une chaîne de transformations[31], il faut chercher les descendants (ou les ancêtres) de tous les éléments contenu dans le premier modèle (ou le dernier) en prenant le dernier modèle (ou le premier) comme modèle final de la recherche. Comme nous l'avons déjà évoqué, le méta-modèle de trace locale peut être utilisée pour conserver la trace réduite. Lorsque les descendants (ou ancêtres) d'un élément sont trouvés, un lien *Link* de trace est tissé entre l'élément sur lequel la recherche a été lancée et les éléments retrouvés.

31. Pour une chaîne de n transformations, cela revient à construire la trace réduite $(0, n-1)$.

3.6 CONCLUSIONS

Dans ce chapitre, nous avons présenté notre mécanisme de traçabilité reposant sur deux modèles de trace, un modèle de trace locale et de trace globale. Le modèle de trace locale permet de conserver les liens de traçabiltié tissés lors d'une transformation de modèle alors que le modèle de trace globale permet de naviguer entre les différents modèles et modèles de trace produits dans une transformation de modèle. Le mécanisme de trace que nous proposons a été défini indépendemment de tout langage de transformation et peut être utilisé quel que soit le langage de transformation utilisé. Afin de réduire les temps de calculs et la taille des traces produites, nous avons proposé la construction et l'utilisation d'une trace réduite. Cette trace permet de masquer une partie de la chaîne de transformations lorsque les modèles et les traces intermédiaires produites par ce bout de chaîne ne sont pas nécessaire.

Finalement, nous avons vu quelles sont les problématiques engendrées par les transformations localisées et, plus généralement, par les transformations *sur place* que nous avons résolues en générant, dans un premier temps, la trace de transformation de modèle asymétriquement. Nous avons ensuite présenté les algorithmes de recherche de descendants et d'ancêtres adaptés aux chaînes de transformations localisées.

La trace locale et globale, générées pour elles même, ne sont pas particulièrement utiles. Dans une dynamique de compréhension générale de la transformation et de la chaîne de transformations, ces traces peuvent servir à conserver un lien sur la totalité d'une chaîne de transformations. Comme nous l'avons évoqué dans l'introduction, il faut pouvoir fournir une liaison entre l'exécution de l'application et les modèles de conception pour pouvoir fournir un support à la correction et à l'amélioration de performances des modèles de conception en fonction d'une exécution. La trace de transformation de modèles considérée dans les chaînes de transformations apporte un premier élément de réponse, car elle permet de lier les différents éléments produits et consommés par une chaîne de transformations, des modèles de conception jusqu'au dernier modèle produit.

Dans le prochain chapitre, nous soutenons que la trace peut être utilisée comme base pour le tissage de ce lien entre exécution et modèle de conception. Nous montrons comment ce lien peut être automatiquement exploité avec les spécifications de l'architecture sur laquelle

s'exécute l'application pour améliorer les performances de l'application générée, directement à partir de ses modèles de haut niveau.

Troisième partie

DE L'ÉXECUTION AUX MODÈLES

4

CORRECTION ET OPTIMISATION DE MODÈLES

Comme nous l'avons vu au chapitre 1, modéliser un système logiciel permet de travailler à un niveau d'abstraction plus élevé et d'éviter de se perdre avec les détails liés à l'implémentation et à la plate-forme d'exécution. Cependant, même si le passage à un niveau d'abstraction supérieur facilite grandement la conception de l'application, cela n'assure pas que le comportement ou les performances du programme généré à partir de ses modèles seront ceux effectivement attendus.

Dans un tel contexte, puisque la chaîne de compilation est considérée comme digne de confiance, le développeur doit chercher l'erreur soit dans le code source de son application, soit dans son modèle de conception. En effet, comme pour un programme écrit dans un langage de programmation classique, un modèle peut comporter des erreurs introduites par le développeur et nécessite d'être corrigé.

Le comportement général d'un programme peut poser des problèmes de différentes manières. Il peut ne pas effectuer les tâches pour lesquelles il est créé, ou posséder des performances médiocres. Les performances de l'application sont tout aussi importantes que son comportement général, surtout lorsque les applications modélisées sont déployées sur des systèmes devant assurer une réponse rapide et efficace.

Dans ce chapitre, nous nous intéressons aux deux aspects présentés précédemment, à savoir, la correction d'erreurs dans les modèles et l'amélioration des performances d'une application. En section 4.1, nous présentons les travaux existants relatif, à la correction des erreurs présentes dans les modèles de conception, puis nous montrons les techniques et principes utilisés pour l'amélioration des performances d'un modèle en section 4.2 avant de conclure en section 4.3.

4.1 CORRECTION DE MODÈLES

Il est à noter que dans cette section, nous adressons uniquement la correction comportementale de modèles, nous ne traitons pas de la correction structurelle de modèles.

Pour fournir un moyen efficace de corriger des modèles par rapport à leurs comportements, Haberl et al. ont proposé dans [59] une approche à base de simulateur. Le code de l'application est généré à partir des modèles de conception grâce à une transformation de modèle avant d'être exécuté. La trace d'exécution est ensuite transformée en un modèle de simulation qui permet de regarder graphiquement le

comportement de l'application. Néanmoins, les modèles de conception proposés dans ce travail sont des modèles d'assez bas niveau et possèdent une structure très similaire au code généré. De plus, il est difficile d'adapter cette technique à une chaîne de compilation, car les informations obtenues pour la simulation sont trop proches du modèle ayant servi à générer le code. Ainsi, si ces informations, sont retournées sur les modèles de conception, elles ne sont plus toutes pertinentes puisqu'elles sont relatives aux derniers modèles proches du code. Déterminer alors les informations qui méritent d'être retournées sur les modèles de conception n'est pas une chose aisée.

Un autre travail [35] propose un *debugger* pour UML. Les modèles UML construits sont transformés vers du JAVA et les résultats de l'exécution sont visualisables par animation de diagramme sur les modèles de conception. Néanmoins, le *debugger* proposé ne s'occupe que de la partie comportementale d'UML et utilise JAVA pour fournir un support d'exécution pour les modèles UML. Afin de fournir le retour sur les modèles de conception de l'exécution proposée, les auteurs utilisent la traçabilité modèle vers code, donc, la traçabilité entre UML et le code JAVA généré. Pour ce travail aussi, les auteurs ne considèrent pas de chaînes de transformations. Comme extensions à leur travail, les auteurs ont identifié deux *challenges* importants. Le premier est de fournir un *framework* de *debugging* assez générique pour pouvoir être réutilisé pour n'importe quel langage de modélisation et quel que soit le langage de programmation visé par la génération de code. Le second est de fournir un moyen efficace pour traiter avec le passage à l'échelle. En effet, la visualisation tel qu'ils le proposent peut vite devenir chaotique lorsque des gros modèles sont utilisés.

De manière générale, cette approche de récupération d'informations d'exécution, puis de retour sur les modèles de conception, présentée dans les travaux énoncés, a aussi été abordée en [62, 63]. Les modèles de conception sont transformés vers des modèles de réseaux de pétri qui sont, par la suite, simulés. Les informations récupérées par cette simulation sont ensuite reportées sur les modèles de conception. Le retour des informations obtenues à la simulation du modèle de réseaux de pétri est ici assez simple compte tenu du fait qu'une seule transformation est utilisée et que l'exécution proposée est effectuée par transformation de modèles implémentant la sémantique du modèle mathématique des réseaux de pétri. Néanmoins, lorsque des modèles traitent avec des applications pour des architectures matérielles, la traduction vers des réseaux de pétri peut être vraiment complexe et ne pas refléter entièrement le comportement réel de l'application lors de l'exécution.

De manière assez semblable, le travail présenté dans [113] propose de transformer un modèle UML vers un modèle ALLOY [66], pour permettre aux concepteurs d'effectuer plusieurs vérifications de propriétés sur leurs modèles. Afin de retourner les informations sur les modèles, ce sont, encore une fois, les traces obtenues à l'exécution de la transformation de modèles qui sont utilisées. La transformation de modèles étant écrite en QVT, ce sont les traces de modèles de QVT qui sont utilisées pour remonter les informations sur les modèles de conception. Cependant, la traduction vers *Alloy* n'est utilisée ici que pour vérifier certaines propriétés sur l'application modélisée et, comme pour le travail précédent, cela ne reflète pas son comportement général.

Les travaux présentés dans cette section nous permettent de voir que deux approches générales se distinguent. La première repose sur la

trace d'exécution produite lors de l'exécution de l'application générée et la trace de modèles vers texte produite lors de la génération de l'application. Cette dernière est utilisée pour mettre en relation le modèle et le code qu'il a généré. Ainsi, chaque élément de la trace d'exécution produite peut être lié à l'élément de modèle correspondant. Cependant, ces travaux considèrent des modèles dont la structure est très proche du code généré et ne traitent pas avec des chaînes de compilation IDM. La seconde approche, qui se dégage des travaux présentés, propose de passer par une exécution à base de réseaux de pétri ou de modèles Alloy dans le but de vérifier certaines propriétés sur les modèles de conception. Cependant, l'exécution à base de modèles proposée peut ne pas refléter le comportement global de l'application à l'exécution.

4.2 amélioration des performances d'un système

En IDM, le processus d'analyse des performances d'une application porte le nom de SPE [1]. Le processus SPE utilise plusieurs outils d'évaluation de performances en fonction de l'état de l'application et du volume de données de performances disponibles. Actuellement, le SPE est une approche relativement mature et, est normalement associée à la prédiction de performance de l'application pendant les premières étapes de conception.

1. Software Performance Engineering

Plusieurs approches de modélisation ont été proposées dans la littérature pour l'optimisation de modèles, en comptant les approches par simulation et les approches à bases de modèles [2]. Nous allons présenter plusieurs de ces travaux selon deux axes : les travaux d'optimisation reposant sur des approches statiques et ceux reposant sur des approches dynamiques.

2. Approches reposant sur UML en particulier.

4.2.1 *Optimisation de modèles par approche statique*

Le MDSPE [3], une extension du SPE, est proposé en [122]. Ce processus propose d'annoter les modèles UML avec des informations dédiées aux performances. Le MDSPE est utilisé pour la prédiction des performances des systèmes. Il consiste en la production de modèles de performance à partir des modèles annoté avec le profil SPT [4] [103]. Concrètement, le processus MDSPE se déroule en deux étapes importantes : une étape d'*annotation des performances* et une étape d'*analyse des performances*. La première étape, effectuée par le concepteur, consiste à annoter manuellement le modèle avec le profil SPT pour indiquer les spécifications et contraintes matérielles directement sur le modèle. La seconde étape est implémentée par un analyseur de performance qui calcule, en fonction de différentes métriques et des informations ajoutées au modèle, les performances du logiciel.

3. Model Driven SPE

4. Schedulability, Performance and Time

Dans [9], les auteurs proposent une autre approche reposant sur le profil SPT. Une fois le modèle de conception annoté, celui-ci est transformé en un modèle de simulation où chaque paramètre de performance annoté est transformé en paramètre spécifique du modèle de simulation. Le modèle de simulation est ensuite exécuté et les résultats sont reportés sur le modèle de conception (modèle UML).

Un autre outil, ArgoSPE proposé dans [57], reposant aussi sur un modèle UML profilé SPT permet d'évaluer les performances de logiciels dans les premières étapes du développement. Du point de vue du concepteur, ArgoSPE est dirigé par un ensemble de « requêtes de

performances » que l'outil peut exécuter pour obtenir une analyse quantitative du système modélisé. Pour ArgoSPE, une requête de performance est une procédure par laquelle le modèle UML est analysé pour obtenir automatiquement une estimation de performance. Chaque requête est associée à un diagramme UML et les requêtes qu'il est possible de demander sont différentes en fonction du diagramme. Les diagrammes supportés sont ceux de collaboration, de déploiement et les diagrammes d'état transitions. De manière générale, le résultat de la requête est calculé grâce à une interprétation du modèle UML, sauf pour le diagramme d'état transition ou ArgoSPE passe par un réseau de pétri afin de délivrer le résultat de la requête. Le résultat obtenu est ensuite retourné au concepteur sur son modèle par annotations grâce à la trace de transformations.

Les travaux précédents s'appuient tous sur SPT et sur UML uniquement. Cependant, le profil MARTE [102] propose aussi des concepts pour l'analyse de performances : le package PAM[5] qui permet au concepteur de définir les spécifications de la plate-forme d'exécution sur les modèles de conception. Cependant, comme pour UML SPT, les spécifications de la plate-forme sont modélisées sur le modèle de conception et suppose une connaissance approfondie de la plate-forme d'exécution de la part du concepteur de modèle.

5. Performance Analysis Modeling

Beaucoup d'autres travaux existent sur le SPE et UML, mais proposent des solutions assez semblables aux travaux déjà présentés tel que le montre l'enquête réalisé par Balsamo et al. dans [8]. Cependant, dans [51, 52] est présenté une chaîne de transformations prenant un modèle de processus en entrée et produisant un modèle utilisé par un outil d'analyse de performances. Les résultats obtenus sont ensuite reportés sur un des modèles de la chaîne grâce aux traces produites lors de l'exécution de la chaîne de transformations. Ce travail s'applique uniquement dans le domaine des modèles de processus et utilise un outil d'analyse de performances travaillant par simulation.

Les travaux présentés dans cette sous-section proposent tous d'utiliser un profil UML dédié aux performances pour permettre au concepteur d'annoter le modèle avec les spécifications et contraintes du matériel sur lequel l'application sera exécutée. Une fois que le concepteur a annoté les modèles de conception, ceux-ci sont transformés pour permettre à des outils dédiés d'estimer les performances de l'application modélisée. Ces travaux supposent tous une double compétence de la part des concepteurs de modèles. En effet, ils doivent non seulement concevoir le système, mais aussi connaître et être capable d'annoter correctement les modèles avec les spécifications de la plate-forme d'exécution. Ces approches introduisent donc un effort de conception important en plus de la conception du système.

4.2.2 *Optimisation de modèles par approche dynamique*

Dans le cadre de la programmation classique, l'amélioration des performances du logiciel grâce aux retours obtenus à l'exécution du programme porte le nom de *profiling*. À ce jour, nous n'avons pas connaissance d'outils proposant d'utiliser l'exécution du code généré pour récupérer les informations de *profiling* afin d'améliorer les modèles de haut niveau.

Néanmoins, le travail présenté dans [81] vise au *refactoring* des modèles de conception pour en améliorer les performances. Cette approche

repose sur l'optimisation des tâches parallèles et séquentielles sur un accélérateur FPGA. Dans ce cadre, avant la génération du code VHDL, l'application au niveau RTL [6] est analysée par un système d'optimisation qui exploite l'allocation des ressources FPGA. Ensuite, les modèles d'entrée sont modifiés en fonction des résultats de l'outil d'optimisation de performance. Cette façon de procéder possède l'avantage de ne pas imposer au concepteur de solides connaissances sur la plate-forme visée. Cependant, elle nécessite une bonne compréhension des retours proposés par les outils de *profiling* et de la chaîne de compilation IDM utilisée pour pouvoir modifier effectivement le modèle généré. En effet, déterminer au premier regard quel est l'élément à modifier en fonction de résultats obtenus à partir d'un niveau intermédiaire dans la chaîne de compilation n'est pas facile. Pour effectuer cette approche automatiquement, il faut pouvoir déterminer facilement les liaisons entre les différents éléments dans une chaîne de compilation.

6. Register Transfer Level

4.3 CONCLUSIONS

Dans ce chapitre, nous avons vu différents travaux traitant de la correction des modèles de conception ainsi que de l'amélioration de leurs performances. Comme nous avons pu le voir, la correction de modèles est une thématique assez neuve. Seuls quelques travaux adressent le problème de la correction de modèles pour les diagrammes de comportement présents dans UML. De manière générale les approches abordées proposent de transformer les modèles de conception vers :

- d'autres modèles afin de fournir une simulation à base de modèles,
- un code source pour une exécution de l'application modélisée.

Dans les deux cas, une trace d'exécution est produite contenant des informations de l'« exécution » proposée. Ces informations sont ensuite remontées vers les modèles de conception grâce aux traces de transformation de modèles à modèles ou les traces de modèles vers texte générées lors de la transformation des modèles de conception. Ces approches pourraient être une solution pour la correction de modèles de conception dans GASPARD2, mais aucune d'entre elles ne traitent avec des chaînes de compilation IDM et toutes ces approches reposent sur l'utilisation d'un langage en particulier pour produire l'exécution (JAVA, ALLOY ou les réseaux de pétri). De plus, selon [35], deux problèmes sont encore à résoudre pour les mécanismes de corrections de modèles : il faut qu'ils puissent être assez génériques pour pouvoir être adaptés à n'importe quel langage de modélisation et il faut qu'ils proposent une solution pour effectuer un tri dans les informations récupérées lors d'un passage à l'échelle.

Dans ce chapitre, nous avons aussi présenté des travaux relatifs à l'amélioration des performances des modèles de conception. La plupart des publications que nous avons référencées tentent de placer l'optimisation des performances au plus tôt dans le processus de développement. Ainsi, à partir des modèles de conception, une analyse statique des modèles propose une estimation des performances du système modélisé. En revanche, tous ces travaux supposent une très bonne connaissance des spécifications de l'architecture d'exécution de la part des concepteurs et de l'interprétation des résultats des outils d'estimation de performances. De plus, la transformation du modèle de conception vers un modèle de performance dédié impose un effort de développement supplémentaire. En effet, dans le cadre d'une chaîne

de compilation IDM, cela reviendrait à construire une chaîne différente de la chaîne de compilation principale pour transformer le modèle de conception vers le modèle de performances qui sera analysé par un outil dédié. Une solution pour éviter cet effort de développement et cette double compétence de la part du concepteur serait de profiter de la chaîne de compilation IDM existante pour générer le code de l'application et d'utiliser des outils de *profiling* pour récupérer des mesures précises des performances de l'application. De cette façon, les données ne seraient plus estimées en fonction d'annotations laissées sur modèles, mais directement mesurées par des outils adéquats à l'exécution de l'application.

5 OPTIMISATION DE MODÈLES

Dans une chaîne de compilation IDM, les modèles spécifiés par le concepteur sont successivement raffinés avant d'être transformés en code source d'un programme. Dans certains cas, lorsque l'application modélisée est assez précise, la génération peut être entièrement automatique et le code source généré ne requiert aucune intervention humaine pour pouvoir fonctionner. Cependant, la génération automatique et complète du programme n'assure pas que son fonctionnement sera pleinement performant en terme, par exemple, de temps ou encore de consommation mémoire, même si des transformations d'optimisation sont proposées dans la chaîne de compilation IDM. Ainsi, un programme généré depuis ces modèles de spécification peut ne pas remplir les attentes du concepteur en terme de performance ; des modifications doivent donc être effectuées pour optimiser le modèle.

Classiquement, c'est le code source généré du programme qui est manuellement modifié. Cependant, cette façon de procéder mène à une désynchronisation entre les modèles de conception et le code source. Il devient donc difficile de maintenir, modifier ou faire évoluer les modèles d'entrée. De plus, une nouvelle génération à partir du modèle initial peut entraîner la suppression des modifications effectuées sur le code. Une des solutions envisageable serait donc de profiter d'un mécanisme de synchronisation entre le code source et le modèle. Ainsi, toutes modifications effectuées sur le code source pourrait être directement impactée sur les modèles de spécifications.

Néanmoins, la mise en oeuvre d'un tel mécanisme est particulièrement complexe. En effet, une chaîne de compilation peut traiter avec beaucoup de transformations de modèles qui sont rarement bidirectionnelles et les modifications apportées au code source peuvent être diverses. Dans le cas d'un ajout par exemple, il faudrait pouvoir interpréter complètement les lignes de programmation ajoutées pour pouvoir générer automatiquement les éléments correspondant dans les modèles de spécification. Par ailleurs, il faudrait analyser l'impact d'une telle modification dans les modèles de spécifications sur le code source généré. De plus, la modification du code source suppose une très bonne connaissance, à la fois, du langage de programmation utilisé et du code généré de la part du concepteur.

Une autre solution possible pour garder cette synchronisation entre le code source et le modèle consiste à générer à nouveau le code source à partir des modèles de spécification préalablement modifiés. En revanche, il faut pouvoir identifier les éléments des modèles de

spécification qui posent les problèmes de performances à l'exécution. De plus, même en ayant identifié les éléments de modèles posant problème, les modifier efficacement n'est pas une activité aisée sachant que pour atteindre de meilleures performances, des détails techniques sur la plate-forme d'exécution doivent être pris en compte.

Dans ce chapitre, nous présentons une approche d'optimisation de modèles basée sur la traçabilité pour remonter automatiquement sur les modèles de spécification, des détails et mesures de performance obtenus durant l'exécution du programme généré. Les mesures de performances sont récupérées à partir d'un outil de *profiling* dédié et annotent les éléments adéquats des modèles de spécification. En plus de ces informations « brutes » de *profiling*, des « conseils » sont remontés sur les modèles de spécification. Ces conseils proposent notamment d'affiner rapidement et efficacement la modélisation. Une fois que les modèles sont modifiés, le code du programme est à nouveau généré, assurant la cohérence entre les deux artefacts.

Dans un premier temps, nous abordons en section 5.1 la structure générale de notre approche. Nous présentons ensuite en section 5.2 le mécanisme que nous utilisons pour conserver un lien entre l'exécution et les modèles de conception. Nous nous concentrons ensuite sur la construction des conseils en section 5.3 et la remontée des mesures de performance obtenues en section 5.4.

5.1 VUE GÉNÉRALE DU PROCÉDÉ D'OPTIMISATION

L'approche que nous présentons dans cette section est illustrée figure 51. Elle se situe dans le contexte d'une chaîne de compilation IDM et se compose de 3 grandes parties (détaillées en 7 étapes) :

1. la génération de l'application et sa compilation (étapes 1 et 2),
2. l'exécution de l'application sur la plate-forme et la production des informations de *profiling* (étapes 3 et 4),
3. l'analyse des résultats du *profiling* et la remontée des informations sur le modèle de haut niveau (étape 5 à 7).

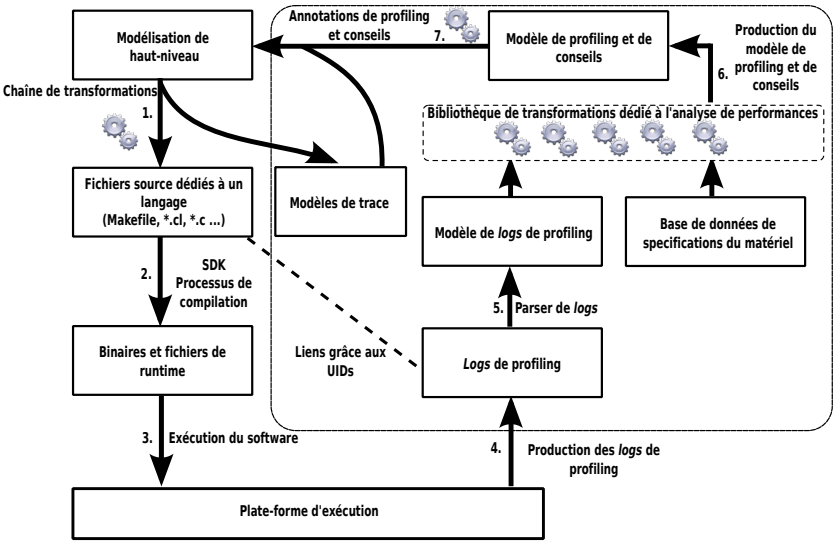


FIGURE 51: Processus d'optimisation de modèles

Après le design des modèles de haut-niveau, les fichiers source de l'application sont générés grâce à une chaîne de transformations (étape 1). Durant cette génération, des modèles de traces conservant le lien entre les différents modèles de la chaîne sont produits (plusieurs modèles de trace locale ainsi qu'un modèle de trace globale). Les sources du programme sont ensuite compilées en utilisant un SDK (*Software Development Kit*) dédié au langage de programmation utilisé (étape 2). Une fois les binaires produits, l'application est exécutée (étape 3). Au cours de l'exécution, un outil de *profiling* dédié génère des *logs* [1] de *profiling* contenant des mesures relatives aux performances de l'application, comme un temps d'exécution ou la consommation mémoire du programme (étape 4). Les informations de *profiling* produites contiennent également des références aux éléments du dernier modèle de la chaîne de transformations (lien en pointillé). Ce lien est assuré grâce à un mécanisme basé sur des UIDs [2]. Les informations de *profiling* sont parsées pour produire un modèle de profiling (étape 5) qui est utilisé comme entrée d'un système expert (noté « Bibliothèque de transformation dédiées à l'analyse de performances » sur la figure 51). Le système expert utilise, en complément du modèle de *logs* de *profiling*, une base de données contenant les spécifications et contraintes matérielles de la plate-forme d'exécution. En utilisant à la fois les valeurs mesurées à l'exécution et les spécifications de la plate-forme d'exécution, le système expert génère un modèle relayant les informations issues des *logs* de *profiling* et conseillant le concepteur de modèle sur comment modifier les modèles de haut niveau pour atteindre de meilleurs performances (étape 6). Finalement, en utilisant les modèles de traces produits par l'étape 1, le modèle de *conseils* généré est remonté sur le modèle de haut niveau où les conseils et les informations de *profiling* annotent les éléments qui devraient être modifiés (étape 7). Une fois les annotations ajoutées au modèle de conception, le concepteur de modèles peut facilement effectuer une ou plusieurs des modifications suggérées. Les modifications effectuées, la chaîne de transformations est jouée à nouveau et le programme exécuté de nouveau pour vérifier que les performances ont bien été améliorées.

1. Dans cette thèse, nous utilisons le terme de *log* pour parler de la trace d'exécution d'un programme.

2. Pour UNIQUE IDENTIFIERS, le mécanisme est présenté en section 5.2.

5.2 GARDER LE LIEN ENTRE EXÉCUTION ET MODÈLES

Pour conserver le lien entre modèles et exécution, lors de la génération du code source, nous proposons l'utilisation d'*Unique IDentifiers* (UIDs). Les UIDs permettent de faire référence à un élément de modèle via une chaîne de caractère, représentant un identifiant unique. Si ces UIDs apparaissent dans chaque entrée d'un *log* généré lors de l'exécution d'une application, il est donc possible de savoir à quel élément doivent être liées les informations contenues par chaque entrée du *log*. Dans notre approche d'optimisation de modèles, les UIDs présents dans le *log* de *profiling* sont ceux faisant référence aux éléments du dernier modèle avant la génération de code. Ainsi, lors de l'exécution de l'application, les entrées des *logs* de *profiling* produits par l'outil peuvent garder une référence aux éléments des derniers modèles de la chaîne. En utilisant ce mécanisme à base d'UIDs, il est donc possible de reconnecter une représentation de l'exécution du software aux éléments du dernier modèle avant la génération de code.

Afin d'ajouter ces UIDs dans le *log* de *profiling*, il faut analyser l'outil utilisé pour déterminer le lien qui existe entre le code source généré

et les informations obtenues dans le *log*. Généralement, le nom des fonctions est utilisé par les outils de *profiling* pour indiquer au développeur à quelles parties du code source font référence les informations obtenues. Dans ce cas, pour faire apparaître l'UID dans le *log*, il suffit de modifier la génération de code pour qu'elle concatène l'UID de l'élément au nom de la fonction générée.

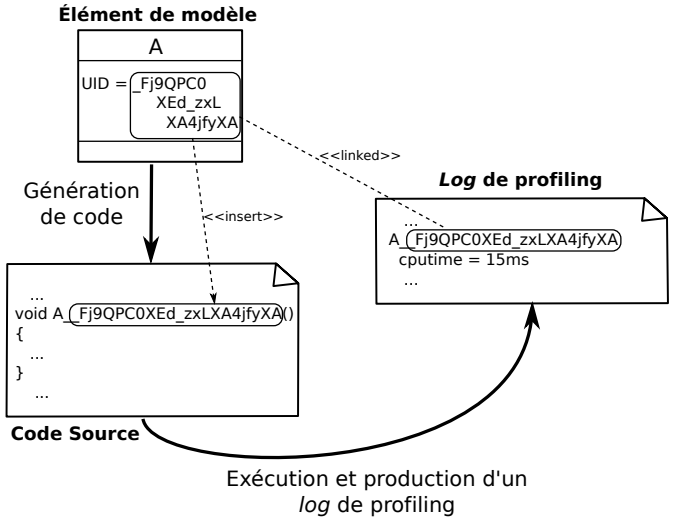


FIGURE 52: Lien entre modèle et exécution

Cette façon de procéder est illustrée en figure 52. Il y est présenté un élément de modèle nommé *A* et possédant la chaîne *_Fj9QPCoXEd_zxLXA4jfyXA* comme UID. Lors de l'étape de génération de code, cet UID est concaténé au nom de la fonction généré pour l'élément *A*. Une fois le code source de l'application compilé, celle-ci est exécutée. Le *log* de *profiling* généré à l'exécution contient une entrée portant le nom de la fonction auquel est concaténé l'UID de l'élément. Cette entrée précise que la fonction a pris 15ms pour être exécutée sur le *CPU*. Grâce à l'UID contenu dans le nom de la fonction, il nous est donc possible de déduire qu'à un moment donné, l'élément *A* à pris 15ms pour s'exécuter sur le *CPU*. L'utilisation des UIDs a donc permis de reconnecter les informations obtenues à l'exécution de l'application avec le monde des modèles. Néanmoins, pour le moment, les modèles avec lesquels l'exécution est reconnectée sont les derniers modèles générés par la chaîne de compilation.

Il faut pouvoir faire le lien entre ces éléments et ceux des modèles de conception. Il est donc important de garder un lien d'un bout à l'autre de la chaîne de compilation. Les traces locales générées et la trace globale sont utilisées pour connaître les *ancêtres*[3] d'un élément du dernier modèle de la chaîne. Les seuls éléments cherchés sont ceux des modèles de conception, nous utilisons une trace réduite qui permet à partir des éléments du dernier modèle généré d'avoir accès à ceux du modèle de conception sans passer par les modèles intermédiaires. De cette façon, les algorithmes et les recherches d'éléments sont simplifiés.

3. *C.f*, Chapitre 3, page 57.

5.3 LA CONSTRUCTION DES « CONSEILS »

Une fois que les deux premières parties du procédé sont complétées (étapes 1 à 4), les *logs* produits par l'exécution sont analysés et un modèle de *profiling* et de « conseils » est construit. Pour mémoire, ceci correspond aux étapes 5 et 6 de notre approche (*c.f*, figure 51). Nous allons montrer dans un premier temps comment les *logs* de *profiling* et la base de connaissance sur la plate-forme d'exécution sont modélisés. Par la suite, nous présenterons le système expert qui analyse et traite les données qui lui sont passées en paramètres pour produire le modèle de *logs* de *profiling* et de conseils.

5.3.1 *Parsing des logs de profiling*

Dans un premier temps, les informations contenues dans le *log* de *profiling* sont parsées (étape 4) pour construire un modèle de *log* de *profiling* conforme au méta-modèle montré en figure 53. Ce méta-modèle propose de stocker de manière simple toutes les informations qui peuvent être trouvées dans un *log* de *profiling*. Ainsi, il est aisé de construire un fichier texte XMI représentant un modèle conforme à ce méta-modèle quelque soit la solution employée pour effectuer le parsing du *log* [4].

4. Pour les expérimentations, la construction du modèle a été effectuée avec de simples scripts BASH

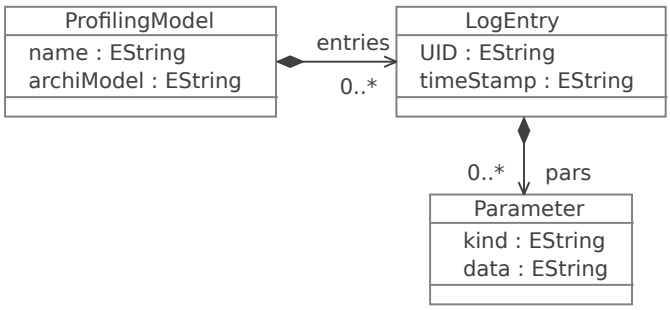


FIGURE 53: Méta-modèle de *log* de profiling

Le méta-modèle de *log* de *profiling* est composé de trois méta-classes. La méta-classe *ProfilingModel* représente la racine du méta-modèle. L'attribut *archiModel*, spécifie le modèle d'une architecture matérielle (par exemple, le modèle TESLA T10 pour un matériel GPU) sur lequel le programme généré est exécuté. Un modèle de *profiling* est composé de zéro ou plusieurs *logEntry* représentant une entrée du *log* de *profiling*, c'est-à-dire, un évènement ou une mesure spécifique généré par l'outil de *profiling*. Les entrées du *log* possèdent un *timestamp* permettant d'indiquer l'ordre dans lequel elles sont apparues. De plus, elles gardent l'*UID* correspondant à l'identifiant d'un élément du dernier modèle de la chaîne (comme présenté en section précédente). Chaque entrée peut contenir un certain nombre de paramètres (*Parameter*) représentant une mesure ou une valeur particulière. Ainsi, un *Parameter* possède un type : *kind* (*ex.* : occupation, temps d'exécution ou consommation mémoire) et une valeur : *data*. Ce méta-modèle est suffisamment générique pour pouvoir stocker facilement tout type d'information présente dans un *log* de *profiling*. Une fois les modèles de *logs* de *profiling* obtenus, ils sont analysés par un système expert.

5.3.2 *Le système expert*

Le système expert permet d'analyser les informations obtenues grâce à l'outil de *profiling*. Il repose sur l'analyse des informations obtenues à l'exécution de l'application en utilisant une base de connaissance précisant les spécifications des architectures matérielles sur lesquels les applications sont exécutées.

Les spécifications matérielles

En plus des informations contenues dans les *logs* de profiling, une connaissance sur le matériel est nécessaire pour pouvoir comprendre et se servir au mieux des résultats de l'outil de *profiling*. En effet, les contraintes liées au matérielle peuvent impliquer des pertes de performances lorsque le système n'est pas correctement modélisé. Cependant, ces spécifications ne sont pas spécialement connues des concepteurs de modèles. Nous proposons de modéliser une base de connaissance pour que le système expert puisse en tirer les informations qui lui seront nécessaires. Cette base de connaissance est représentée sous forme d'un méta-modèle qu'il est possible d'étendre pour prendre en compte de nouveaux détails sur l'architecture. Nous avons choisi d'utiliser une représentation sous forme de méta-modèle plutôt qu'une base de données traditionnelle pour pouvoir facilement bénéficier des supports apportés par les langages de transformations de modèles et le framework EMF. Le méta-modèle utilisé pour représenter la base de données est présenté en figure 54.

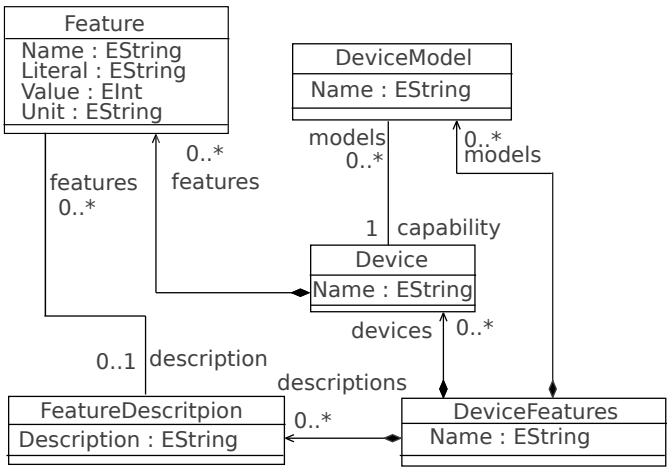


FIGURE 54: Méta-modèle de spécifications de matériel

La méta-classe *DeviceFeatures* représente la racine d'une base de connaissances. Celle-ci peut contenir divers matériels *Device*, divers modèles de matériels *DeviceModels* et des descriptions de propriétés d'un matériel *FeatureDescription*. Les modèles de matériels qui peuvent être modélisés par *DeviceModel* font référence à un modèle de matériel propriétaire particulier (*ex.*, Tesla T10 pour un matériel GPU ou Intel Core Duo *E6850* pour un matériel CPU). Bien évidemment, chaque matériel possède des caractéristiques uniques. Chacun des *Devices* se compose de plusieurs *Features* représentant une caractéristique spéciale du matériel qui peut porter soit une valeur numérique *Value*, soit une valeur textuelle *Literal*. Une unité de mesure *Unit* vient préciser la

valeur indiquée dans le champ *Value* ou *Literal*. Ces caractéristiques du matériel peuvent être associées à une description (*FeatureDescription*) donnant une information textuelle et générale sur la *Feature*.

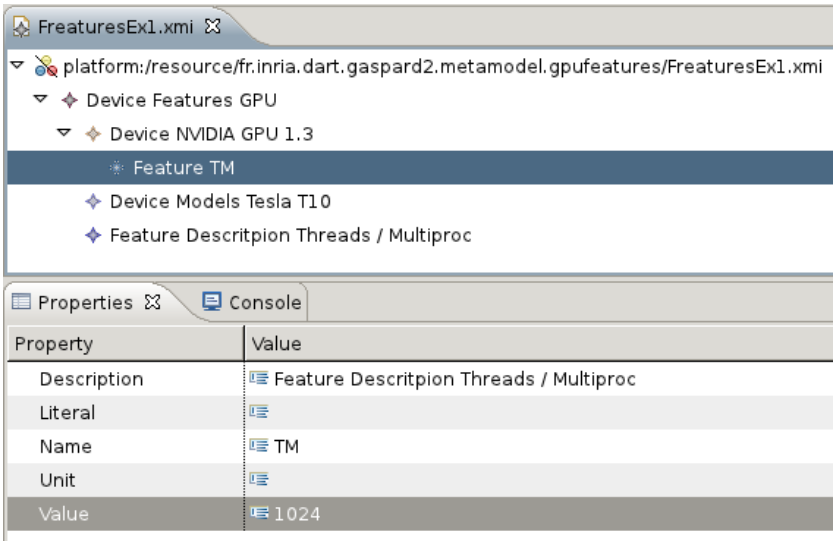


FIGURE 55: Exemple de modèle de spécification de matériel

Comme illustration de la base de connaissance, la figure 55 présente sous forme arborescente un exemple de modèle de base de connaissance pour un matériel GPU en particulier. Le matériel spécifié est un *NVIDIA GPU 1.3* que l'on retrouve sur beaucoup de cartes graphiques. Ce matériel est associé au modèle de GPU *Tesla T10* et possède, sur cet exemple, une caractéristique *TM* représentant le nombre de *threads* qu'il est possible de lancer en même temps par multiprocesseur. Sur cet exemple, il est précisé que le *NVIDIA GPU 1.3* peut lancer jusqu'à 1024 *threads* en même temps.

Le système expert

Le système expert est composé d'une bibliothèque de transformations dédiées, écrites par un expert du matériel utilisé pour l'exécution de l'application, prenant en compte à la fois des informations obtenues à partir de deux types d'informations : les *logs* de *profiling* (grâce au modèle de *logs* de *profiling* présenté en section 5.3.1) et les spécifications de la plate-forme sur laquelle l'application est exécutée (grâce à la base de connaissance présentée en section 5.3.2) pour produire un modèle d'*Advice*. Le premier type d'informations donne des données factuelles sur l'exécution alors que le second type d'informations renseigne sur les caractéristiques de la plate-forme d'exécution. En combinant ces deux sources, il est possible, dans certains cas, de déduire les modifications à apporter au modèle de conception pour améliorer un critère de performance.

Le retour des informations de *profiling* brutes sur les modèles constitue une aide précieuse pour le concepteur de modèles mais peut être parfois complexe à lire. Les suggestions alors proposées par le systèmes expert permettent au concepteur de rapidement voir comment les modèles peuvent être modifiés.

Pour ce faire, le système expert embarque un certain nombre de formules et d'algorithmes capables d'estimer les valeurs à modifier dans les modèles de spécifications pour améliorer les performances de l'application générée. Ces algorithmes et formules sont représentés sous la forme de transformation de modèles prenant en entrée le modèle de *log* de *profiling* généré à l'exécution et produisant un modèle contenant à la fois les informations récupérées par l'outil de *profiling* et un conseil pour modifier les modèles de conception. Une fois encore, c'est la simplicité et la commodité de manipulation des modèles d'entrée qui nous a orienté vers les langages de transformation de modèles. Évidemment, tout langage de programmation capable de naviguer à travers des modèles et capable d'en produire peut être candidat à l'écriture des algorithmes. Les informations et suggestions produites par les algorithmes sont présentées dans un modèle de *conseils*. Celui-ci est conforme au méta-modèle illustré en figure 56.

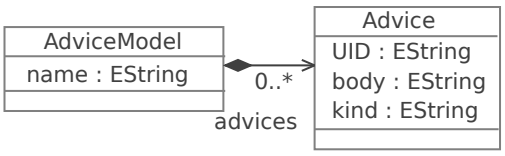


FIGURE 56: Méta-modèle de *logs* de *profiling* et de conseils

Ce méta-modèle contient seulement deux méta-classes : *AdviceModel* et *Advice*. La première fait office de racine pouvant contenir plusieurs *Advices*. Chacun d'eux représente un conseil particulier calculé par le système expert. Chaque *Advice* possède un type *kind* qualifiant le critère de performance qu'il propose d'améliorer (*ex.*, *cputime* ou *occupancy*, cet attribut est équivalent à celui présenté en figure 53) ainsi que le corps du conseil *body* sous forme textuelle. Le corps du conseil possède à la fois les valeurs mesurées par l'outil de *profiling* (issues du modèle de *logs* de *profiling*) et le conseil calculé par le système expert. Finalement, pour continuer à conserver le lien avec les modèles produits par la chaîne de compilation, l'UID de l'entrée du modèle de *logs profiling* utilisé dans les calculs est propagé par l'attribut *UID*.

5.4 REMONTÉE D'INFORMATION

Une fois le modèle de *conseils* généré, les modifications à appliquer aux modèles de conception sont accessibles. Cependant, tant qu'elles ne sont pas raccrochées aux éléments qui doivent être modifiés, elles sont difficilement compréhensibles par le concepteur des modèles. Une ultime étape[5] permet d'annoter les modèles de conception avec les conseils calculés par le système expert.

5. *C.f.* étape 7 sur la figure 51.

Les annotations peuvent être remontées sur le modèle de conception grâce à l'utilisation conjointe du mécanisme d'UIDs que nous avons précédemment présenté et de la trace réduite produite. La figure 57 présente les liaisons induites par l'utilisation de la trace réduite et du mécanisme d'UIDs entre les différents modèles.

Dans cette illustration, la chaîne de transformations composée de n transformations ($n-1$ transformations de modèles et 1 transformation de génération de texte) traduit les modèles de conception m_0 (notés modèles de haut-niveau) vers le modèle m_{n-1} représentant le dernier

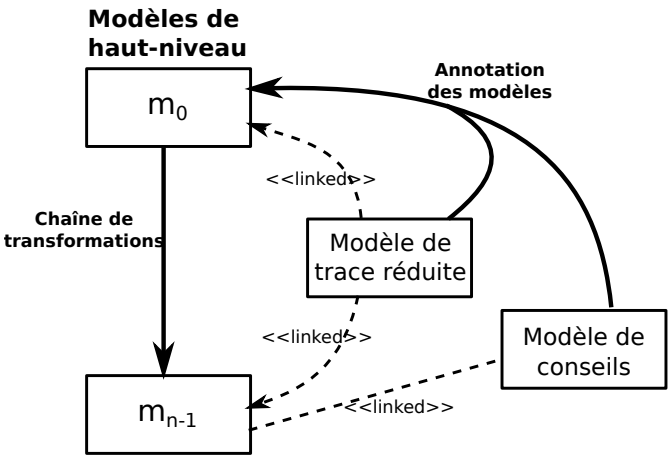


FIGURE 57: Remontée d'informations de *profiling*

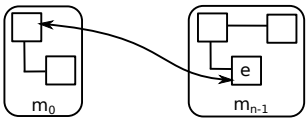
modèle avant la génération de code. Comme nous l'avons vu en section 5.2 un lien est gardé entre les *logs* de *profiling* et le modèle m_{n-1}. Ce lien est propagé (via la propagation de l'UID) jusque dans le modèle de *conseils*. Ainsi, pour un *Advice*[6] du modèle de *conseils* donné, il est possible d'identifier un concept dans m_{n-1}. Dès que cet élément est identifié, il ne reste qu'à parcourir la trace réduite produite pendant l'exécution de la chaîne de transformations pour trouver les éléments qui ont conduit à sa génération dans m_0 (les ancêtres de l'élément dans m_0) et leur attacher le conseil calculé.

6. *C.f.*, méta-modèle figure 56.

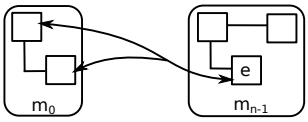
Lors de cette étape de recherche des *ancêtres*[7] dans les modèles de conception m_0 à partir d'un élément e de m_{n-1}, deux cas peuvent intervenir :

7. *C.f.*, chapitre 3.

- un seul ancêtre est trouvé pour e, par navigation des liens de trace du modèle m_{n-1} au modèle m_0



- plusieurs ancêtres sont trouvés pour e, par navigation des liens de trace du modèle m_{n-1} au modèle m_0



L'information est, bien évidemment reportée sur tous les ancêtres identifiés. Cependant, si le nombre d'ancêtre est trop important, remonter l'information sur tous peut rendre la lecture et la compréhension des informations complexes, surtout lorsqu'une modification proposée par le système expert est suggérée pour un seul élément dans le modèle de conception. Pour éviter cette confusion, les différentes algorithmes embarqués dans le système expert peuvent spécifier un ensemble de « types » (types du méta-modèle d'entrée de la chaîne de transformations) sur lesquels le conseil devrait être remonté.

Pour pouvoir annoter les modèles de conception, il faut évidemment que le méta-modèle relatif puisse être capable d'ajouter des annotations. Nous considérons ici UML comme langage de modélisation qui

permet d'ajouter des commentaires[8] à n'importe quel élément et donc d'annoter facilement les différentes instances d'UML. Ainsi, les conseils sont reportés dans les modèles de conception UML sous forme de commentaire. Évidemment, si un autre langage de modélisation avait été utilisé, le concept qui se rapproche le plus de la notion de commentaire en UML aurait été utilisé. Dans l'éventualité où aucun concept de cette sorte n'existe, il aurait dû être introduit dans le méta-modèle pour pouvoir produire et attacher les annotations à des éléments des modèles.

8. Via le concept *Comment*

L'algorithme 3 présente[9] la fonction d'annotation utilisée par l'algorithme de remontée de conseils sur les modèles de conception. Il prend en entrée une trace réduite ($tr : LocalTrace$[10]), un conseil, ($ad : Advice$), qui donne lieu à un commentaire et un ensemble de types sur lesquels raccrocher le commentaire produit ($types : Set(String)$).

9. Le langage utilisé pour décrire l'algorithme est un langage pseudo OCL.

10. Nous rappelons que la trace réduite est représentée grâce à une trace locale (*cf*, chapitre 3).

Algorithme 3 Algorithme d'annotation de modèles UML : *Annotate*

Require: $tr : LocalTrace$, $ad : Advice$, $types : Set(String)$

```
1:  el ← tr.destEltsContainer.elementRefs.selectOne(e | e.uuid = ad.uid)
2:  ancestors ← el.destLink.srcElements
3:  if types = ∅ then
4:    finalElts ← ancestors
5:  else
6:    finalElts ← ∅
7:    for each t ∈ types do
8:      finalElts += ancestors.select(e | e.oclIsKindOf(t))
9:    end for
10: end if
11: comment ← createComment()
12: comment.body ← ad.body
13: comment.annotatedElements ← finalElts
14: el.getModelRoot().ownedPackageableElements += comment
```

L'algorithme commence par rechercher l'élément concerné par le conseil ad à partir de la trace réduite tr (ligne 1). Pour effectuer cette tâche, on sélectionne un élément ($selectOne$) parmi les éléments des containers destination de la trace réduite $destEltsContainer.elementRefs$ possédant le même UID que celui embarqué dans le conseil ad. Par une simple navigation à partir de cet élément, ses ancêtres sont retrouvés (ligne 2). Avant de créer le commentaire et le connecter aux modèles de conception, les lignes 3 à 10 vérifient si le commentaire doit être attaché à des types en particulier. Si aucun type n'a été précisé (ligne 3), on considère l'ensemble de tous les ancêtres (ligne 4). Sinon, on construit l'ensemble des ancêtres qui porteront le commentaire (ligne 6 à 9). Ainsi, pour chaque type t appartenant à l'ensemble des types passés en paramètres, on sélectionne les ancêtres qui sont de type t (ligne 8). Une fois l'ensemble des ancêtres calculé, le commentaire est créé (ligne 11), son corps ($body$) est rempli avec le corps du conseil ad (ligne 12) et l'ensemble des ancêtres calculé lui sont attachés[11] (ligne 13). Finalement, le commentaire créé est ajouté à la racine du modèle de conception (ligne 14).

11. Dans UML, un *Comment* possède une référence *annotatedElements* vers les éléments qu'il annote.

5.5 CONCLUSION

Dans ce chapitre, nous avons présenté une approche pour remonter des informations de *profiling* sur les modèles de conception à l'aide de la trace réduite et d'un mécanisme d'UIDs. Les informations de *profiling* récupérées directement sur le modèle aident le concepteur de modèle à identifier les éléments impliquant les pertes de performances. Ces informations lui permettent de comprendre pourquoi les modèles de conception génèrent un code non efficace sans avoir à posséder une connaissance approfondie du code de l'application généré ou du langage de programmation utilisé. Afin d'aider au maximum le concepteur, un système expert génère des suggestions proposant des modifications à effectuer sur les modèles de conception pour atteindre de meilleures performances.

La remontée d'informations que nous avons proposé dans ce chapitre est, ici, appliquée aux informations de *profiling*, mais peut être réutilisée dans d'autres contextes. En effet, pour peu que les traces de transformations soient utilisées conjointement avec le mécanisme d'UIDs, il est possible de remonter n'importe quelle représentation de l'exécution d'une application sur les modèles de conception.

Dans le chapitre suivant, nous proposons d'illustrer notre approche sur un cas d'étude dans le contexte de GASPARD2.

6

OPTIMISATION DE MODÈLES DANS LE CONTEXTE DE GASPARD2

Dans le chapitre précédent, nous avons proposé une technique d'optimisation permettant de traiter avec l'exécution de l'application lorsque le code source de celle-ci est généré grâce à une chaîne de compilation IDM. Dans ce chapitre, nous allons illustrer cette technique sur l'une des chaînes de compilation de l'environnement Gaspard2.

Pour mémoire, l'environnement Gaspard2 propose de modéliser à haut niveau des applications dédiées au traitement de signal intensif pour des architectures pouvant être massivement parallèles. En partant d'un modèle UML profilé MARTE, l'environnement propose plusieurs chaînes de compilation, chacune visant un langage dédié.

Plus précisément, nous allons montrer ici comment les outils d'optimisation peuvent être utilisés, pour améliorer les performances de l'application modélisée, en tenant compte d'informations obtenues lors de l'exécution du programme associé.

L'application que nous allons utiliser est le *DownScaler* utilisé, entre autre, sur beaucoup de systèmes embarqués. La modélisation est effectuée avec MARTE et le code associé est généré grâce à l'environnement Gaspard2 pour une architecture matérielle GPU [1]. L'architecture GPU est une architecture dédiée aux calculs hautes performances. Pour tirer pleinement partie des capacités du GPU, l'application doit tenir compte de ces limitations matérielles et requiert des « réglages » pour pouvoir aspirer à des performances optimales. Nous allons donc voir sur cet exemple que les outils d'optimisation fournissent un bon environnement de *profiling* pour modèles.

1. Graphics Processing Unit

Nous commençons par présenter en section 6.1 l'algorithme de *DownScaling* de flux vidéo ainsi qu'une première modélisation de l'application en MARTE. En section 6.2, nous décrivons la structure de la

chaîne de compilation visant les GPU et des modifications requises pour intégrer notre approche. Par la suite, nous montrons comment visualiser et améliorer les performances de l'application en section 6.3. Finalement, nous concluons en section 6.4.

6.1 L'APPLICATION DE *downscaling*

Afin d'illustrer nos travaux sur l'observation et le *profiling* de modèles, nous allons utiliser comme support une application de mise à l'échelle d'un flux vidéo modélisé avec MARTE dans l'environnement GASPARD2. Lorsque l'on travaille avec des systèmes embarqués possédant un écran de taille réduite, il est difficile d'afficher un flux vidéo de résolution supérieure à celle proposée par la plate-forme. Il faut donc adapter la résolution du flux à celle de l'écran pour pouvoir le lire. Le *DownScaler* permet cette adaptation en passant d'une image, non compréssée, d'une certaine résolution vers une résolution plus petite. Le principe général de l'application est présenté en figure 58.

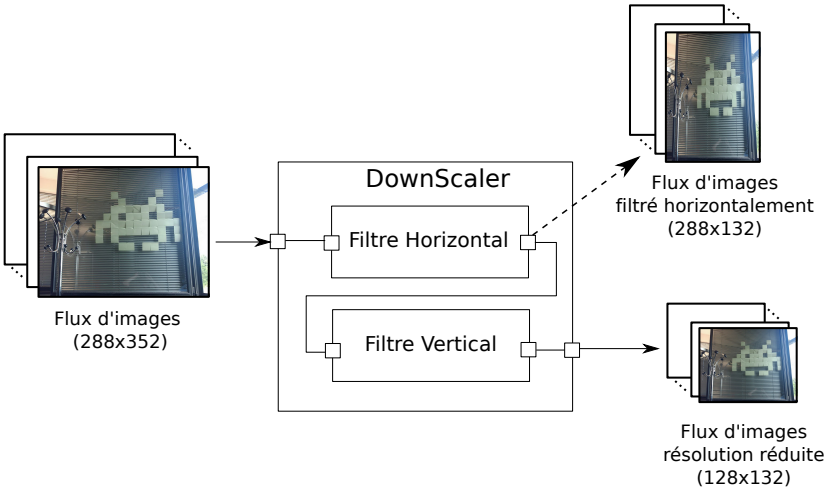


FIGURE 58: Principe du *DownScaler*

Un flux vidéo d'une résolution de 288x352 pixels est donné en entrée du *DownScaler* et celui-ci produit en sortie un flux d'images avec une résolution plus petite : 128x132 pixels. Afin de réduire la résolution du flux vidéo, le *DownScaler* s'occupe de réduire la résolution de chaque *frame*[2] qui lui est passée en entrée. Le *DownScaler* fait passer chaque *frame* par une série de deux filtres : un filtre horizontal et un filtre vertical. Le premier filtre, le filtre horizontal, s'occupe de réduire la taille de la *frame* horizontalement[3] alors que le filtre vertical termine le travail en réduisant la taille de la *frame* verticalement.

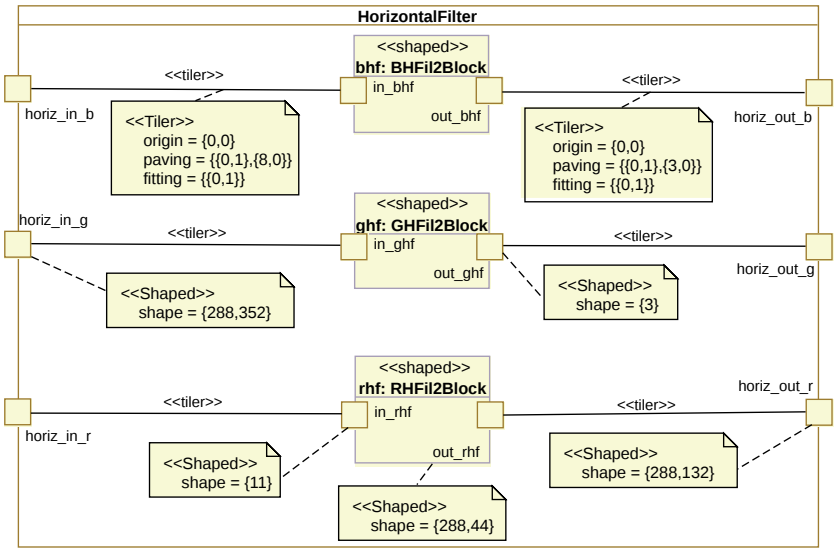
Dans les sous-sections suivantes, nous montrons comment le filtre horizontal et le filtre vertical sont représentés en MARTE avant de regarder la structure générale de l'application.

2. Une *frame* est une image d'une vidéo à un instant t et peut être encodée sous forme de tableau.

3. Sur la figure 58, cela correspond au flux d'images 288x132.

6.1.1 *Filtre horizontal*

Le filtre horizontal et le filtre vertical, travaillent plus ou moins de la même manière en effectuant une réduction sur une toute petite partie de la *frame*. Afin de couvrir toute l'image, la réduction est répétée sur chaque morceau de la *frame*.



Afin de ne pas surcharger l'illustration et étant donné l'égalité de certaines *tagged value*, nous ne les avons pas toutes représentées. Les *Tiler* d'entrée ont tous la même valeur (même *origin*, *paving* et *fitting*), les ports d'entrée du filtre portent la même *shape* ($\{288, 352\}$), les ports d'entrée des sous composants ont tous la même *shape* : $\{11\}$, de même que les ports de sortie (*shape*$= \{3\}$), les *Tiler* de sortie du filtre et les ports de sortie du filtre.

FIGURE 59: Filtre horizontal du *DownScaler*

Le composant de filtre horizontal, exprimé en MARTE, est représenté en figure 59. Chaque *frame* qu'il reçoit est préalablement décomposée en trois composantes selon le format RGB[4]. Le composant *HorizontalFilter* possède donc trois ports d'entrée, *horiz_in_r*, *horiz_in_g* et *horiz_in_b* recevant, respectivement, les composantes RED, GREEN et BLUE de la *frame* à traiter. Il est précisé que chaque port d'entrée prend une *frame* de $288, 352$ pixels[5] et que chaque port de sortie produit une *frame* de $288, 132$ pixels. Le filtre horizontal traite ensuite les trois composantes grâce à 3 sous composants : *rhf :RHFil2Block*, *ghf :GHFil2Block* et *bhf :BHFil2Block*.

La modélisation proposée ici tire avantage d'un parallélisme fort du calcul des données. Les données de la *frame* d'entrée sont manipulées par morceaux qui sont traités parallèlement sans ordre prédéfini. Chaque sous composant possède un port d'entrée et un port de sortie, par exemple, le sous composant *rhf :RHFil2Block* possède un port d'entrée *in_rhf* et un port de sortie *out_rhf*. Les valeurs de leurs *shapes* précisent qu'en entrée du sous composant, 11 ($\mathtt{shape} = \{11\}$) éléments sont récupérés et qu'en sortie, 3 éléments sont produits ($\mathtt{shape} = \{3\}$). Ce sous composant, représentant une *tâche* particulière est exécuté plusieurs fois en parallèle. La *shape* de ce sous composant précise qu'il est répété $288 * 44$, soit sur les 288 lignes et les 44 colonnes du flux d'entrée. Au final, il va donc consommer $11 * 288 * 44$ données en entrée et produire $3 * 288 * 44$ données.

Les indices des données récupérées sur la *frame* d'entrée sont calculés en fonction des *Tilers*[6] d'entrée, situés entre les ports d'entrée du composant *HorizontalFilter* et des sous composants. Les données calculées par chaque sous composant sont ensuite données par groupe de 3 pixels aux ports de sortie du filtre pour recomposer la *frame* réduite

4. RED GREEN BLUE

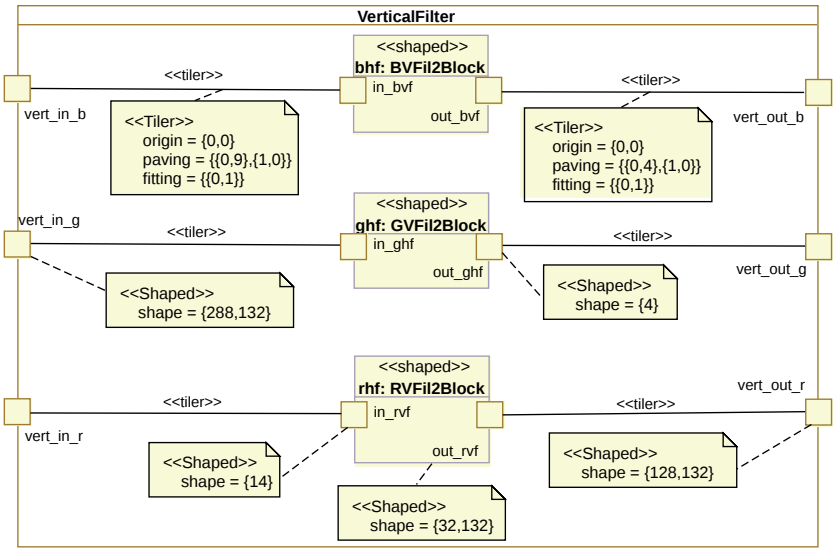
5. Information donnée par la *tagged value shape* associées aux ports d'entrée.

6. Un *Tiler* est un objet qui exprime de manière formelle la façon de récupérer et de distribuer des données de tableaux dans un contexte de tâches parallèles [19].

horizontalement. Le placement des données par groupe de 3 dans la *frame* de sortie est spécifié par le *Tiler* de sortie.

6.1.2 *Filtre vertical*

À l'image du filtre horizontal, le filtre vertical, représenté en figure 60, possède la même structure. Il possède trois ports d'entrée : *vert_in_r*, *vert_in_g* et *vert_in_b* correspondant aux trois composantes RED, GREEN et BLUE d'une *frame*. La *frame* entrant dans le filtre est, en revanche, consommée par les sous composant effectuant les réductions par groupe de 14 éléments et 4 éléments de sorties sont générés.



Afin de ne pas surcharger l'illustration et étant donné l'égalité de certaines *tagged value*, nous ne les avons pas toutes représentées. Les *Tiler* d'entrée ont tous la même valeur (même *origin*, *paving* et *fitting*), les ports d'entrée du filtre portent la même *shape* ($\{288, 132\}$), les ports d'entrée des sous composants ont tous la même *shape* : $\{14\}$, de même que les ports de sortie (*shape*$= \{4\}$), les *Tiler* de sortie du filtre et les ports de sortie du filtre.

FIGURE 60: Filtre vertical du *DownScaler*

Finalement, la structure générale du *DownScaler* en MARTE est représentée en figure 61. Chaque *frame* est donnée en entrée du *Downscaler* selon les trois composantes de base RGB. La *frame* passe par ce filtre horizontal *ihf :HorizontalFiler*, puis par le filtre vertical *ivf : VerticalFiler* avant d'être donnée en sortie du *Downscaler*.

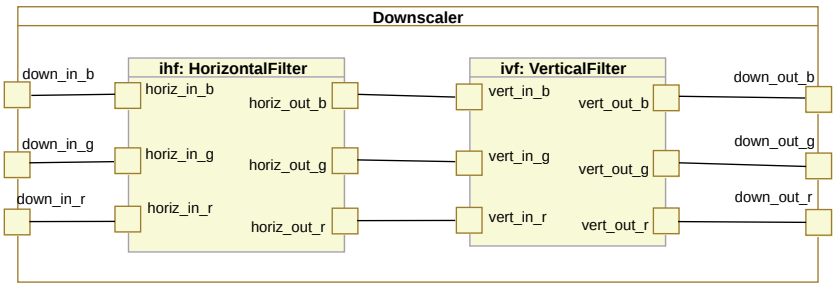


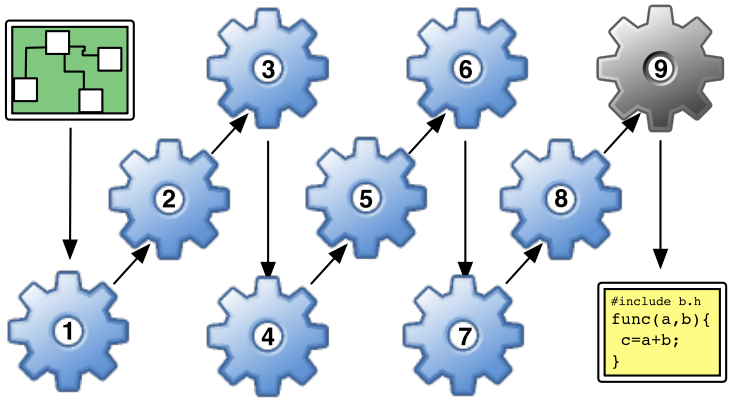
FIGURE 61: Structure du *DownScaler*

6.2 DE MARTE VERS DU CODE GPU

Une chaîne de compilation IDM de l'environnement GASPARD2 permet de générer du code OPENCL efficace [94, 95] à partir d'un modèle UML profilé MARTE. Dans les sections suivantes, nous présentons la structure générale de la chaîne de compilation avant de nous intéresser à l'outil de *profiling* que nous avons utilisé pour les expérimentations.

6.2.1 *Structure générale de la chaîne de compilation*

La chaîne de compilation a déjà été présentée au chapitre 1, page 18. Celle-ci est représentée à nouveau en détail en figure 62.



Modules de Transformation

1 : UML vers MARTE
2 : Identification des instances
3 : Calcul des Tilers
4 : Génération du graphe local
5 : Génération du graphe global
6 : Ordonnancement statique
7 : Allocation mémoire et définition de variables
8 : Conception hybride
9 : Génération de code

FIGURE 62: Chaîne de compilation GASPARD2 vers du code OPENCL

La chaîne de compilation vers OPENCL comprend 9 transformations : une séquence 8 transformations suivies par 1 transformation de modèle vers code. Chacune des transformations permet de raffiner petit à petit le modèle initial pour se rapprocher au maximum de la structure du code généré.

1. La première transformation permet de passer le modèle UML entièrement dans le domaine MARTE. Il s'agit d'un type de transformation classique permettant de passer du domaine UML profilé vers le méta-modèle correspondant. Cette première transformation est en partie une transformation 1 pour 1, c'est-à-dire qu'à un élément dans le modèle UML correspond un élément dans le modèle MARTE.
2. La seconde transformation, comme les 6 suivantes, est une transformation localisée. Elle ajoute une notion d'*instance de port* au méta-modèle MARTE. En effet, en MARTE comme en UML, si une classe possède des ports, les instances de cette classe n'ont pas d'instance de port mais font directement référence au port de la classe dont elles sont instances. Afin de manipuler le port d'une instance, il faut donc manipuler le couple (instance de classe, port de la classe), ce qui est loin d'être pratique quand la classe compte plusieurs instances et, surtout, plusieurs ports. Pour remédier à

ce manque, le concept de *PortInstance* correspondant directement à une instance de port est introduit.

3. La troisième transformation s'occupe de transformer les *Tilers*, considérés dans MARTE comme des connecteurs, en tâches prenants des éléments d'une certaine façon dans un tableau pour les ranger, éventuellement autrement, dans un autre.
4. La quatrième transformation génère un graphe de dépendances de tâches pour chaque composant présent dans le modèle en fonction des *PortInstances* introduits par la seconde transformation.
5. La cinquième étape utilise les différents graphes de dépendances générés pour chaque composant afin d'en déduire le graphe de dépendances de tâche global, c'est-à-dire, sur l'ensemble du modèle d'application et non plus sur chaque composant uniquement.
6. La sixième étape réutilise le graphe de dépendances global pour effectuer un ordonnancement statique des tâches. C'est cette transformation qui détermine l'ordre d'exécution de chacune des tâches présentes dans le modèle.
7. La septième transformation définit, à partir des ports placés sur une mémoire, l'espace d'adressage des variables, correspondant à chacun de ses ports, qui seront déclarées dans le code cible.
8. La huitième transformation permet de distinguer et de séparer les tâches qui seront exécutées sur un CPU de celles qui seront exécutées sur un GPU.
9. Finalement, la neuvième et dernière transformation génère le code OpenCL de l'application.

Il faut noter que les opérations effectuées par les transformations de la chaîne ne sont pas dédiées aux architectures GPU, mais peuvent être utilisées pour un autre langage cible. Les seules transformations spécifiques aux architectures GPU sont la transformation 8 et, évidemment, la transformation 9 générant le code OpenCL.

6.2.2 *Les outils de profiling OpenCL, format et informations récupérées*

Pour le *profiling* de code OpenCL, nous avons utilisé l'outil proposé par NVidia : Compute Visual Profiler [93]. Cet outil permet d'effectuer un certain nombre de mesures lors de l'exécution de l'application sur le GPU et de les reporter dans un fichier texte classique que nous appelons *log* de *profiling* dans la suite du chapitre. À partir de ce fichier, l'outil de *profiling* peut aussi générer une vue graphique des informations de *profiling* récupérées.

Un exemple du *log* de *profiling* généré à l'exécution est montré en listing 6.1. De manière générale, les 4 premières lignes du fichier donnent les informations suivantes :

ligne 1. la version de l'outil de *profiling*,

ligne 2. le modèle du GPU utilisé,

ligne 3. si le format utilisé pour les donnée est le CSV[7],

ligne 4. un *timestamp*.

7. Comma-Separated Values

Dans l'exemple présenté, il est précisé[8] que le matériel GPU sur lequel l'application a été lancée est un Tesla T10 et que le format utilisé dans le *log* pour reporter les informations est le CSV.

8. Sans relever les informations liées à la version de l'outil et au *timestamp*.

```
1 # OPENCL_PROFILE_LOG_VERSION 2.0
2 # OPENCL_DEVICE 0 Tesla T10 Processor
3 # OPENCL_PROFILE_CSV 1
4 # TIMESTAMPFACTOR fffff6f3ec7ecd10
5 method,gputime,cputime,ndrangesizeX,ndrangesizeY,workgroupsizeX,
      workgroupsizeY,workgroupsizeZ,occupancy
6 memcpyHtoDasync,6.016,30.000
7 memcpyHtoDasync,5.248,22.000
8 k_KRN,7.520,21.000,100,1,10,1,1,0.250
9 memcpyDtoHasync,5.504,78.000
```

Listing 6.1: Extrait d'un fichier généré par l'outil de *profiling*

La ligne 5 détaille les informations représentées dans la suite du fichier ainsi que l'ordre dans lequel elles sont rangées. Dans ce *log*, chaque ligne contient 9 informations, qui peuvent être choisies par l'utilisateur. Par exemple, le premier champ : *method* est le nom du *kernel*[9] exécuté sur le GPU et le second champ : *gputime* indique le temps mis par la méthode pour s'exécuter sur le GPU [93]. Il est à noter que, concernant le nom de la méthode exécutée (le champ *method*), il existe deux noms génériques utilisés pour représenter les transferts de mémoire du CPU vers le GPU et vice-versa. Les noms proposés dans ces cas sont *memcpyHtoDasync* (ligne 6 et 7) et *memcpyDtHasync* (ligne 9).

9. Un *kernel* correspond à une fonction spécifique que le GPU peut exécuter.

6.2.3 *Préparation de la chaîne de compilation, interfaçage avec l'outil de profiling*

Comme nous l'avons vu dans le chapitre précédent, pour pouvoir remonter les informations contenues dans le fichier produit par l'outil de *profiling*, nous avons fourni un mécanisme reposant sur les UIDs des éléments du dernier modèle[10]. Ces UIDs doivent apparaître dans le *log* de *profiling* afin de pouvoir reconnecter les mesures obtenues par le *profiler* à leurs éléments de modèles respectifs. Il faut donc un moyen d'ajouter ces UIDs au *log* généré par l'outil Compute Visual Profiler.

10. Dans la chaîne OpenCL, ce modèle correspond à celui produit par la transformation 8 et consommé par la transformation 9.

Compte tenu des différentes mesures et des différents champs disponibles dans le *log* de profiling généré, la seule information commune entre le code OpenCL et le *log* de *profiling* est le nom de la méthode exécutée par le GPU. C'est par le biais de ce nom que, lors d'une activité de *profiling* classique, un développeur OpenCL rattache les informations obtenues par le *profiler* à son code. Pour pouvoir appliquer notre approche, nous avons besoin de faire apparaître les UIDs dans ce *log* de *profiling*. Nous les ajoutons au nom des méthodes exécutées par le GPU[11].

11. Selon le même principe que celui abordé au chapitre précédent.

Pour ajouter les UIDs sur les noms des méthodes, deux approches sont possibles. La première consiste à modifier la génération de code (transformation 9 de la chaîne OpenCL) afin d'y ajouter la génération de l'UID ; cette modification pouvant être importante. La seconde approche consiste à insérer une nouvelle transformation dans la chaîne OpenCL, avant l'étape de génération de code. Cette transformation concatène l'UID des éléments à leur nom, lorsque ceux-ci en possèdent. De cette façon, la transformation de modèles vers texte n'est pas modifiée et les UIDs apparaissent dans le code OpenCL généré sans modifier une seule transformation originale de la chaîne.

La figure 63 présente la chaîne de transformations OpenCL implémentant la seconde approche proposée.

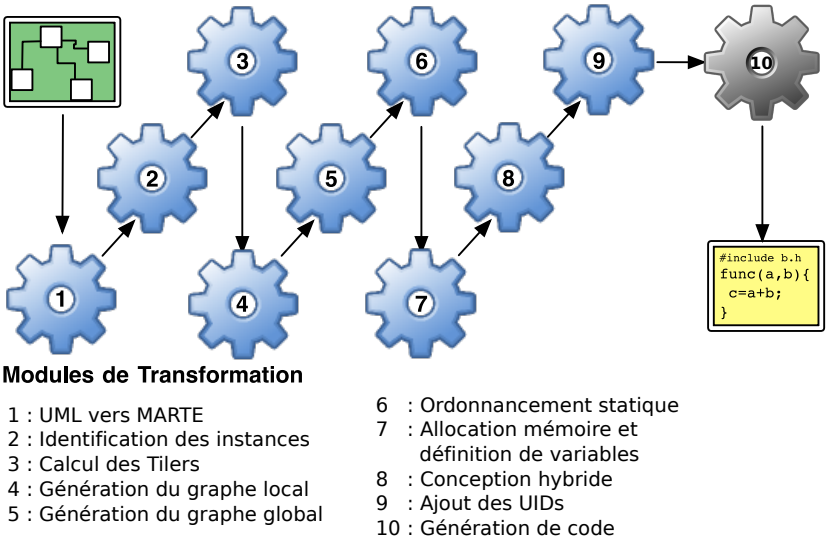


FIGURE 63: Chaîne de compilation GASPARD2 vers du code OpenCL modifiée

La chaîne comporte maintenant 10 transformations : 9 transformations de modèles à modèles et 1 transformation de modèle vers texte. La transformation ajoutée à la chaîne est la neuvième transformation qui s'occupe uniquement de concaténer au nom de chaque éléments du modèle son UID.

6.2.4 *Création de la base de connaissances pour le* GPU

Au chapitre précédent, nous avons fait état d'une base de connaissances embarquant les diverses spécifications du matériel sur lequel l'application est exécutée. Pour cet exemple, la base de connaissances doit être construite pour les GPU. Afin d'ajouter quelques informations supplémentaires au modèle de base de connaissance, nous avons étendu le méta-modèle de spécification de matériel que nous avons présenté au chapitre précédent [12]. Le méta-modèle que nous avons considéré dans cette étude de cas est présenté en figure 64. Cependant, compte tenu de la très forte dépendance du vocabulaire utilisé au monde GPU, nous ne rentrerons pas dans les détails sur le sens de tout les termes utilisés. Il est évident que les modèles de cette base de connaissances orientée GPU sont produits par des experts GPU.

12. *C.f*, figure 54 page 100.

Nous avons ajouté au méta-modèle initial une méta-classe : *GPU_Device* ainsi que deux énumérations *ComputeCapability* et *AllocationGranularity*. Ces deux énumérations sont utilisées par les attributs *CC* et *AG* de la méta-classe *GPU_Device*. Le premier attribut indique la version de la capacité de calcul du GPU, par exemple *cc10* représente la version 1.0 de la capacité de calcul. Quant à l'attribut *AG*, il permet de préciser quelle est la granularité des allocations qui sont faites sur le GPU.

Un extrait du modèle de base de connaissances du matériel GPU est présenté en figure 65. Sur cet extrait, il est possible de voir les 10 *Features* du matériel *NVidia GPU 1.3*. La *Feature* mise en surbrillance dans cet extrait : *TW* indique le nombre de *Workitems* par *Warp* qui est de 32. Cette information représente la quantité de *threads* qui peuvent être actifs en même temps sur l'architecture.

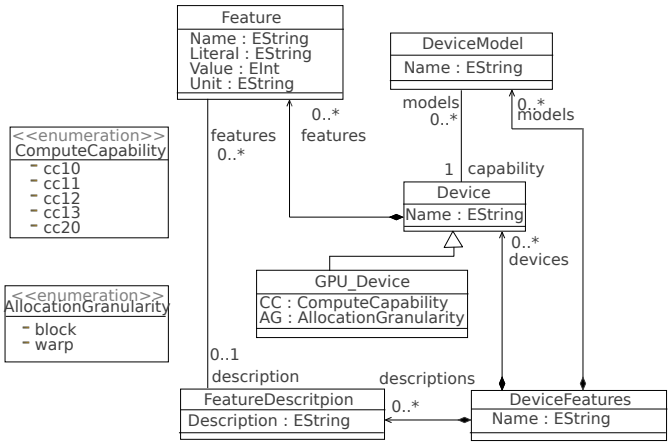


FIGURE 64: Méta-modèle de spécification de matériel étendu

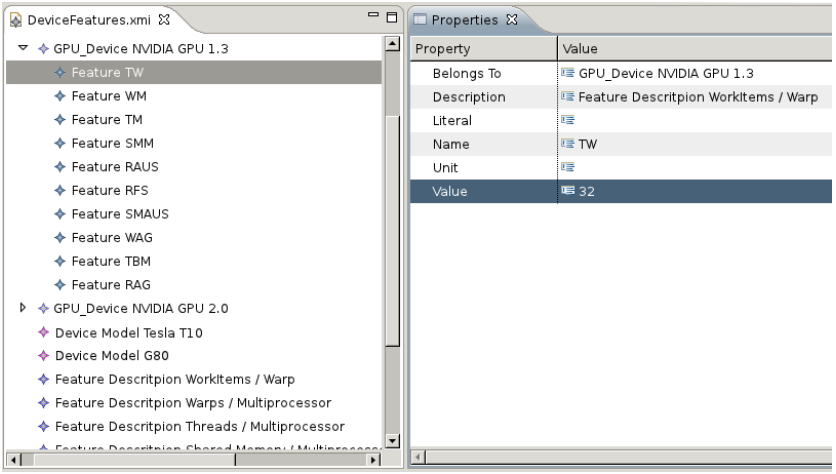


FIGURE 65: Extrait du modèle de base de connaissances pour GPU

6.3 AMÉLIORATION DU MODÈLE MARTE

Les étapes de la section précédente [13] sont effectuées par les développeurs de la chaîne et les spécialistes GPU afin d'ajouter un « support au informations de *profiling* » pour les modèles. Elles sont transparentes et invisibles au concepteur de modèles qui n'en a pas connaissance.

13. L'analyse de l'outil de *profiling*, la modification de la chaîne de transformations et la création de la base de connaissances des spécifications du matériel.

Une capture d'écran de l'environnement proposé au concepteur par GASPARD2 est visible en figure 66. Cet environnement est intégré à ECLIPSE et repose sur l'outil *Papyrus UML* pour modéliser graphiquement le système embarqué. Sur cette capture d'écran, l'application modélisée est le *DownScaler*, on y retrouve les composant *HorizontalFilter* et *Downscaler* présenté dans les sections précédentes.

Dans les sous sections suivantes, nous présentons le paramètre que nous voulons améliorer, les différentes étapes produites pour la remontée d'informations sur le modèle de conception, puis comment le modèle est modifié pour impacter les performances de l'application. Pour effectuer nos tests, nous avons utilisé la configuration matérielle suivante :

- GPU AMD 8-CORE@2.4GHz avec 64GB de RAM,
- GPU NVIDIA S1070 4 devices TESLA T10 (4GB RAM chacun) - COMPUTE CAPABILITY 1.3,

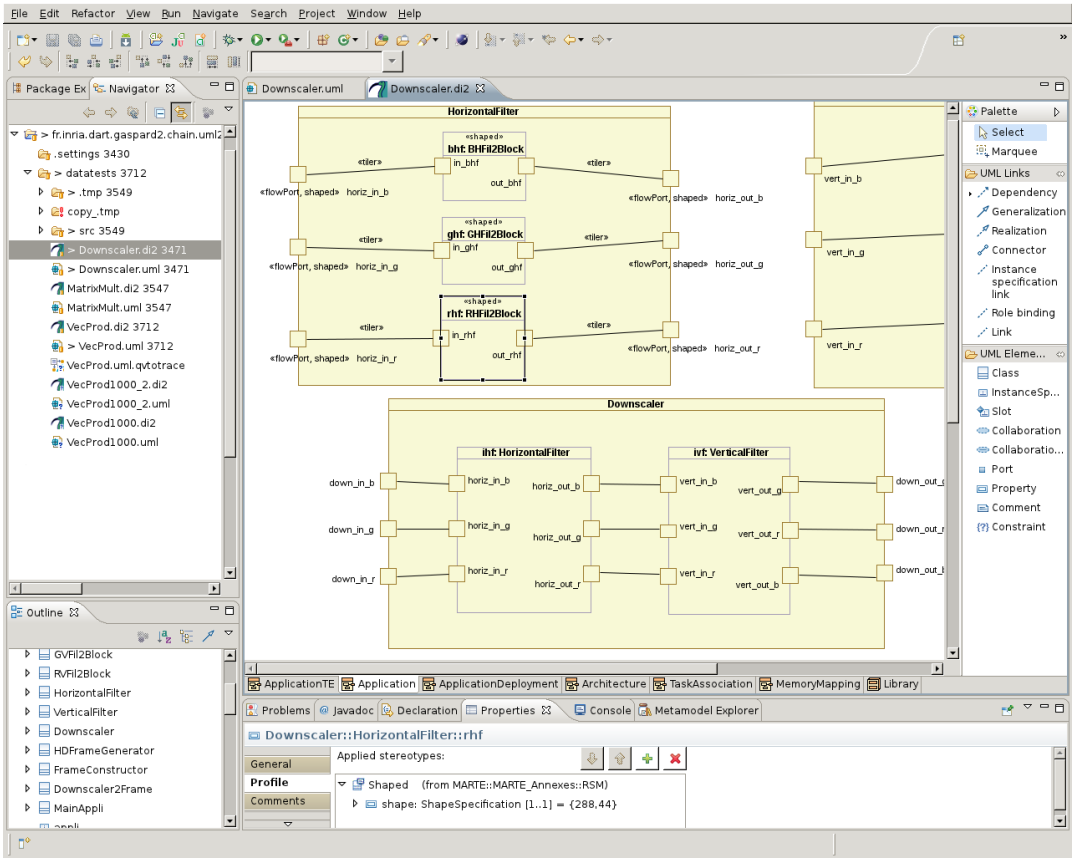


FIGURE 66: Extrait du *DownScaler*modélisée dans l'envrionnement GASPARD2

– LINUX, GCC 4.1.2, OPENCL 1.0.

6.3.1 *Amélioration de l'occupation* GPU *par l'application*

Parmi toute les mesures provenant du *profiler*, le facteur d'*occupation* d'un *kernel* sur le GPU peut avoir un impact important sur les performances. En effet, lors du lancement d'un *kernel*, le GPU génère plusieurs *work-items* [14]. Chaque *work-item* peut être soit en activité, c'est-à-dire qu'il effectue un ensemble de calcul, soit en attente. Le facteur d'occupation correspond au pourcentage de temps durant lequel l'ensemble des *work-items* issus d'un *kernel* est en activité. Plus ce facteur est bas, moins la capacité matérielle est exploitée. Au contraire, plus il est élevée, plus la puissance de calcul à disposition est élevé. Sur les architectures *gpu*, améliorer ce paramètre entraîne, généralement, une amélioration des temps d'exécution de l'application. Nous allons essayer d'améliorer ce paramètre de performance pour l'application du *DownScaler* et ainsi tenter de d'améliorer les performances en terme de temps de l'application.

14. Ou *threads* dans la terminologie générale.

6.3.2 *Exécution du DownScaler*

Une fois que l'application du *DownScaler* est modélisée, la chaîne de transformations OPENCL, avec l'ajout des UIDs, est exécutée afin de générer automatiquement le code de l'application. Le code est ensuite compilé, puis exécuté. Afin de limiter le nombre de résultats obtenus

et la taille du fichier de *profiling* généré, lors des expérimentations, nous avons lancé le *DownScaler* sur un ensemble de 2000 frames, ce qui correspond à, environ 80s de vidéo, ce qui est néanmoins suffisamment représentatif pour pouvoir parler de temps d'exécution. Dans un premier temps, nous présentons un extrait du code OpenCL produit par la chaîne modifiée [15], puis nous montrons un extrait du fichier de *profiling* produit pendant l'exécution du *DownScaler*.

15. La chaîne avec la transformation ajoutant les UIDs au dernier modèle avant la génération de code

Code OpenCL *généré par la chaîne de transformations*

Le listing 6.2 représente un morceau de code OpenCL généré par la chaîne de compilation et qui sera exécuté sur le GPU. Ce code correspond à, environ, 10% du code de l'application générée. Il contient 61 lignes s'articulant autour de 3 grandes étapes et fonctions :

- la consommation des données à traiter (lignes 22 à 41),
- l'appel à l'opération de réduction correspondant au filtre horizontal (ligne 42),
- le stockage des données récupérées par l'opération de réduction (lignes 43 à 58).

Quant à la déclaration et implémentation de l'opération de réduction, celle-ci va de la ligne 8 à 12.

```
1  /*
2    Gaspard2 MDE
3    Auto Generated OpenCL Code for GPU
4    Kernel Name: rhf_KRN
5  */
6
7  // +- IP Functions
8    void horizontal_filter(int* a, int* b) {
9      b[0] = ((a[0] + a[1] + a[2] + a[3] + a[4] + a[5]) / 6) - ((a
           [0] + a[1] + a[2] + a[3] + a[4] + a[5]) % 6);
10     b[1] = ((a[2] + a[3] + a[4] + a[5] + a[6] + a[7]) / 6) - ((a
           [2] + a[3] + a[4] + a[5] + a[6] + a[7]) % 6);
11     b[2] = ((a[5] + a[6] + a[7] + a[8] + a[9] + a[10]) / 6) - ((a
           [5] + a[6] + a[7] + a[8] + a[9] + a[10]) % 6);
12   }
13
14 //Kernel Start
15 __kernel void rhf_KRN__xYqtt97iEeCKW4L9SqmXFw(uint iNumElements,
       __global int* out_rhf_rhf_KRNPAR,const __global int*
       in_rhf_rhf_KRNPAR)
16 {
17   int in_rhf_rhf[6];
18   int out_rhf_rhf[1];
19   int iGID = get_global_id(0)+get_global_size(0)*get_global_id(1)+
         get_global_size(0)*get_global_size(1)*get_global_id(2);
20
21   // bound check
22   if (iGID < iNumElements)
23   {
24     { //input tiler processing
25
26       uint tlIter[2];
27       uint tl[1];
28       uint ref[2];
29       uint index[2];
30
```

```
31      tlIter[0] = iGID%288;
32      tlIter[1] = abs(iGID/288);
33
34      ref[0] = 0 + 1*tlIter[0] + 0*tlIter[1];
35      ref[1] = 0 + 0*tlIter[0] + 8*tlIter[1];
36      for(tl[0] = 0; tl[0] < 11; tl[0]++) {
37       index[0] = (ref[0]+ 0*tl[0])%288;
38       index[1] = (ref[1]+ 1*tl[0])%352;
39       in_rhf_rhf[tl[0]*1] = in_rhf_rhf_KRNPAR[index[0]*352 +
             index[1]*1];
40      }
41     }
42     horizontal_filter(in_rhf_rhf, out_rhf_rhf); //IP Call
43     { //output tiler processing
44      uint tlIter[2];
45      uint tl[1];
46      uint ref[2];
47      uint index[2];
48
49      tlIter[0] = iGID%288;
50      tlIter[1] = abs(iGID/288);
51
52      ref[0] = 0 + 1*tlIter[0] + 0*tlIter[1];
53      ref[1] = 0 + 0*tlIter[0] + 3*tlIter[1];
54      for(tl[0] = 0; tl[0] < 3; tl[0]++) {
55       index[0] = (ref[0]+ 0*tl[0])%288;
56       index[1] = (ref[1]+ 1*tl[0])%132;
57       out_rhf_rhf_KRNPAR[index[0]*132 + index[1]*1] = out_rhf_rhf
             [tl[0]*1];
58      }
59     }
60   } else return;
61 }
```

Listing 6.2: Extrait du code OpenCL généré pour la composante Red du filtre horizontal du *DownScaler*

Dans ce code, on remarque que le *kernel* qui sera exécuté sur le GPU porte, en plus de son nom, un UID (encadré ligne 15). Ici, l'UID est *_xYqtt97iEeCKW4L9SqmXFw*, cela signifie que le l'élément portant l'UID *_xYqtt97iEeCKW4L9SqmXFw* dans le dernier modèle de la chaîne de compilation a servi à produire entièrement, ou, tout du moins, une partie du code du *kernel*.

Ce code, généré à partir des modèles de conception, est ensuite compilé puis exécuté. Avec la version actuelle des modèles de conception, le *DownScaler* compilé met 13, 035s pour traiter 2000 frames d'une vidéo.

Fichier de profiling généré

Le fichier de *profiling* généré pendant l'exécution de l'application contient 21005 lignes dont 21000 lignes de mesures. Le listing 6.3 présente un extrait du fichier généré. Cette fois, nous avons demandé plus d'informations d'exécution que pour le listing 6.1. Nous pouvons voir, entre autre, qu'une mesure de l'occupation est présente dans le *log* (ligne 5). Cette mesure est disponible pour chaque exécution de *kernel* (ligne 9, 10 et 11). Par exemple, la ligne 9 reporte une occupation de 75% de la part du *kernel* : *rhf_KRN__xYqtt97iEeCKW4L9SqmXFw* [16]. À

16. Ce kernel correspondant à celui présenté au listing 6.2.

partir du nom du *kernel*, il est possible de retrouver l'UID de l'élément à partir duquel le code du *kernel* a été généré.

```
5  timestamp,gpustarttimestamp,method,gputime,cputime,ndrangesizeX,
       ndrangesizeY,workgroupsizeX,workgroupsizeY,workgroupsizeZ,
       stapmemperworkgroup,regperworkitem,occupancy,streamid,
       local_load,local_store,gld_request,gst_request,
       memtransfersize,memtransferdir,memtransferhostmemtype
6  1398860.000,12437ed6ea214f80,memcpyHtoDasync,76.064,686.000, , ,
       , , , , , ,1, , , , ,405504,1,0
7  1399745.000,12437ed6ea294cc0,memcpyHtoDasync,75.008,334.000, , ,
       , , , , , ,1, , , , ,405504,1,0
8  1400445.125,12437ed6ea33eec0,memcpyHtoDasync,75.488,319.000, , ,
       , , , , , ,1, , , , ,405504,1,0
9  1400916.000,12437ed6ea392900,rhf_KRN__xYqtt97iEeCKW4L9SqmXFw,
       94.432,224.000,288,1,44,1,1,28,20,0.750,1,0,0,176,48
10 1401266.000,12437ed6ea3e5080,bhf_KRN__xYqtpd7iEeCKW4L9SqmXFw,
       93.152,197.000,288,1,44,1,1,28,20,0.750,1,0,0,154,42
11 1401565.000,12437ed6ea42d800,ghf_KRN__xYqt0t7iEeCKW4L9SqmXFw,
       93.344,200.000,288,1,44,1,1,28,20,0.750,1,0,0,154,42
```

Listing 6.3: Extrait du fichier de *profiling* généré pour le *DownScaler*

Ce fichier de *profiling* est ensuite *parsé* pour produire un modèle de *log* de *profiling*. Afin de générer le modèle de *log* de *profiling*, il faut convenir d'une correspondance entre les concepts du méta-modèle de *log* de *profiling* et les informations contenues dans le *log* de *profiling*. Cette correspondance est donné dans la table 2.

MÉTA-CLASSE / ATT.	*log* DE *profiling*
ProfilingModel	Fichier
name	Nom du fichier
archiModel	Valeur contenue par la ligne d'entête du fichier OPENCL_DEVICE
LogEntry	Ligne du fichier de *log* en dehors de l'entête et de la description des champs
UID	*Uid* contenu dans le nom de la valeur du champ *method*
timestamp	Valeur contenue par le champ *timestamp*
Parameter	Champ, différent du champ *timestamp* ou *method*
kind	Nom du champ
data	Valeur du champ

TABLE 2: Table de correspondance entre les champs présent dans le *log* de *profiling* et les concepts du méta-modèle de *log* de *profiling*

Un extrait du modèle de *log* de *profiling* généré à partir du *log* de *profiling* produit par l'exécution du *DownScaler* est présenté en figure 67. On y retrouve les informations qui sont présentes dans le fichier de *profiling*, notamment le pourcentage d'occupation de 75% (ligne 9).

Le modèle produit est ensuite transformé en un modèle d'*Advice* par un système expert. Ce modèle reprend en grande partie les informations obtenues par le *profiler* et ajoute un conseil donnant des directives,

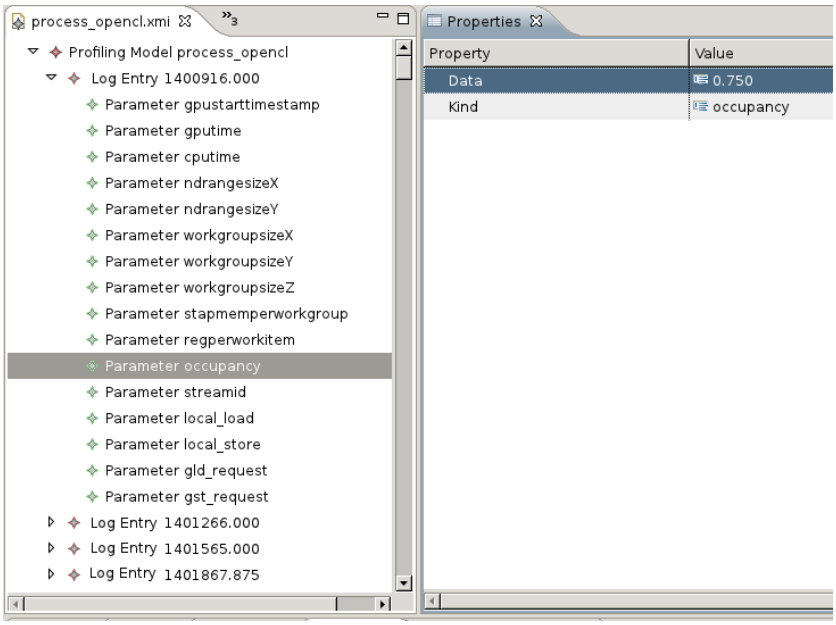


FIGURE 67: Extrait du modèle de *log* de *profiling* généré

en fonction du paramètre de performance que l'on veut améliorer, sur la manière de modifier le modèle. Le modèle d'*Advice* est produit en utilisant à la fois le modèle de *log* de *profiling* et la base de connaissance modélisée. En fonction du paramètre de performance à améliorer, le système expert va interroger la base de connaissance sur le matériel pour déduire, en fonction de ses capacités, les meilleures valeurs à choisir, ou une ligne directrice à suivre pour modifier facilement le modèle de conception et ainsi améliorer le paramètre de performance en question. La base de connaissance utilisée est, ici, la base de connaissance sur les GPU. Dans ce chapitre, nous ne montrons pas d'exemple de ce modèle étant donné qu'il reprend en très grande partie les informations contenues dans le modèle de *log* de *profiling* et que ces informations [17] sont visibles, au final sur le modèle de conception.

17. Les informations de *profiling* avec le conseil généré.

6.3.3 *Remontée d'information*

Les étapes précédentes ont permis :
- de récupérer les informations de *profiling*,
- de produire un modèle de *log* de *profiling* embarquant à la fois ces informations et des données permettant de reconnecter les informations obtenues au monde des modèles,
- de produire un modèle d'*Advice* contenant les informations de *profiling* et un conseil sur la manière de modifier le modèle de conception pour améliorer un paramètre de performance en particulier.

Pour pouvoir fournir au concepteur un retour sur son modèle, il faut maintenant retourner les informations et le conseil obtenus sur le modèle de conception. Pour remonter les informations, la trace produite pendant la génération du code est utilisée. Plus précisément, c'est la trace réduite qui est utilisée dans ce cas. La figure 68 présente un extrait de la trace réduite produite dans le cas du *DownScaler* et pour la chaîne de compilation vers OPENCL. On y retrouve l'élément

rhf_KRN__xYqtt97iEeCKW4L9SqmXFw [18] à partir duquel une partie du code du *kernel* présenté au listing 6.2 a été produit. La trace réduite nous indique que cet élément porte l'UID *_xYqtt97iEeCKW4L9SqmXFw* et qu'il possède 3 *Links* liés aux éléments du modèle d'entrée l'ayant produit.

18. Élément sélectionné

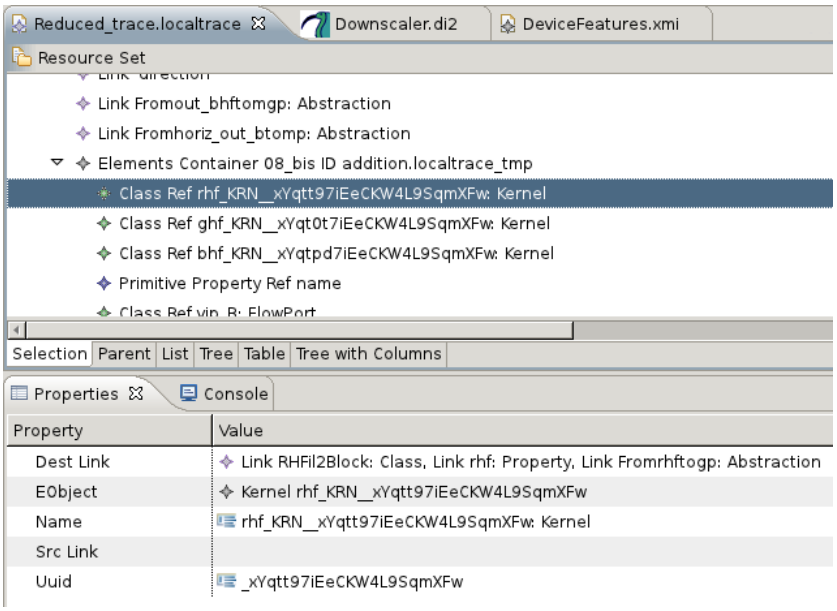


FIGURE 68: Extrait de la trace réduite générée

La table 3 contient un résumé des éléments du modèle de conception liés à l'élément en surbrillance de la figure 68. Il est possible de voir que 3 éléments du modèle de conception *RHFIL2Block : Class*, *rhf : Property* et *fromrhftogp : Abstraction* sont liés à l'élément *rhf_KRN : Kernel* dans le dernier modèle avant la génération de code par 3 *Links* : *RHFil2Block*, *rhf* et *fromrhftogp*.

ÉLEMENT SOURCE	LINK	ÉLEMENT DESTINATION
RHFIL2Block : Class	RHFIL2Block	rhf_KRN : Kernel
rhf : Property	rhf	
fromrhftogp : Abstraction	fromrhftogp	

Sur cette table, nous avons volontairement enlevé l'UID qui est concaténé au nom de l'élément destination pour des raisons de place.

TABLE 3: Ensemble d'éléments récupérés par la trace réduite

L'algorithme de remontée est donc lancé en utilisant le modèle d'*Advice* embarquant les informations de *profiling* et le conseil généré, ainsi que la trace réduite pour pouvoir reconnecter les éléments du modèle d'*Advice* au modèle du *DownScaler*. L'algorithme requiert aussi en entrée un ensemble de *type* sur lesquels les éléments doivent être remonté. Pour la gestion de l'*occupation*, les *types* intéressants sont *Property* et *Abstraction*. En effet, les éléments de type *Property* en MARTE représente les tâches qui sont exécutées sur le matériel et les éléments de type *Abstraction* représentent le placement des tâches sur le matériel. Ces deux types sont donc aussi donnés en paramètre de l'algorithme de remontée.

Par exemple, en suivant les indications de la table 3, pour l'élément portant l'UID *_xYqtt97iEeCKW4L9SqmXFw* (l'élément rhf_KRN : Kernel), les informations sont remontées sous forme de commentaire sur le modèle du *DownScaler* sur les éléments fromrhftogp : Abstraction et rhf : Property uniquement. La figure 69 présente le modèle du *DownScaler* avec les annotations sur le filtre horizontal. Sur la figure, il est possible de voir que les commentaires remontés sont associés aux sous composants de *HorizontalFiler* : *rhf : RHFil2Block*, *ghf : GHFil2Block* et *bhf : BHFil2Block*. Ces commentaires contiennent les diverses informations que l'on peut retrouver dans le *log* de *profiling* ainsi qu'un conseil proposé pour améliorer le paramètre d'*occupation*. Par exemple, pour le sous composant *ghf : GHFil2Block*, on peut facilement voir que l'*occupation* est de 75% et que pour améliorer ce paramètre, il est possible de changer la première ou la seconde composante de la *shape* de ce sous composant sans dépasser 512.

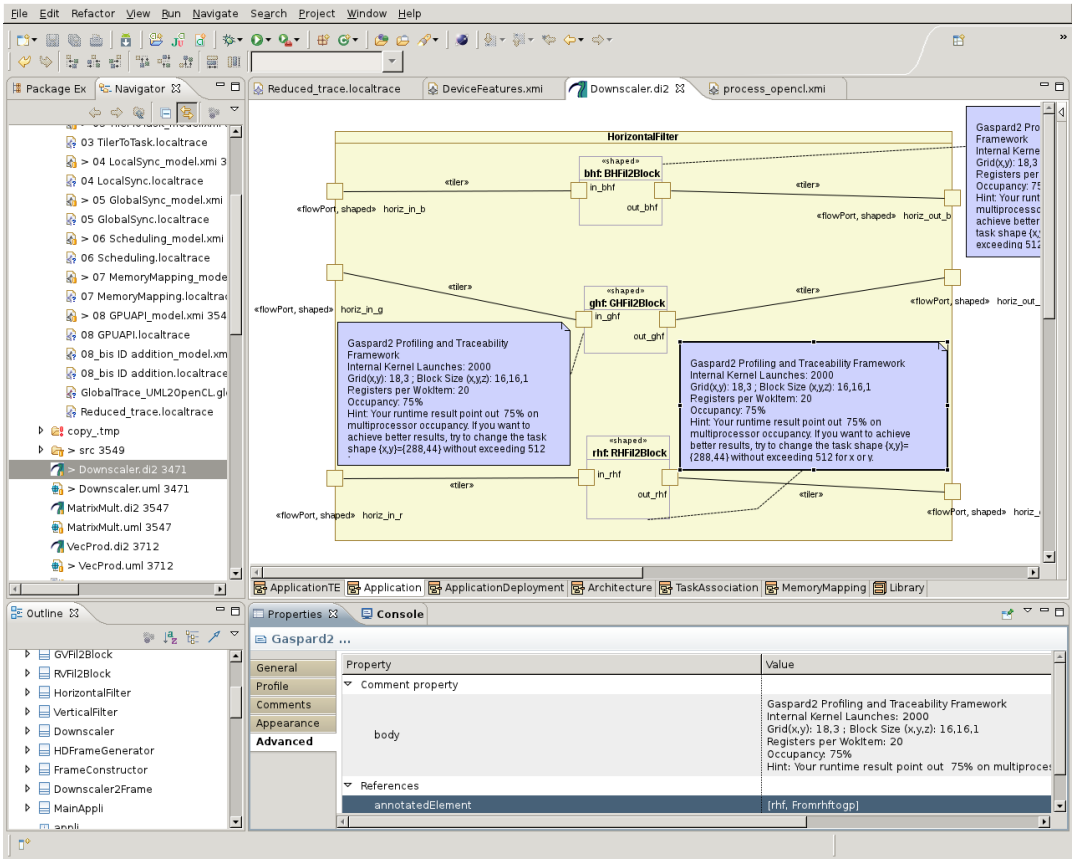


FIGURE 69: Entrait du modèle du *DownScaler* annoté avec les informations de *profiling* obtenues à l'exécution de l'application

6.3.4 *Analyse des résultats*

Les annotations retournée sur le modèle de conception donnent au concepteur de modèle une indication sur le moyen d'améliorer l'*occupation* des sous composants du filtre horizontal et du filtre vertical. Pour la suite du chapitre, nous ne nous intéresserons qu'au filtre horizontal du *DownScaler*.

6.3.5 *Modification du modèle de conception*

Pour les trois sous composants du filtre horizontal, les mesures obtenues et le conseil proposé sont les mêmes. Par exemple, pour le sous composant *rhf : RHFil2Block*, le conseil propose de modifier sa *shape*. Originalement, la *shape* $\{x, y\}$ du sous composant vaut $\{288, 44\}$ et la valeur 512 est la valeur à ne pas dépasser pour le x ou pour le y de la *shape*. Si nous regardons la *shape* du port de sortie du filtre horizontal *horiz_out_r*, nous voyons que sa première valeur (288) est la même que la première valeur de la *shape* du sous composant. De même, nous pouvons voir que la seconde valeur de la *shape* (132) du port *horiz_out_r* est égale au produit de la seconde valeur de la *shape* (44) du port *out_rhf* et de la *shape* de son port de sortie *out_rhf* ($\{3\}$) : $44 * 3 = 132$. Nous allons donc augmenter la seconde valeur du *shape* du sous composant *rhf : RHFil2Block* sans dépasser les 512. Cependant, si la *shape* du sous composant est modifiée, pour que l'application continue de fonctionner correctement, il faut aussi modifier :

- les *shapes* de ses ports d'entrée et de sortie, soit les *shapes* des ports *in_rhf* et *out_rhf*,
- les valeurs des *Tiler* entrant et sortant du sous composant,
- l'IP utilisé pour la réduction.

En effet, il faut, dans un premier temps, pouvoir garder le rapport existant : $y * z = 132$ entre la seconde composante du *shape* de *rhf : RHFil2Block* (noté y dans l'équation) et la valeur du *shape* de son port de sortie (noté z dans l'équation). Pour pouvoir augmenter la valeur y, il faut, indubitablement, diminuer celle de z. Pour pouvoir prendre la valeur de y maximale, nous décrémentons autant que possible la valeur de z que nous plaçons à 1. Par conséquent, la valeur y vaut 132. La *shape* de la sous composant et donc modifiée en : $\mathtt{shape} = \{288, 132\}$ et la *shape* du port de sortie du composant est aussi modifiée en : $\mathtt{shape} = \{1\}$.

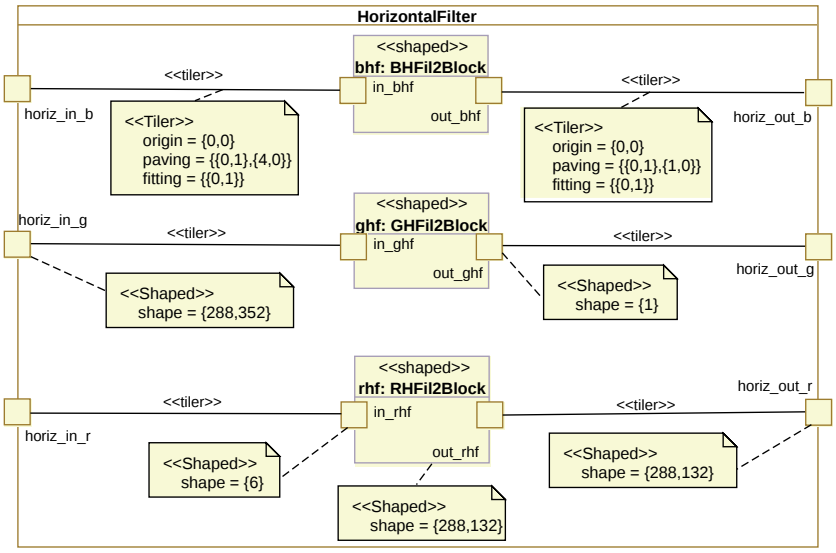
Puisqu'il n'y a plus qu'une seule valeur produite en sortie du sous composant par l'IP de réduction, il faut ranger les valeurs différemment dans le port de sortie. Au lieux de les ranger par groupe de 3, nous allons les ranger par groupes de 1. Le *paving* du *Tiler* de sortie est donc modifié en : $\mathtt{origin} = \{0, 0\}, \mathtt{paving} = \{\{0, 1\}, \{1, 0\}\}, \mathtt{fitting} = \{\{0, 1\}\}$[19].

Le port d'entrée du sous composant prenait, originalement, 11 valeurs pour en produire 3. Puisque le sous composant produit désormais 1 seule valeur, il faut réduire le nombre d'élément qui est pris en entrée. L'analyse de l'IP nous montre que pour chaque donnée produite, 6 données sont nécessaires. La *shape* du port d'entrée du sous composant est donc modifiée en : $\mathtt{shape} = \{6\}$. Comme pour le *paving* du *Tiler* de sortie, le *paving* du *Tiler* d'entrée doit être aussi modifié. Celui-ci est modifié en $\mathtt{origin} = \{0, 0\}, \mathtt{paving} = \{\{0, 1\}, \{4, 0\}\}, \mathtt{fitting} = \{\{0, 1\}\}$.

Finalement, l'IP utilisé pour la réduction est aussi modifié pour prendre 6 données en entrée et ne produire qu'une seule donnée. Il est à noter que la modification de l'IP n'est possible que si son implémentation est connue. Dans le cas ou cette modification n'est pas possible, les modifications que nous proposons ici ne peuvent pas être effectuée.

La figure 70 présente le filtre horizontale avec les modifications que nous venons de faire grâce au conseil généré et reporté sur le modèle de conception.

19. Plus d'informations sur la sémantique du *Tiler* sont présents dans [19].



Afin de ne pas surcharger l'illustration et étant donné l'égalité de certaines *tagged value*, nous ne les avons pas toutes représentées. Les *Tiler* d'entrée ont tous la même valeur (même *origin*, *paving* et *fitting*), les ports d'entrée du filtre portent le même *shape* ($\{288, 352\}$), les ports d'entrée des sous composants ont tous la même *shape* : $\{6\}$, de même que les ports de sortie (*shape*$= \{1\}$), les *Tiler* de sortie du filtre et les ports de sortie du filtre.

FIGURE 70: Filtre horizontal du *DownScaler* modifié

Retours sur le modèle de conception

Une fois que le modèle du *DownScaler* est modifié, le code OPENCL est généré et l'application compilée est exécutée à nouveau pour 2000 frames. Les informations récoltées par le *profiler* sont ensuite remontées, comme précédemment, automatiquement sur les modèles de conception. La figure 71 présente, une fois de plus, le modèle du *DownScaler* avec les annotations récupérées après l'exécution de l'application modifiée grâce aux précédent retours.

Cette fois, nous voyons que nous avons un taux d'*occupation* de 84%. Notre but initial était d'améliorer le taux d'*occupation* pour fournir une meilleure exécution de l'application. Grâce au conseil et aux retours donnés au concepteur de modèles sur le modèle de conception, il nous a été possible de passer d'un taux d'*occupation* de 75% à un taux d'*occupation* de $84,4\%$ pour les sous composants du filtre horizontal du *DownScaler*. Cependant, le temps d'exécution que nous avons mesuré est légèrement supérieur : pour réduire la taille des 80s de vidéo, l'application embarquant les modifications effectuées à partir des modèles de conception prend $13,463$s. Comme nous l'avions déjà évoqué, améliorer le taux d'occupation n'entraîne pas toujours une amélioration du temps d'exécution de l'application. En effet, pour les GPUs, il arrive un point ou l'amélioration de l'*occupation* n'entraîne plus une accélération générale de l'application. Finalement, notre but original d'amélioration du taux d'*occupation* est un succès, mais cela n'a pas entraîné l'effet escompté, à savoir l'amélioration du temps d'exécution de l'application. Il faudrait donc relancer l'approche en essayant de jouer sur une autre critère.

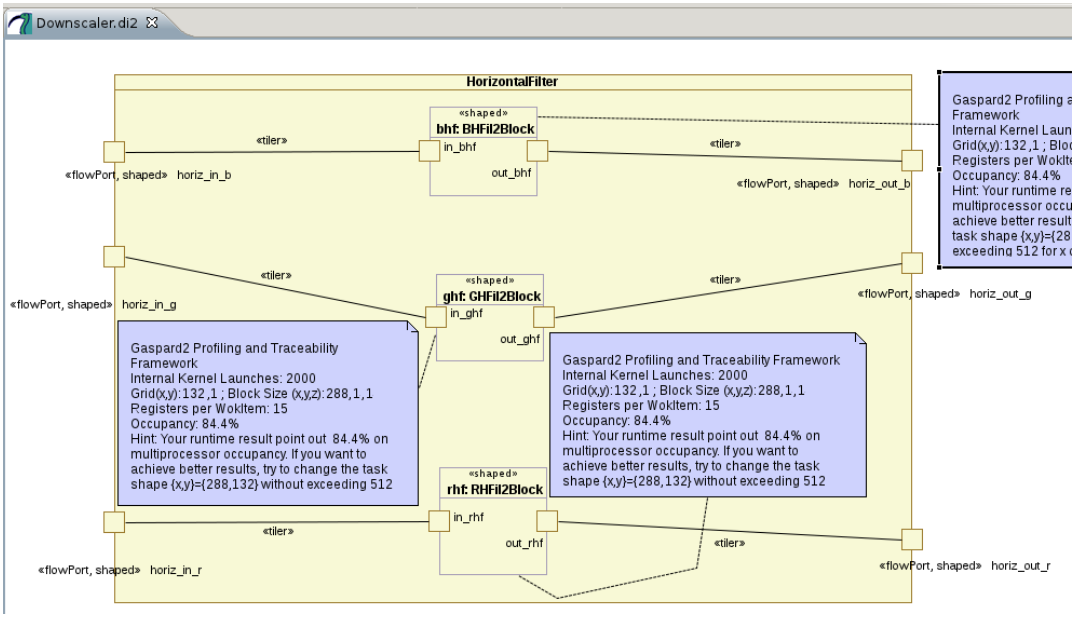


FIGURE 71: Entrait du modèle du *DownScaler* annoté avec les informations de *profiling* obtenues à l'exécution de l'application

6.4 CONCLUSION

Dans ce chapitre, nous avons montré la faisabilité et la pertinence de notre approche pour le *profiling* de modèles sur un cas d'étude modélisé en UML avec le profil MARTE dans l'environnement GASPARD2. L'application que nous avons utilisée est le *DownScaler*, utilisé, notamment, par les systèmes embarqués pour réduire la taille d'un flux vidéo. Pour essayer d'améliorer les temps d'exécution de l'application générée en OPENCL, nous avons augmenté le taux d'*occupation* du multiprocesseur en fonction d'informations récoltées à l'exécution de l'application et d'une base de connaissance sur les GPUs. En effet, l'augmentation de l'*occupation* permet généralement de réduire les temps d'exécution.

Une fois l'application modélisée, le code a été généré, compilé, puis exécuté. Le *log* de *profiling* produit par un outil dédié lors de l'exécution de l'application a ensuite été traité et les informations qu'il contient ont été automatiquement remontée sous forme de commentaires sur le modèle de conception. Ces commentaires faisant état des différentes mesures obtenues à l'exécution contiennent aussi un conseil généré par un système expert utilisant le modèle base de connaissances du matériel pour indiquer au concepteur le meilleur moyen de modifier son modèle pour améliorer un paramètre de performance en particulier. Dans ce cas d'étude, le taux d'*occupation* remonté sur les sous composant du filtre horizontal du *DownScaler* était de 75%.

En suivant le conseil prodigué par le système expert, nous avons modifié le modèle de l'application, généré à nouveau le code OPENCL, compilé, puis exécuté l'application. Cette fois, les retours obtenus sur le modèle de conception à partir des *log* de *profiling* ont fait état d'un taux d'*occupation* de 84, 4%. Cependant, le temps d'exécution de l'application n'a pas été accéléré et nous avons noté une légère augmentation du temps d'exécution. Comme nous l'avons dit, le taux d'*occupation* n'est pas toujours synonyme d'accélération de temps d'exécution. Néanmoins, dans ce chapitre, le but était d'améliorer un paramètre de

performances grâce aux retours de mesures obtenues à l'exécution de l'application, ce qui est chose faite avec l'*occupation*.

Dans ce chapitre, nous avons donc pu améliorer un paramètre de performances grâce à notre approche, présentée au chapitre précédent. Cependant, les informations que nous avons récupérées à l'exécution sont entièrement dépendantes de l'outil de *profiling* utilisé et sont récupérées pour l'ensemble de l'application, ce qui peut entraîner des *log*s de *profiling* aux tailles conséquentes. Dans le prochain chapitre, nous allons montrer un moyen de pouvoir choisir quels éléments observer et quelles informations récupérer à l'exécution de l'application.

7

OBSERVATION PARTIELLE DE MODÈLES

Dans le chapitre précédent, nous avons vu :

- comment il était possible de lier une représentation de l'exécution d'un programme à ses modèles de haut-niveau via un mécanisme d'UIDs,
- comment retrouver directement ces informations d'exécution par annotation des modèles de haut-niveau en utilisant une trace réduite.

Cependant, la représentation de l'exécution que nous avions utilisée était relative aux performances de l'application uniquement et étaient générées à partir d'un outil de *profiling* dédié. Il était alors impossible d'influencer les types d'informations à récupérer pendant l'exécution du programme et nous pouvions uniquement visualiser et corriger sur le modèle de conception les problèmes de performances de l'application.

Les problèmes liés aux performances ne sont pas les seules pouvant être rencontrés durant l'exécution d'une application en développement. En effet, le comportement de l'application peut être différent de celui que le concepteur attend. Dans ce cas, fournir une représentation de l'exécution ne reposant pas uniquement sur des informations de performances est indispensable pour permettre au concepteur d'avoir accès à un maximum de détails sur l'exécution de son application. Il serait donc intéressant pour le concepteur de modèles de posséder un mécanisme permettant d'observer, directement sur ses modèles de conception, une représentation de leurs exécutions.

Dans ce chapitre, nous présentons un moyen d'observer, à la demande du concepteur, l'exécution de parties d'un modèle. Ce travail repose sur l'utilisation conjointe du mécanisme de remontée d'informations présenté dans le chapitre précédent et d'un mécanisme d'insertion d'observations que nous présentons dans ce chapitre. Dans un premier temps, nous expliquons en section 7.1 quelle est la stratégie que nous adoptons pour récupérer une représentation de l'exécution d'un

programme. En section 7.2, nous poursuivons en présentant ce que nous entendons par « observation de modèles » et la structure générale de notre approche. Nous présentons ensuite en section 7.3 comment produire et créer des observations utilisables par le concepteur avant de montrer en section 7.4 comment ces observations sont traitées et ajoutées au programme pour fournir un compte rendu de son exécution. En section 7.5, nous montrons comment les informations générées par l'exécution de l'application sont remontées sur le modèle de conception. Finalement, en section 7.6, nous présentons nos conclusions sur ce travail.

7.1 RÉCUPÉRER DES INFORMATIONS À L'EXÉCUTION D'UNE APPLICATION

Pour récupérer des informations lors de l'exécution d'un programme classique, il faut obligatoirement passer par une modification du programme, on parle alors d'instrumentation du programme. L'instrumentation d'un programme revient à ajouter un certain nombre de lignes de codes récoltant des informations lorsque le programme est exécuté. Il existe plusieurs types d'instrumentation de code qui propose d'ajouter des fragments de code à différents niveaux [120]. Ces types d'instrumentations sont les suivants :

- manuelle,
- automatique au niveau source [54],
- assistée par le compilateur [24],
- par traduction de code binaire [60],
- à l'exécution [120],
- par injection à l'exécution [64].

Dans ce chapitre, nous discutons principalement des trois premiers types d'instrumentation et laissons de côté les trois derniers types d'instrumentation car ils sont extrêmement dépendants des possibilités offertes par l'architecture, le système d'exploitation et/ou le langage de programmation utilisé.

L'instrumentation manuelle impose au programmeur d'ajouter lui-même les fragments de code au niveau du code source de son programme. Il s'agit de la façon la plus répandue parmi les programmeurs pour obtenir rapidement une représentation partielle de l'exécution de leur programme. Cependant, cette tâche peut être longue et complexe, surtout si le programmeur n'a pas de connaissance précise du code source de l'application. Dans l'environnement GASPARD2 et tout autre environnement travaillant avec des chaînes de compilation IDM, le concepteur de modèle n'est pas obligatoirement un expert dans le langage visé par la compilation et le code source de l'application généré peut faire plusieurs milliers de lignes. Dans ce cas, il n'est pas apte à modifier correctement le code source de l'application dans le but de l'instrumenter.

L'instrumentation assistée par le compilateur propose de laisser le compilateur ajouter, au moment de la compilation, des fragments de code pour récolter des informations à l'exécution. Typiquement, les fragments de code ajoutés sont ceux insérés, par exemple, par GCC lors d'une phase de compilation avec l'option *-g*. Ce type d'instrumentation est intéressant car il ne requiert aucune intervention de l'utilisateur, mais reste très dépendant du compilateur, du langage utilisé et aussi des possibilités de l'architecture et du système d'exploitation. De plus,

l'environnement GASPARD2 dans lequel nous travaillons propose plusieurs langages cibles comme traduction du modèle de conception.

Finalement, le type d'instrumentation automatique au niveau source est le type d'instrumentation qui considère, à notre avis, le plus d'avantages. Ce type d'instrumentation propose d'ajouter automatiquement, grâce à un outil, des fragments de code au code source de l'application. Cette instrumentation est contrôlée selon une politique particulière précisant à quels endroits les fragments de code peuvent être ajoutés. C'est ce type d'instrumentation que nous retenons pour notre approche car elle peut être facilement adaptée à n'importe quel langage de programmation. Choisir d'interfacer un compilateur plutôt qu'un autre rendrait notre solution dépendante d'un langage en particulier et obligerait le concepteur à se focaliser sur un langage en particulier comme langage cible pour son application.

Actuellement, les approches reposant sur l'instrumentation automatique d'un code source tentent d'instrumenter le plus possible la totalité du code source d'un programme [118] afin de donner une « cartographie » la plus précise possible de l'exécution d'un programme, par exemple, en terme de nombre de fonctions exécutées ou encore d'allocations mémoire effectuées. Cependant, lorsque l'on travaille avec des chaînes de compilation en IDM, l'artefact de premier plan n'est plus le programme en lui-même, mais le modèle de conception. Récupérer le plus de détails possibles sur l'exécution du programme n'est pas pertinente puisque l'implémentation est cachée au maximum au concepteur. Effectuer un tri entre les informations qui proviennent de la modélisation et celles qui proviennent des détails d'implémentation automatiquement ajoutés à la génération de code devient difficile. De plus, il est évident que l'instrumentation de code, donc l'ajout de fragments de code dans un programme, introduit une modification des performances de l'application et peut même, dans certains cas, introduire des modifications de comportement. Instrumenter systématiquement la totalité du code peut donc fortement impacter l'exécution du programme et contrôler au maximum l'instrumentation apparaît donc comme une nécessité.

La programmation par aspect [74], représentant un autre paradigme de programmation, peut aussi être utilisé pour proposer une instrumentation automatique du code source d'une application [31]. En effet, la programmation par aspect permet au programmeur de définir un ensemble de règles indiquant des sections spécifiques du code où des fragments de code particuliers peuvent être ajoutés. L'utilisation de la programmation par aspect peut donc permettre d'ajouter plus ou moins automatiquement des fragments de code dans un code source existant. Cependant, comme nous l'avons déjà évoqué, l'utilisation des compilateurs IDM masque le code source de l'application au concepteur de modèles. Lui demander de définir les règles pour l'ajout de fragments de code possède peu de sens puisque celui-ci n'a aucune connaissance du code généré et décider de récolter systématiquement des informations sur l'ensemble du programme pose le mêmes problème de trie d'informations que celui énoncé précédemment.

Dans ce contexte de compilateur IDM, il faut pouvoir proposer une façon d'instrumenter le code facilement, en fonction des connaissances et des artefacts manipulés par le concepteur, à savoir, les modèles de conception. C'est donc directement sur le modèle de conception qu'il faut pouvoir travailler. Afin d'instrumenter automatiquement le code

source de l'application, le concepteur doit donc pouvoir effectuer deux décisions importantes. Il doit pouvoir choisir pour quel élément du modèle de conception il veut obtenir des informations et il doit pouvoir aussi choisir le type d'informations qu'il veut obtenir à l'exécution. L'idée est donc de fournir au concepteur un moyen d'instrumenter automatiquement le code source de l'application généré à partir de ses modèles de conception. Trois questions se posent alors :

- Comment spécifier les parties du modèle a observer ?
- Comment instrumenter correctement et automatiquement le code en fonction des informations que le concepteur veut obtenir ?
- Comment fournir au concepteur un moyen de visualiser intelligiblement les informations qu'il a demandées ?

7.2 OBSERVER UN MODÈLE, VUE GÉNÉRALE

Dans ce document, nous utilisons le terme d'*observations* de modèles pour parler de la visualisation, sur les modèles de conception, des informations liées à l'exécution d'un programme généré à partir de ces modèles. Ainsi, au chapitre précédent, la visualisation des informations liées à la performance de l'application peut être considérée comme un type d'observation de modèles. Cependant, cette observation de performance couvrait l'intégralité de l'application. Nous définissons donc un deuxième terme : *observation partielle* de modèle qui permet de visualiser des informations liées à l'exécution d'une *partie* de l'application générée à partir des modèles de conception.

Il faut noter que ce sont ces observations, spécifiées par le concepteur de modèles sur ses modèles de haut niveau, qui vont préciser quels sont les fragments de code qui vont être ajoutés au code source de l'application. Il existe donc un lien fort entre une *observation* et un *fragment de code*.

Dans ce chapitre, nous proposons un mécanisme permettant d'observer un modèle. Cette observation s'effectue en deux points clefs : la spécification des éléments de modèles à observer et l'instrumentation automatique du code source en conséquence. Cette approche repose en très grande partie sur l'utilisation de la trace locale et plus précisément de la trace réduite. La figure 72 présente notre approche d'observation partielle de modèles. Cette approche possède une structure assez similaire à celle proposée au chapitre précédent, mais donne une place plus importante à la trace, utilisée à deux reprises, et requiert autorise l'ajout d'observations sur les modèles de conception. Le flot d'exécution de notre approche est le suivant. Une fois la conception du modèle de haut niveau terminée, le concepteur peut vouloir observer certains éléments de son modèle. Il ajoute donc des *observations* sur son modèle de haut-niveau [1]. La chaîne de compilation est ensuite lancée (étape 1). Comme pour l'approche d'optimisation de modèles, des traces sont produites tout au long de la chaîne de compilation. Ces traces sont ensuite utilisées, lors de l'étape de génération de code, par le procédé d'instrumentation de code [2]. L'étape de génération de code terminée, le code source de l'application est instrumenté et embarque des fragments de code relatifs aux observations demandées par le concepteur. Le code source est ensuite compilé et exécuté (étape 2 et 3). L'exécution de l'application instrumentée crée des *log* contenant les informations relatives aux observations demandées par le concepteur (étape 4). Ici aussi, les *logs* d'observations référencent des UIDs des éléments de

1. Les observations qu'il est possible d'utiliser son détaillées en section 7.3.

2. Cette étape est détaillée en section 7.4.

modèle pour garder le lien entre les observations générées et le monde des modèles. Ces *logs* sont ensuite *parsés* pour obtenir un modèle de *logs* d'observation (étape 5) qui est ensuite remonté sur les modèles de haut niveau, en utilisant la trace (étape 6).

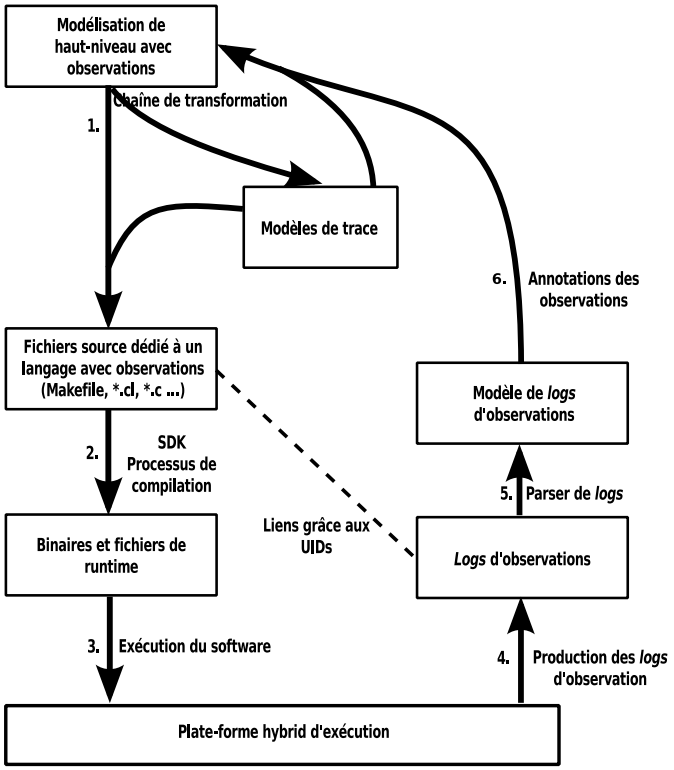


FIGURE 72: Processus d'observation de modèle

7.3 SPÉCIFIER CE QUE L'ON VEUT OBSERVER

Comme nous l'avons précisé, une de nos exigences est que les observations demandées par le concepteur doivent apparaître directement sur le modèle de conception. Dans le cadre de GASPARD2, cela revient à ajouter des observations sur un modèle UML profilé MARTE. Nous avons donc utilisé un des moyens fournis par UML pour proposer de nouveaux concepts : les profils UML et nous avons proposé un profil d'observations.

7.3.1 *Un profil* UML *pour les observations*

Le profil que nous proposons permet de spécifier sur le modèle de conception le type d'observation que le concepteur veut pouvoir réaliser.

Le profil d'observation que nous proposons est présenté en figure 73. Il possède 10 stéréotypes dont : 5 sont abstraits : *Observation*, *Basics*, *Breaking*, *UserInjection* et *Advanced* et 5 sont concrets : *Activation*, *Increment*, *Exit*, *InjectComment* et *InjectCodeFragment*.

La base du profil est le stéréotype abstrait *Observation*. Il spécialise la méta-classe UML *Element* indiquant ainsi qu'il peut se greffer sur n'importe quelle instance du méta-modèle UML. Ce stéréotype possède

possède deux *tagged values* : *placeInCode* et *reportOn*. La première *tagged value* permet de spécifier à quel endroit du code généré pour l'élément, le fragment relatif à l'observation doit être inséré. Les trois valeurs qu'elle peut prendre sont *before*, *inside* et *after*, ce qui représente l'ajout du fragment, juste *avant*, à l'*intérieur* ou juste *après* le code de l'élément à observer. Par exemple, en supposant que pour un élément *A*, une fonction *A_run()* est généré. Grâce à cette *tagged value*, le concepteur peut préciser qu'il souhaite voir son observation placée en début de la fonction (*before*), à l'intérieur de la fonction (*inside*) ou à la fin de la fonction (*after*)[3]. La seconde *tagged value* permet de contraindre la visualisation de l'observation sur le modèle de conception. En effet, lors de l'étape de retour, il est possible que plusieurs éléments soient concernés par les informations obtenues à l'exécution. Ainsi, cette *tagged vale* permet de préciser si les retours doivent être effectués : uniquement sur l'élément (*onlyMe*) ou sur tous les éléments concernés (*all*).

3. Nous reviendrons plus en détail sur le positionnement des observations dans le code en section 7.4.

Les observations de base qu'il est possible d'effectuer grâce à ce profil sont présentées et classées en 3 grandes catégories :

- ***Basics***, pour des observations basiques d'affichage de valeurs.
- ***Breaking***, pour l'ajout de structures d'arrêt de l'exécution.
- ***UserInjection***, pour l'ajout dans le code source d'informations utiles pour sa compréhension par l'utilisateur.

Seuls les observations de type *Basics* ou *Breaking* produisent des entrées dans le *log* d'observations. Finalement, ces catégories intègrent au total 5 actions :

- ***Activation***, de type *Basics*, permet de déterminer si le code relatif à l'élément stéréotypé est exécuté.
- ***Increment***, de type *Basics*, compte le nombre de fois que le code relatif à l'élément stéréotypé est exécuté.
- ***Exit***, de type *Breaking*, interrompt l'exécution du programme.
- ***InjectComment***, de type *UserInjection* ,insère la chaîne *value* comme commentaire dans le code source de l'application.
- ***InjectCodeFragement***, de type *UserInjection*, insère un fragment de code (*code*) dans le code source de l'application.

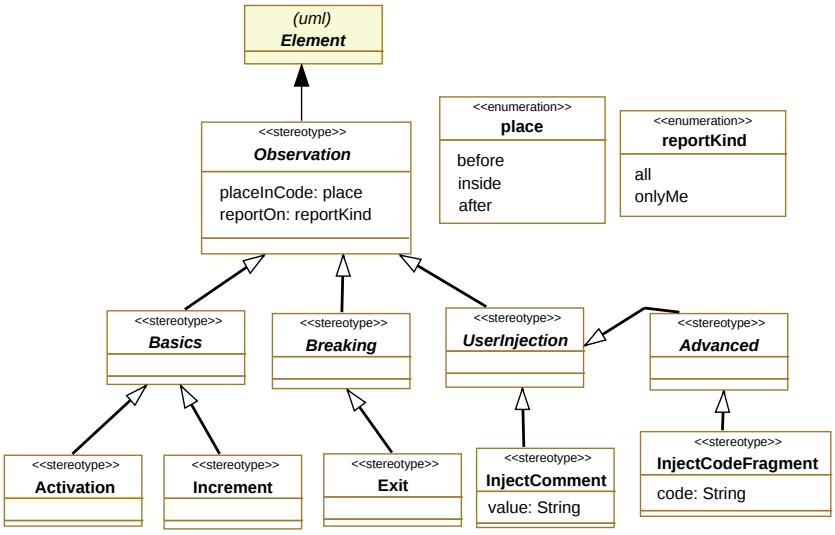


FIGURE 73: Profil d'observation de modèle

La demande d'observation d'un élément du modèle de conception se fait alors simplement en stéréotypant l'élément à observer avec un des stéréotypes du profil d'observations. Ce profil que nous avons fourni est volontairement simple et léger pour ne pas imposer une méthodologie complexe au concepteur lorsqu'il veut observer un modèle.

7.3.2 *Vers un modèle d'observation indépendant*

Afin d'être réutilisable lors de l'étape de génération et d'instrumentation du code, les observations placées sur le modèle de conception doivent être portées jusqu'au bout de la chaîne de compilation. Pour réaliser cette tâche, deux solutions sont envisageables. La première consiste à embarquer les informations d'observation dans les différents modèles d'un bout à l'autre de la chaîne de compilation. Cette solution oblige les différents modèles intermédiaires utilisés par le compilateur à embarquer des concepts qui ne sont pas relatifs au domaine métier pour lequel le modèle de conception est créé. Cela impose donc la modification des différentes transformations de la chaîne[4] pour faire suivre les informations relatives aux observations d'un bout à l'autre de la chaîne. Cette solution est donc assez lourde de conséquence pour la chaîne de compilation existante. De plus, si le concepteur veut effectuer d'autres observations, pour pouvoir propager les nouvelles observations, il faudrait rejouer l'intégralité de la chaîne de compilation.

4. Sauf dans le cas des transformations localisées

La seconde solution consiste à produire un modèle d'observation indépendant des modèles utilisés par la chaîne de compilation et qui sera réutilisé uniquement à la dernière étape de génération et instrumentation du code source de l'application. C'est cette solution que nous retenons. Les modifications sur la chaîne de compilation déjà existante sont mineures, voir inexistantes, puisque le modèle d'observation produit est indépendant. De plus, cette solution permet une plus grande flexibilité lorsque des changements d'observations sont demandés par le concepteur. En effet, pour pouvoir instrumenter le code source de l'application en fonction des nouvelles demandes de l'utilisateur, il suffit de générer à nouveau le modèle d'observation et de relancer la génération et l'instrumentation du code. Ainsi, le modèle métier ne change pas, seule les demandes d'observations sont différentes.

Construction du méta-modèle d'observations

Produire un modèle d'observation indépendant implique d'utiliser un méta-modèle d'observation. Ce méta-modèle d'observation est montré en figure 74 et possède une structure très proche du profil UML que nous avons présenté en figure 73. On y retrouve les concepts principaux déjà présents dans le profil, avec la même hiérarchie. Les concepts ajoutés sont le concept d'*ObservationModel*, qui représente la racine des modèles d'observations et le concept d'*EObject* qui représente un élément auquel est lié une observation par le biais de la référence *annotate*.

L'utilisation d'un concept *EObject* pour pouvoir lier une observation à un élément de modèle est indispensable. En effet, c'est grâce à cette référence *annotate* vers le concept de *EObject* qu'une observation posée sur un élément du modèle de conception va pouvoir être liée à ses *descendants*[5] dans le dernier modèle avant la génération de code. De cette façon, lors de la phase de génération et d'instrumentation du code,

5. *C.f*, chapitre 3

il sera possible de déterminer quels sont les éléments que le concepteur veut observer et quel type d'informations il veut récupérer à l'exécution.

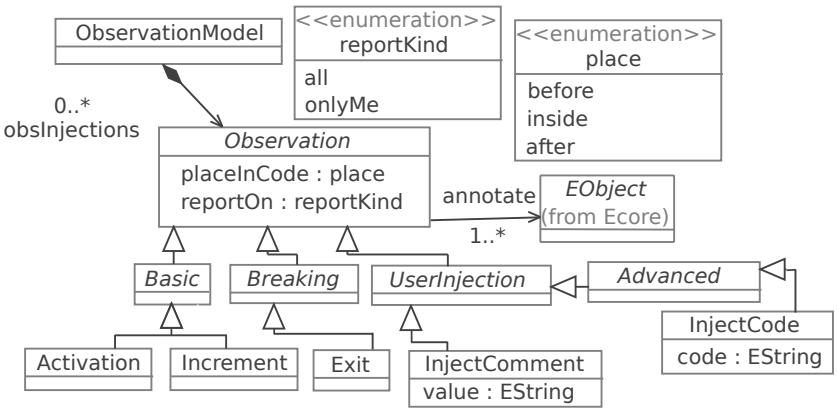


FIGURE 74: Méta-modèle d'observation

Aperçu de la gestion du modèle d'observation

Nous illustrons en figure 75 la façon dont le modèle d'observation est produit. Dans le modèle de conception, l'idée est de lier les observations non plus uniquement aux éléments du modèle de haut niveau sur lesquels le concepteur a placé ces observations [6], mais aussi à tous les éléments du dernier modèle de la chaîne qui ont été générés à partir de ces éléments à observer. Pour ce faire, nous utilisons la trace produite pendant la chaîne de compilation.

6. Sous forme de stéréotypes.

Plus précisément, pour produire le modèle d'observation, chaque élément du modèle de conception stéréotypé par une observation est transformé en concept dans le modèle d'observation. Puis, la trace est interrogée pour déterminer les éléments du dernier modèle qui ont été produit à partir de chacun de ces éléments. Chaque concept du modèle d'observation est ensuite lié aux éléments récupérés par la trace.

7.4 INSTRUMENTATION DU CODE

Comme nous l'avons précisé en section 7.1, pour pouvoir récupérer des informations pendant l'exécution d'un programme, cela implique l'ajout de fragments de code spécifique dans le code source de l'application. Cette insertion dans le code source n'est pas faite pas le concepteur puisqu'il n'a pas de connaissance de la structure du code source produit et il n'a, potentiellement, pas les connaissances nécessaires dans le langage utilisé pour pouvoir instrumenter manuellement le code source de l'application. Pour pouvoir instrumenter le code à la guise du concepteur, nous avons mis en place un mécanisme d'insertion de fragment de code à des places spécifiques du code en fonction de l'élément à observer.

Les étapes de mise en place de l'instrumentation du code que nous présentons dans cette partie sont effectuées par le concepteur de la chaîne et sont effectuées une seule fois pour chaque chaîne à instrumenter.

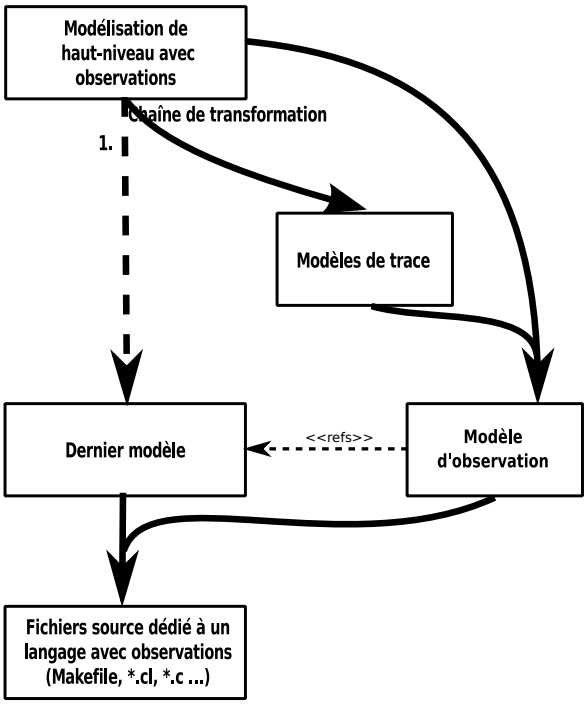


FIGURE 75: Détail de la génération d'observations

7.4.1 *Quelle place pour une observation ?*

Pour instrumenter correctement le code source d'une application et, plus précisément, le morceau de code relatif à l'élément que l'on veut observer, il faut décider d'une place à laquelle il est correct d'ajouter du code. En effet, nous montrons ici que la place à laquelle est insérée un morceau de code est importante car elle peut influer sur l'intégrité du code source de l'application.

Dans le contexte des transformations de modèles vers texte, un *template*[7] génère un bloc de code d'éventuellement plusieurs lignes. Il est donc absurde de supposer l'insertion d'un fragment de code toujours à la même place. En effet, si l'insertion d'un fragment de code se fait, par exemple, toujours avant le corps du *template*, cela peut mener à insérer des erreurs dans le code source.

7. *Cf.* chapitre 1.

```
1  <%script type="A" name="codeForA"%>
2  <%codeForHeader%>
3  {
4    // variables
5    ...
6    <%codeForBody%>
7  }
```

Listing 7.1: Exemple de template

Pour illustrer nos propos, le listing 7.1 présente un court *template* : *codeForA* écrit en ACCELEO sur lequel nous allons voir les problèmes que peuvent engendrer un ajout de code non contrôlé sur un *template* particulier. Le *template* présenté est celui d'une fonction générant du langage C. La ligne 1 présente la signature du *template*. Il est précisé qu'il s'applique sur les éléments de type *A* et qu'il porte le nom *codeForA*. La seconde ligne appelle directement un autre *template* : *codeForHeader*,

puis génère une accolade en ligne 3. En ligne 4 un commentaire est ajouté, puis le *template codeForBody* produisant une partie du corps de la fonction est appelé. Finalement, une accolade fermante est générée à la ligne 7. Dans un tel *template*, si l'on ajoute un fragment de code :

- **Avant** le code du *template*, cela produirait une insertion de code hors du corps de la fonction, ce qui pourrait mener à une erreur dans le code produit.
- **Au milieu** du code du *template* (entre les accolades). Il est impossible de savoir où placer le code automatiquement. Le faire aléatoirement pourrait poser autant de problèmes que s'il est placé avant le code du *template*.
- **Après** le code du *template*, c'est le même cas que pour le placement du code avant le *template*.

Finalement, de manière générale, pour insérer correctement et sans introduire d'erreur un fragment de code dans ce *template* il aurait fallu insérer le fragment code à l'intérieur des deux accolades. Exactement, si l'on avait voulu insérer le code :

- **Avant**, sous entendu, en début du *template*, il aurait fallu placer le fragment de code à partir de la ligne 4.
- **Au milieu**, tout dépend alors du corps du *template*, mais il faudrait convenir d'une place exacte où le fragment de code peut être ajouté.
- **Après**, sous entendu, en fin du *template*, il aurait donc fallu placer le fragment de code juste avant la ligne 7.

On remarque alors que le choix de la place à laquelle l'observation peut être insérée est primordiale pour minimiser le risque d'insérer des erreurs dans le code. Il faut donc choisir une place attitrée dans un *template* où le fragment de code pourra être ajouté par le mécanisme d'instrumentation du code.

Le choix du développeur et la liberté du concepteur

Comme nous venons de le voir, si un fragment de code doit être inséré dans un *template*, il faut donc lui préciser à quels endroits il a le droit d'y apparaître. Une solution envisageable serait de laisser le choix au concepteur du modèle de préciser à quel ligne un fragment devrait être ajouté. Cependant, vouloir laisser un tel degré de liberté au concepteur de modèles est illusoire. En effet, encore une fois, le concepteur n'a pas connaissance des *templates* utilisés par la transformation générant l'application. Il ne peut donc pas indiquer la ligne à laquelle il lui semble juste d'ajouter le fragment de code. En revanche, le développeur de la chaîne possède les connaissances nécessaires pour indiquer sans risque où un fragment de code a le droit d'être inséré.

Pour indiquer ces places, nous fournissons un système de points d'entrée permettant au développeur de la chaîne de compilation de préciser dans les *templates*, les places correctes auxquelles un fragment de code pourrait être ajouté. Le listing 7.2 présente à nouveau le code du *template* présenté au listing 7.1 avec un exemple de placement de point d'entrée.

```
1  <%script type="A" name="codeForA"%>
2  <%codeForHeader%>
3  {
4    -> POINT D'ENTREE AVANT
5    // variables
6    ...
```

```
7     -> POINT D'ENTREE AU MILiEU
8     <%codeForBody%>
9     -> POINT D'ENTREE APRES }
```

Listing 7.2: Exemple de template avec les points d'entrée

Trois points d'entrée sont précisés dans le *template*. Ils indiquent que les lignes auxquelles il est possible d'ajouter un morceau de code sans modifier l'intégrité du code sont : la ligne 4, la ligne 7 et la ligne 9. Ces trois points d'entrées sont donc les places potentielles auxquelles les fragments de code peuvent être insérés. Évidemment, certains fragments relatif à certaines observations en particulier demandent un traitement spécial et nécessitent d'être insérés à d'autres endroits. Afin de traiter avec ces fragments de code particulier, nous avons introduit un mécanisme de points d'entrée spécifique que nous détaillons en section 7.4.3.

Afin de laisser la liberté au concepteur de modèle de choisir à quel place le fragment de code doit être ajouté pendant la phase d'instrumentation du code, la *tagged value placeInCode* du profil d'observations précise le point d'entrée à utiliser. Par exemple, le concepteur peut demander le placement de l'observation *Exit before*, indiquant que l'application interrompt son exécution avant d'avoir exécuté le code de l'élément stéréotypé. Cette même observation peut aussi être placée *after*, indiquant ici que le programme est interrompu après avoir exécuté le code de l'élément stéréotypé.

Un service d'insertion pour les points d'entrée

Les points d'entrée que nous fournissons font partie d'un service d'insertion. En fonction de ce que nous avons précisé ci-dessus, nous avons proposé trois points d'entrée : *insertBefore*, *insertInside* et *insertAfter*. Lors de la phase de génération et d'instrumentation de code, ces trois points d'entrée travaillent directement avec le modèle d'observation généré. Lorsqu'un *template* est exécuté pour un élément, celui-ci est parcouru de manière séquentielle en partant de la première ligne du *template* jusqu'à la dernière. Lorsque l'appel à un point d'entrée est rencontré, celui-ci cherche dans le modèle d'observations celles qui sont liées à l'élément en cours d'évaluation. Pour toutes les observations trouvées, seules les observations possédant leur *tagged value* : *placeInCode* en correspondance avec le point d'entrée en cours d'évaluation sont conservées [8] et les fragments de code associés à l'observation sont ajoutés. Une fois l'évaluation du point d'entrée terminé, l'évaluation du *template* continue et le procédé est répété à chaque point d'entrée rencontré. De cette façon, nous pouvons voir qu'il est possible de demander plusieurs observations sur un même élément puisque les points d'entrée sont tous évalués lors de l'évaluation du *template*.

8. Par exemple, si le point d'entrée en cours d'évaluation est le *insertBefore* et que l'observation trouvée possède sa *tagged value placeInCode* avec la valeur *inside*, cette observation n'est pas prise en compte.

7.4.2 *Liaison d'un point d'entrée à un fragment de code*

Les points d'entrée permettent à la fois de définir une place à laquelle un fragment de code peut être inséré et de vérifier, à l'exécution de la génération de code, si une observation est liée à un élément. Cependant, les points d'entrée ne fournissent pas directement le fragment de code relatif à une observation qu'il faut insérer dans le code.

Afin de définir le lien qu'il existe entre une observation et un fragment de code, nous fournissons un mécanisme de *mapping* reposant sur :

- plusieurs fichiers de *template*,
- un fichier de *mapping* permettant de lier un des fichiers de *template* à une observation.

Templates d'observations

Les *templates* d'observations correspondent aux fragments de code qui vont servir à l'instrumentation du code source de l'application. Chacun d'eux représente le fragment de code d'une observation en particulier et sont dédiés à un langage spécifique. Par exemple, si le langage de programmation visé par la transformation de modèles vers texte est le langage C, les *templates* d'observation seront évidemment codés en C.

Mapping de templates

Ce fichier est le plus important, c'est lui qui permet de lier un fragment de code, donc un *template*, à une observation en particulier. Plus précisément, ce fichier est composé d'une fonction qui regarde le type d'une observation qui lui est passée en paramètre et qui appelle le *template* correspondant.

La figure 76 illustre sur un exemple la séquence des appels qui sont effectués lorsqu'un point d'entrée est évalué. Sur cet exemple, le *template* en cours d'évaluation est le *template A*, il se charge de générer le code correspondant à un élément de type *A* sur lequel une observation *Exit* avec le placement à l'intérieur du *template* (*inside*) a été demandé.

La génération de code par le *template A* commence (noté 1 sur la figure). Un premier point d'entrée *insertBefore* est atteint, le *template A* fait donc appel au service de point d'entrée pour déterminer si une observation est liée à l'élément dont le *template A* est en train de générer le code. Le service termine directement car l'observation *Exit* trouvée n'a pas sa *tagged value* `placeInCode` égale à *before*. L'évaluation du *template A* continue donc (noté 2 sur la figure), puis rencontre un autre point d'entrée *insertInside*. Le service de point d'entrée est donc appelé. Celui-ci trouve, une nouvelle fois, une observation *Exit* liée à l'élément en cours d'évaluation par le *template A*. De plus, le placement à bien été demandé à l'intérieur (*inside*) du *template*. Afin d'ajouter le fragment de code associé, le service de point d'entrée appelle donc le service de *Mapping* pour déterminer quel fragment de code générer. Celui-ci associe l'observation *Exit* récupérée au *template Exit*. Une fois que le *template Exit* est évalué, il retourne au service de *Mapping*, qui retourne à son tour au service de point d'entrée qui retourne au *template A*. L'exécution du *template A* reprend donc son cours (noté 3 sur la figure). Finalement, un dernier point d'entrée *insertAfter* est atteint. L'appel au service de point d'entrée est effectué, mais comme pour l'appel *insertBefore*, le service de point d'entrée retourne directement car l'observation trouvée n'est pas demandé pour la place *after*.

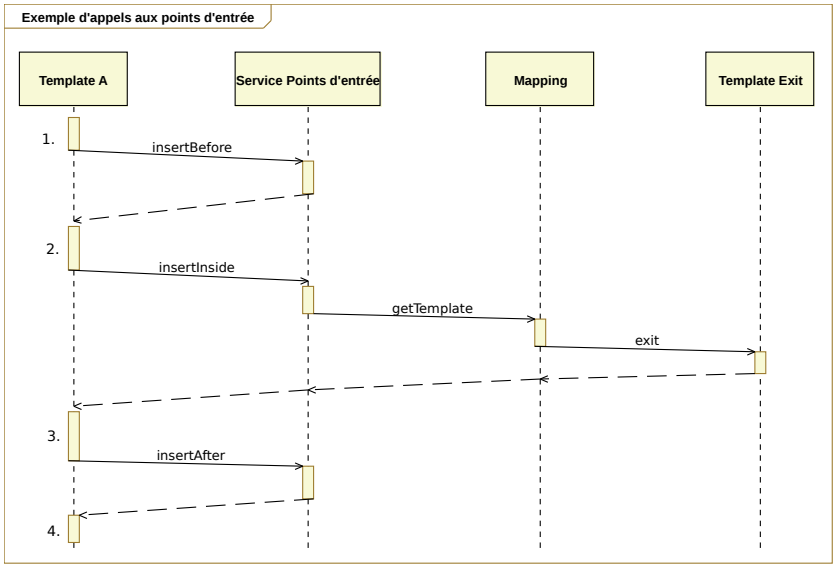


FIGURE 76: Exemple d'appels aux points d'entrée pour un *template A*

Ajout d'observations

Le profil UML d'observations et le méta-modèle d'observations, que nous avons présentés précédemment proposent un certain nombre d'observations de base. Cependant, pour des utilisations ou des langages de programmation spécifiques, de nouvelles observations peuvent être utiles. Dans ce cas, il est nécessaire de créer le *template* générant le fragment de code relatif à l'observation ajoutée. Pour rendre l'ajout de l'observation effective et manipulable par le concepteur de modèle, certaines évolutions sont nécessaires sur :

- **le profil UML** ; pour pouvoir proposer l'observation au concepteur de modèles,
- **le méta-modèle d'observation** ; ce méta-modèle doit toujours être « synchronisé » avec le profil UML pour pouvoir générer le modèle d'observations,
- **le fichier de *Mapping*** ; pour lier le *template* de génération du fragment de code à l'observation créée.

7.4.3 *Contrôle de la réplication des observations*

En regardant de plus près les appels qui sont passés par les *templates* et les vérifications effectuées par le service de point d'entrée, il est possible de remarquer que l'ajout des observations dans le code de l'application dépend entièrement du modèle d'observations créé. Notamment des liaisons existantes entre une observation et les éléments du modèle utilisé pour la génération de code.

En effet, lorsqu'un point d'entrée est appelé, celui-ci cherche les éléments du modèle d'observations qui sont liées à l'élément dont on génère le code. Cela signifie que si une observation est liée à plusieurs éléments et si chacun des *templates* associés à ces éléments intègre des points d'entrée, la génération du code de chacun d'eux va intégrer le fragment de code de l'observation. Nous nous retrouvons donc avec un fragment de code qui est répliqué lorsqu'une observation est liée à plusieurs éléments. Il faut donc fournir un moyen de limiter la

réplication d'un même fragment de code à plusieurs endroits dans le code de l'application.

La rencontre de liens de trace $1 - n$

Les transformations que nous considérons dans les chaînes de compilation sont des transformations pouvant créer n éléments dans les modèles de sortie pour 1 élément des modèles d'entrée. Les traces que nous générons sont fidèles à la transformation exécutée et il est naturel d'y rencontrer des liens de trace $1 - n$. La trace réduite, construite à partir de ces traces peut donc elle aussi contenir des liens de trace $1 - n$ entre le modèle de conception et les derniers modèles générés pas la chaîne. Comme nous l'avons vu précédemment, la liaison d'une observation à un élément dépend de la trace générée. En effet, lors de la génération du modèle d'observation, la trace est interrogée pour déterminer les *descendants* des éléments portant un stéréotype observation. Ces descendants sont ensuite liés à l'observation créée dans le modèle d'observation.

Étant donné que notre trace est capable de posséder des liens de traçabilité $1 - n$, il est possible de trouver plusieurs descendants à un élément dans le modèle d'entrée, ce qui entraîne, potentiellement, l'insertion d'un même fragment de code à plusieurs endroits du code source, même si on aimerait le voir apparaître à un seul endroit. Pour contrôler cette réplication des fragments de code, nous avons mis à disposition des concepteurs de la chaîne de compilation un moyen de définir et d'utiliser des points d'entrée spécifique.

L'utilisation de points d'entrée spécifique

La solution que nous avons fournie pour éviter qu'une observation soit répliquée à plusieurs endroits dans le code repose sur l'utilisation d'un point d'entrée spécifique. Jusqu'à présent, trois points d'entrée étaient utilisés : *insertBefore*, *insertInside* et *insertAfter*. Ces points d'entrée sont génériques car ils peuvent être utilisés pour plusieurs observations et c'est leur appel au service de *Mapping* qui détermine le bon fragment de code à générer. Au contraire, un point d'entrée spécifique ne fonctionne que pour un type d'observation en particulier.

Ainsi, ces points d'entrée ne sont placés qu'à certains endroits du code et son dédiés à une observation. Pour éviter toute confusion, ils portent un nom différent des points d'entrée génériques, mais ils fonctionnent presque de la même manière, à l'exception près qu'ils n'effectue l'insertion d'un fragment que pour un type d'observation en particulier. Il est à noter que l'utilisation systématiques de points d'entrée spécifique ne pose pas de problèmes particulier, mais ne sont pas recommandé, car elle entraîne un effort de développement plus important de la part des concepteurs de la chaîne puisqu'ils devraient placer les points d'entrée spécifique pour chaque observation et dans *template* de la transformation de modèles vers texte. Le concepteur de la chaîne doit donc utiliser les points d'entrée spécifique uniquement lorsqu'une observation ne peut être demandée que sur un type d'élément en particulier.

7.4.4 *Éviter de modifier la transformation de génération de code*

Comme nous l'avons montré, la solution que nous proposons nécessite l'exécution d'une transformation de modèle vers texte légèrement différente de celle utilisée initialement. Afin d'éviter la modification de cette transformation de génération de code, nous avons utilisé un mécanisme reposant sur une notion d'héritage de *template*.

Généralement, les langages de transformation de modèles vers texte utilisant des *templates* permettent de créer un héritage entre plusieurs transformations [132][9]. Ce mécanisme associé à de la surcharge entre les *templates*[10] nous permet de modifier seulement certains *templates* de la transformation originale pour en créer une nouvelle. Ainsi, la nouvelle transformation est composée de deux parties : la transformation originale ainsi que la transformation modifié avec les *templates* surchargé qui ne prennent en compte que les parties dans lesquelles il est possible d'insérer un fragment de code.

9. C'est notamment le cas d'ACCELEO, utilisé pour la génération de code dans l'environnement GASPARD2.

10. Exactement à l'image de ceux qu'il est possible d'utiliser dans des langages objets traditionnels.

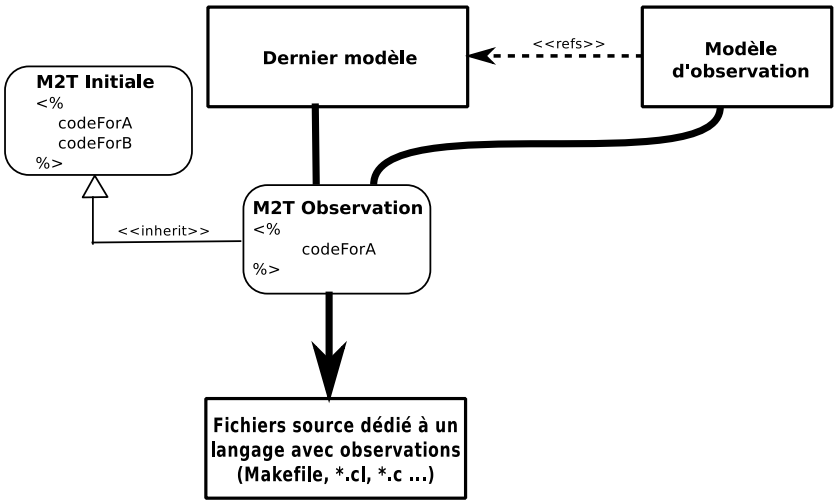


FIGURE 77: Héritage de transformations et surcharge de *template*

La figure 77 montre comment l'héritage et la surcharge des *templates* sont utilisés dans la fin de la chaîne de compilation. Comme nous l'avons vu précédemment, pour effectuer la génération et l'instrumentation du code source de l'application modélisée, nous utilisons deux modèles. Le modèle issu de la compilation de la chaîne (noté *Dernier modèle* sur la figure) et le modèle d'observation. Ces deux modèles sont consommés par une transformation *M2T Observation* qui hérite de la transformation initiale *M2T Initiale*. La transformation initiale propose deux *templates* : *codeForA* et *codeForB*, générant du code pour les éléments de type *A* et *B* respectivement. La transformation servant à instrumenter le code *M2T Observation*, possède un seul *template codeForA*. En supposant que la transformation *M2T Observation* génère le code pour un élément *A*, celle-ci utilise son propre *template codeForA* qui, par le jeu de l'héritage, surcharge le *template codeForA* de la transformation initiale. En supposant maintenant qu'un élément de type *B* doive être traité par la transformation *M2T Observation*, celle-ci ne possédant pas le *template codeForB*, elle utilise automatiquement le *template codeForB* de la transformation *M2T Initiale*. La transformation initiale n'est donc

pas modifiée, seuls les *templates* surchargés prennent en compte les observations.

7.4.5 *Le format des observations*

Dans les sections précédentes, nous avons fourni un moyen d'instrumenter automatiquement le code source d'une application depuis des demandes d'observations spécifiées dans les modèles de haut niveau. Plus précisément, nous avons proposé :

- pour le concepteur de modèles, un moyen de demander l'observation de certains éléments de modèles,
- pour le développeur de la chaîne de compilation, le moyen de préparer sa chaîne pour pouvoir activer le support de l'instrumentation automatique du code.

Cela répond à deux des questions que nous nous étions posées en section 7.1. La dernière question qui reste en suspend et celle relative à la visualisation des informations récupérées lors de l'exécution. Pour effectuer cette tâche, nous allons réutiliser une partie des travaux que nous avons présentés au chapitre précédent.

Pour mémoire, dans le chapitre précédent, afin de remonter les informations obtenues à l'exécution, nous utilisions un mécanisme à base d'UIDs pour pouvoir reconnecter les informations obtenues à l'exécution de l'application au modèles de conception. Pour pouvoir visualiser les informations obtenues à l'exécution de l'application sur les modèles de conception, il faut que des UIDs apparaissent dans le *log* d'observation produit. Les fragments de code liés à une observation qui sont insérés dans le code de l'application doivent donc aussi pouvoir générer des entrées dans le *log* d'observation contenant des UIDs. Nous proposons un format d'entrée de *log* d'observation reposant sur quatre champs :

1. un champ UID,
2. un champ *kind*,
3. un champ *data*,
4. un champ *timestamp*.

Le premier champ permet donc de garder le lien avec le monde des modèles en propageant l'UID utilisé comme nous l'avons présenté dans le chapitre précédent. Le second champ conserve le type d'observation qui est reportée[11] alors que le troisième conserve la valeur de l'observation. Cette valeur est donnée sous forme de chaîne de caractère et peut représenter une valeur entière, un flottant, une chaîne de caractère ou toutes autre information. Finalement, le dernier champ garde une valeur faisant office de tampon temporel pour déterminer l'ordre dans lequel les observations sont apparues durant l'exécution du programme.

11. Par exemple, *Activation* ou encore *Exit*.

À titre d'exemple, le listing 7.3 présente un *template* ACCELEO générant un fragment de code C pour l'observation *Activation*.

```
1  <%script type="EObject" name="activation"%>
2  { log(logger,"<%self().getUUID()%>","ACTIV","<%self().name%>");
3  }
```

Listing 7.3: Template *Activation* écrit pour générer du C

Ce *template* est appelé sur l'élément dont le *template* général est en train de généré le code. Lorsque ce *template activation* est appelé, il commence par produire une accolade gauche (ligne 2) avant de générer un appel à une fonction *log* que nous avons fournie. Cette fonction prend le *logger* comme premier paramètre, une variable, représentant le fichier de *log* dans lequel le programme va écrire lors de son exécution. Lors de l'exécution de l'application, chaque appel à la fonction *log* produit une entrée sous le format énoncé précédemment : à savoir l'UID de l'élément à traiter (second paramètre), le type d'observation (troisième paramètre) et finalement la valeur. Ici, pour l'observation d'*Activation*, c'est le nom de l'élément activé qui est considéré comme valeur du champ *data*. Le *timestamp*, quant à lui, est directement gérée par la fonction de *log* et le *logger*. Finalement, le *template* génère l'accolade fermante qu'il avait ouverte en ligne 4.

En supposant que ce *template* soit appelé à partir d'un élément dont l'UID est *1234abcd* et portant le nom *A*, la ligne suivante serait générée dans le code source de l'application :

```
log(logger,"1234abcd","ACTIV","A");
```

Lors de l'exécution de l'application, quand le code relatif au code de l'élément *A* sera exécuté, la ligne suivante sera générée dans le *log* d'observation :

```
<1234abcd, ACTIV, A, 1>
```

Dans cette entrée, on retrouve l'UID en première position, le type de l'observation effectuée en seconde position, puis la donnée en troisième position et finalement le *timestamp* qui est généré automatiquement à chaque appel de la fonction *log* [12].

12. Comme le *timestamp* est égal à 1, cela veut dire ici que cette entrée du *log* d'observation est la première générée par l'exécution de l'application.

7.5 RÉCUPÉRATION DES INFORMATIONS ET REMONTÉE

Comme pour la remontée d'information de *profiling*, le *log* contenant les résultats des observations produites à l'exécution est *parsé* pour obtenir un modèle d'*Advice*. Nous avons utilisé un modèle d'*Advice* car celui-ci peut facilement prendre en compte les informations présentes dans le *log* d'observation.

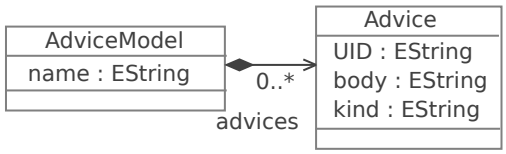


FIGURE 78: Méta-modèle de *logs* de *profiling* et de conseils

La figure 78 présente, à nouveau, le méta-modèle d'*advice* utilisé [13]. Le concept *Advice* est utilisé pour représenter une entrée du *log* d'observation. Pour une entrée, l'attribut *UID* conserve l'UID de l'élément sur lequel l'observation a été effectuée, l'attribut *kind* conserve le type d'observation et l'attribut *body* sert à récupérer la valeur observée ainsi que la valeur du marqueur temporel.

13. *C.f.*figure 56, page 102.

Une fois que le modèle d'*advice* est produit, chacun des *Advices* présents dans le modèle est remonté sur les modèles de conception, comme pour les informations de *profiling*. Cependant, nous utilisons un

algorithme de remontée légèrement différent de celui présenté pour la remontée des informations de *profiling* pour pouvoir prendre en compte le souhait exprimé pour les retours des informations par concepteur de modèle [14]. En effet, cette fois, plutôt que de passer en paramètre de l'algorithme les types sur lesquels les annotations doivent être ajoutées, c'est la *tagged value reportOn* des éléments sur lesquels les informations doivent être retournée qui détermine exactement sur quels éléments l'information doit être ajoutée. De cette façon, c'est directement le concepteur qui décide qui portera l'information sur le modèle d'entrée. Si le concepteur place la valeur de la *tagged value* de son observation à *all*, l'information sera reportée sur tous les éléments impliqué dans la génération de l'information à reporter [15], si elle est placée à *onlyMe*, le résultat de l'observation sera reportée uniquement sur l'élément stéréotypé.

14. Grâce à l'utilisation de la *tagged value reportOn*.

15. Comme pour notre approche d'optimisation de modèles.

7.6 CONCLUSIONS

Dans ce chapitre, nous avons vu comment il était possible de récupérer des informations liées à l'exécution d'une application modélisée et de les retourner sur les modèles de conception. De cette manière, il est possible de bénéficier d'informations liées au comportement de l'application modélisée, directement sur les modèles de conception. Ces informations sont récupérées à la demande du concepteur de modèles. Notre proposition repose sur trois artefacts principaux : un profil UML d'observation, la traçabilité et un mécanisme d'instrumentation de code permettant d'insérer des fragment de code source.

Le profil UML permet au concepteur de modèles de choisir les éléments qu'il veut observer et le type d'observation qui doit être effectuée à l'exécution de l'application modélisée. Ce profil UML est extensible pour pouvoir prendre en compte les spécificités d'une plateforme donnée et ainsi pouvoir ajouter des observations dédiées à un paradigme de programmation.

Avec l'utilisation de la trace et du mécanisme d'insertion, basé sur une hiérarchie de *template*, les fragments de code associés aux observations sont insérées dans le code source de l'application au bon endroit. Les fragments de code d'observations insérées permettent de générer, lors de l'exécution de l'application, un *log* d'observation contenant les résultats des observations. Les informations contenues dans le *log* d'observations sont ensuite remontées sur le modèle de départ grâce au mécanisme de remontée d'informations que nous avons présenté au chapitre précédent.

Les observations proposées dans ce chapitre sont surtout liées à une activité de récolte d'information sur le flot d'exécution de l'application. Cependant, lors des phases de prototypage, nous avons remarqué que celui-ci peut convenir, dans certaines mesures, pour une activité de *profiling* et surtout dans une activité de compréhension de l'application. En effet, les observations posées dans le modèle permettent de représenter l'exécution de l'application générée directement sur les modèles de conception. Ces observations peuvent donc faciliter la compréhension des modèles de conception lorsque ceux-ci sont complexes.

Nous considérons que les travaux présentés dans cette partie propose donc de fournir un premier pas vers un *debug* de modèles de conception compte tenus de leurs comportements une fois le code généré. Ce sont donc les comportements de l'application qui sont corrigés par

rapports aux erreurs ou aux problèmes de performances impliquées par une mauvaise conception des modèles. Dans le chapitre suivant, nous illustrons sur un exemple notre approche d'observation de modèles : de la modification de la chaîne de compilation pour activer le support des observations à l'observation d'un modèle.

8

DEBUGGING DE MODÈLES DANS LE CONTEXTE DE GASPARD

Dans le chapitre précédent, nous avons présenté une approche permettant d'observer l'exécution de certains éléments. Nous avons défini différentes observations ainsi qu'un moyen de préciser un ensemble d'éléments du modèle de conception à observer. Nous allons dans ce chapitre, illustrer notre approche sur une chaîne de compilation IDM de GASPARD2.

Ce chapitre s'articule autour de deux points de vue : les actions effectuées préalablement par le développeur pour fournir le support des observations et les actions effectuées par le concepteur de modèle pour se servir de ce support afin d'observer ses modèles.

Plus précisément, en section 8.1, nous présentons l'application sur laquelle nous nous appuyons pour ce cas d'étude. En section 8.2, nous détaillons la chaîne de compilation et en section 8.3 les modifications qui y sont apportées par le développeur pour y intégrer le support des observations. En section 8.4, nous montrons comment nous utilisons notre mécanisme d'observation sur l'application présentée avant de conclure en section 8.5.

8.1 L'APPLICATION DE MULTIPLICATION DE MATRICES

Comme support à l'illustration de notre approche d'observation de modèles, nous allons utiliser la modélisation de l'application de multiplication de matrices. La multiplication de matrices est très largement utilisée dans les calculs d'approximation et représente un exemple

simple sur lequel nous allons pouvoir détailler et illustrer notre approche. L'application de multiplication de matrices, lorsqu'elle est lancée sur des matrices possédant des tailles importantes, demande des temps de calculs conséquents. Afin de réduire les temps de calculs, la multiplication de matrices est très souvent parallélisée pour tirer pleinement avantage de l'architecture du processeur sur lequel elle est exécutée. La parallélisation d'une telle application n'est pas spécialement complexe, mais il n'est pas rare que le concepteur de modèles y introduise involontairement des erreurs.

Comme aide-mémoire, la figure 79 représente visuellement l'application de multiplication de 2 matrices A et B. Le résultat de la multiplication est une matrice C dont chacun des éléments est calculée en fonction d'une ligne de A et d'une colonne de B. Exactement, une valeur de C située à la ligne $ligne$ et à la colonne col est calculé selon la formule suivante : $C_{ligne,col} = \sum_{i=0}^{n-1} A_{ligne,i}.B_{i,col}$. Il faut noter que pour que la multiplication de matrices soit possible, il faut que le nombre de lignes de la première matrice soit égal au nombre de colonnes de la seconde matrice. En d'autres termes, il faut que $A.nbLignes = B.nbCols$.

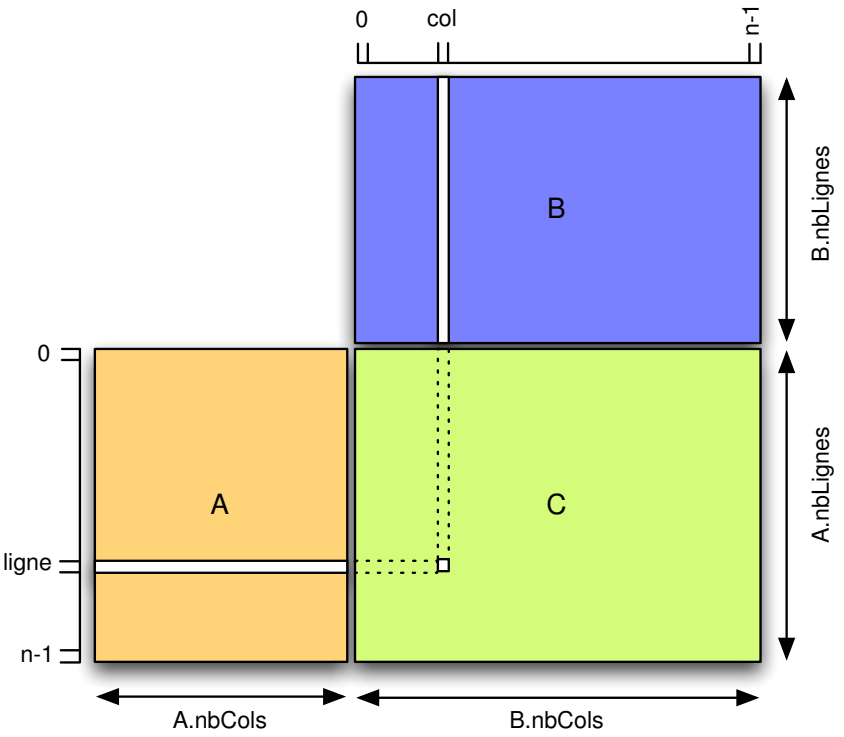


FIGURE 79: Principe de la multiplication de matrices

L'application de la multiplication de matrices est modélisée en figure 80 en UML profilé MARTE dans l'environnement GASPARD2. Cette figure représente uniquement le composant responsable de la multiplication de la matrice, mais ne montre pas la structure de l'application générale. Afin de faire les tests sur la correction de l'application, il est possible de se contenter de matrices de petites tailles. Nous avons alors modélisé la multiplication d'une matrice de taille $\{3, 4\}$ par une matrice de taille $\{4, 2\}$. Sur le modèle, il est indiqué que le composant principal *MultTask* prend 2 matrices sur ces ports d'entrée *A* et *B*. Les *shapes* associées à ces ports indiquent que la matrice attendue par le

port d'entrée *A* est de taille $\{3,4\}$ et que la matrice attendue sur le port d'entrée *B* est de taille $\{4,2\}$. Ces deux matrices sont ensuite envoyées par groupe de 4 éléments [1] tel qu'il est indiqué par la *shape* des ports d'entrée *MA_ROW* et *MB_COL* du sous composant *m : Multiply*. Ce sous composant représente l'équation que nous avons énoncé plus haut, à savoir : $C_{ligne,col} = \sum_{i=0}^{A.largeur-1} A_{ligne,i}.B_{i,col}$. Il produit donc en sortie une seule valeur (comme il est précisé sur la *shape* du port de sortie *MC_PT* du sous composant), qui est rangé à la bonne place dans le tableau de sortie de taille $\{4,2\}$ et fournit en sortie du composant par le biais du port de sortie *C* du composant *MultTask*. Afin de parcourir toutes les lignes de la matrice *A* et toutes les colonnes de la matrice *B*, le sous composant *m : Multiply* possède un *shape* de $\{3,2\}$.

1. Une ligne pour la matrice prise en entrée sur *A* et une colonne pour la matrice prise en entrée sur *B*.

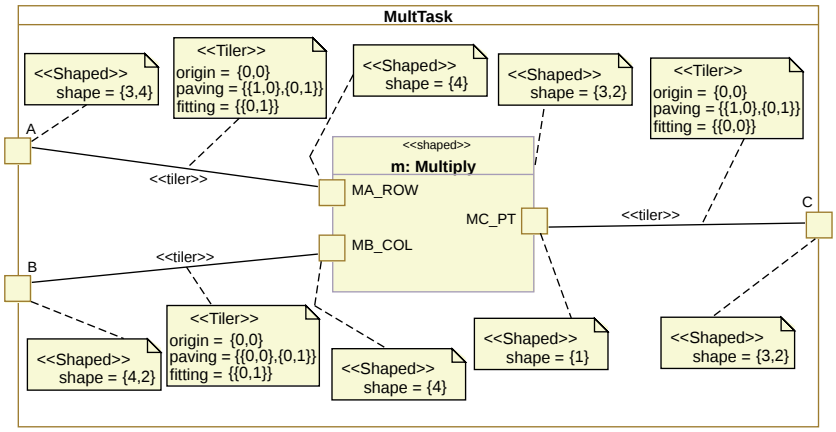


FIGURE 80: Composant modélisant la multiplication de matrices

L'application complète de multiplication de matrices est modélisée en figure 81 par le composant *Application*. Ce composant comprend 4 sous composants : *magen :MA_Gen*, *mbgen : MB_Gen*, *mtask : MultTask* et *mcprint : Mc_Print*. Les deux premiers sous composants : *magen : MA_Gen* et *mbgen : MB_Gen* sont responsables de la génération des deux matrices à multiplier. Le sous composant *mtask : MulTask* est une instance du composant que nous avons présenté en figure 80 s'occupant d'effectuer la multiplication de matrices. Finalement le dernier sous composant : *mcprint : MC_Print* est utilisé pour afficher le résultat obtenu.

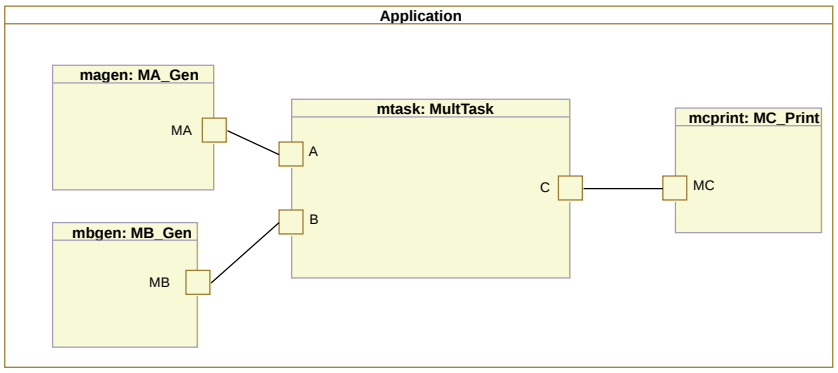


FIGURE 81: Structure générale de l'application de multiplication de matrices

8.2 DE MARTE VERS DU CODE PTHREAD

Une des chaînes de compilation intégrée à l'environnement GASPARD2 propose de générer du code C utilisant la librairie PTHREAD pour générer du code parallélisé. Nous utilisons cette chaîne pour générer automatiquement le code PTHREAD parallélisé de l'application de la multiplication de matrice. La figure 82 présente en détail la chaîne de transformations.

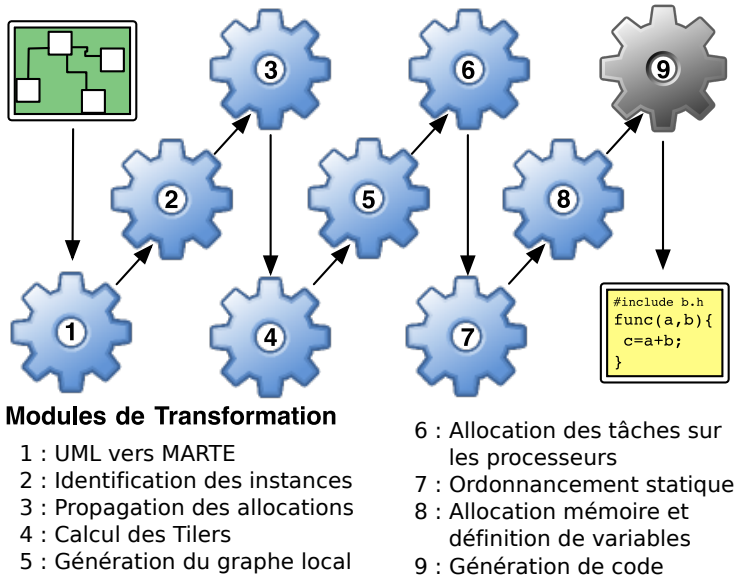


FIGURE 82: Chaîne de compilation GASPARD2 vers du code PTHREAD

Cette chaîne de compilation vers du code PTHREAD est constituée de 9 transformations : 8 transformations de modèles à modèles ponctuées par une 1 transformation de modèles vers texte. Dans cette chaîne, les transformations numéro 1, 2, 4, 5, 7 et 8 sont également présentes dans la chaîne générant du code OPENCL et ont été présentées au chapitre 6. Nous détaillons alors les transformations 3, 6 et 9.

3. Pour simplifier la modélisation, lorsqu'un composant est placé sur un processeur, tous les composants applicatifs qu'il contient sont placés sur le même processeur (sauf si le sous composant est explicitement placé sur un autre processeur). Or, pour générer le code, il est indispensable de préciser explicitement sur quel processeur chaque composant est placé. La troisième transformation s'occupe donc de propager les allocations sur les processeurs pour les composants hiérarchisés afin de placer explicitement tous les composants sur les processeurs.
6. La sixième transformation place les tâches correspondant aux *Tiler* sur les processeurs.
9. La neuvième transformation génère le code PTHREAD de l'application.

Dans cette chaîne de compilation, la plupart des transformations, dont les transformations 3 et 6, sont communes à plusieurs chaînes de compilation. La seule transformation qui est complètement dédiée à la chaîne *PThread* est la transformation de génération de code (transformation 9).

8.3 ajout du support des observations pour la chaîne de compilation *pthread*

Comme nous l'avons vu au chapitre précédent, pour ajouter le support des observations sur une chaîne de compilation existante, le développeur de la chaîne doit ajouter un certain nombre de fichiers relatifs à la génération du code des observations. Dans cette section, nous abordons alors le point de vue du développeur et nous montrons les différentes opérations qui doivent être opérées sur la chaîne de transformations pour activer le support des observations, c'est-à-dire :

- l'ensemble des *templates* générant les codes des observations,
- un fichier liant les différentes observations disponibles à leurs *templates* respectifs,
- un fichier de génération de code indiquant à quels endroits de la transformation de modèles vers texte les codes des observations peuvent être ajoutées.

Nous décrivons ci-après une partie des fichiers créés pour la génération du code des observations en PThread.

8.3.1 *Templates de génération de code*

Nous avons écrit en Acceleo-2.0.0 les différents scripts de génération pour les différents codes PThread des observations de base que nous avons fournies. De manière générale, chaque *template* fait appel à une fonction *log()* appartenant à une librairie que nous avons fournie et permettant de générer une entrée dans un fichier de *log* lors de l'exécution du code.

Dans cette sous section, à titre d'exemple, le listing 8.1 correspond au *template* Acceleo de génération du code de l'observation *Exit* pour du code PThread. Le code généré à partir de ce *template* est très court : 7 lignes (de la ligne 2 à 7). Les lignes 2 et 6 génèrent le début et la fin d'un bloc dans lequel un *mutex*[2] *logRight* est demandé (ligne 3). Nous avons fait le choix de protéger la génération des informations d'exécution par un *mutex* pour éviter que deux *threads* en cours d'exécution ne génèrent, au même moment, une ligne de *log*. Une fois le *mutex* acquis, le programme génère une ligne de *log* grâce à l'appel de la fonction *log()* (ligne 4). Le premier paramètre de cette fonction est une variable : *logger*, instance d'une structure spéciale permettant de gérer les différents accès à un fichier de *log* en particulier. Les paramètres suivant correspondent à l'UID de l'élément en train d'être exécuté (self.getUUID()), le type d'observation généré (EXIT_BREAK) ainsi que la donnée indiquant ici le nom de l'élément demandant l'arrêt du programme. Une fois la ligne de *log* générée, le *logger* est fermé (ligne 5) et l'exécution du programme est stoppée (ligne 6).

2. Technique permettant d'obtenir un accès exclusif à des ressources partagées.

```
1 <%script type="EObject" name="exit"%>
2 {
3   pthread_mutex_lock(&logRight);
4   log(logger,"<%self.getUUID()%>","EXIT_BREAK","exit call by <%
        self.name%>");
5   close(logger);
6   exit(-1);
7 }
```

Listing 8.1: Template *Exit* écrit pour générer du PThread

8.3.2 *Mapping entre templates et observations*

Une fois les *templates* créés, un autre fichier *Acceleo* permet de faire le *mapping* entre les observations qui peuvent être modélisées et les *templates*. Concrètement, cela revient à indiquer quel est le *template* à utiliser pour générer le code d'une observation.

Le listing 8.2 présente un extrait du fichier de *mapping* entre les *templates* et les observations. Pour chacune des observations rencontrées, l'extrait présenté vérifie le type des observations qu'il rencontre dans le modèle d'observation et leur associe le *template* correspondant. Par exemple, la ligne 2 du *template* permet de vérifier si le type de l'observation est *Increment*. Si c'est le cas, la ligne 3 appelle le *template increment*, sinon, les autres lignes du *template* testent successivement le type des observations et effectuent des transferts d'appels vers les *templates* associés, sans générer de code.

```
1  <%script type="obsmm.Observation" name="getTemplate"%>
2   <%if (self().filter("Increment") != null) {%>
3    <%args(0).filter("EObject").increment(args(0))%>
4   <%}else{%>
5    <%if (self().filter("InjectComment") != null){%>
6     <%args(0).filter("EObject").commentInjection%>
7    <%}else{%>
8     <%if (self().filter("Exit") != null){%>
9      <%args(0).filter("EObject").exit%>
10    <%}else{%>
11     <%if (self().filter("Activation") != null){%>
12      <%args(0).filter("EObject").activation%>
13     <%}else{%>
14      <%if (self().filter("InjectCode") != null){%>
15       <%args(0).filter("EObject").codeInjection%>
16      <%}%>
17     <%}%>
18    <%}%>
19   <%}%>
20  <%}%>
```

Listing 8.2: Expression du *Mapping* entre les *templates* et les observations

8.3.3 *Points d'entrée dans la transformation de génération de code* PThread

Une fois le fichier de *mapping* et ceux des différents *template* créés, il faut maintenant rajouter les points d'entrée[3] aux endroits où les codes des observations vont pouvoir être insérés. Le listing 8.3 montre un exemple de *template* qui a été surchargé pour y ajouter les points d'entrée pour l'insertion du code des observations. Le *template* a donc été recopié et les points d'entrées ont été ajoutés aux lignes 5, 9 et 11.

3. Les appels aux services *injectBefore*, *injectInside* et *injectAfter* d'insertion de code.

```
1  <%script type="Pthread.DistributedFunction" name="code" %>
2  <%header%>
3  {
4    unsigned i;
5    <%self().insertBefore(self())%>
6    for(i=0;i<<%repetitions%>/;i++)
7    {
8      <%function.name%>(p0);
9      <%self().insertInside(self())%>
```

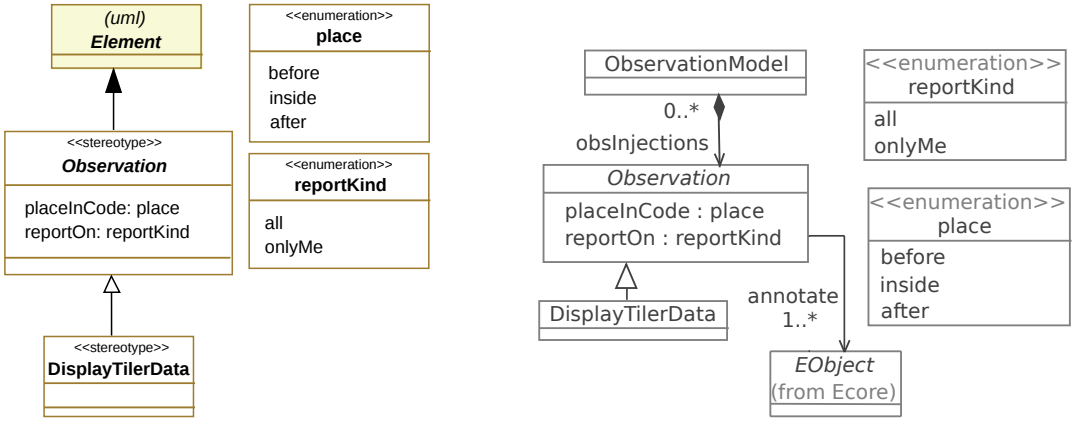
```
10    }
11    <%self().insertAfter(self())%>
12    pthread_exit(NULL);
13  }
```

Listing 8.3: Définition des points d'entrée pour le *template code*

Au total, la transformation de génération de code PTHREAD contient 47 *templates* pour la génération de code et nous avons étendu 8 *templates* pour y ajouter les points d'entrées de la transformation. Tous les *templates* ne sont pas systématiquement étendus, étant donné que beaucoup d'entre eux produisent uniquement des déclarations de variables.

8.3.4 *Ajout d'une observation spécifique*

En plus des observations de base, nous avons fourni un moyen d'observer spécifiquement les éléments *Tiler* de MARTE. En effet, ces éléments possèdent une grande importance dans MARTE car ils permettent de décider comment accéder aux données dans un tableau. Il est intéressant de pouvoir observer les données qui transitent par les *Tiler* lors de l'exécution de l'application. Cette observation est ajoutée en tant qu'observation spécifique, car elle est purement dépendante de MARTE et ne trouve aucun intérêt pour d'autres méta-modèles n'utilisant pas ce concept de *Tiler*.



(a) Extrait du profil d'observations modifié

(b) Extrait du méta-modèle d'observations modifié

Sur les figures, les éléments représentant les observations de bases ont été volontairement enlevées pour des raisons de place.

FIGURE 83: Ajout d'une observation

Modélisation

Afin de pouvoir modéliser l'observation sur le modèle de conception, nous avons étendu le profil UML d'observation que nous avons présenté au chapitre précédent pour y ajouter un stéréotype *DisplayTilerData*. En corrélation avec la modification du profil, le méta-modèle est, lui aussi, modifié pour y intégrer un nouveau concept *DisplayTilerData*. La figure 83 montre à la fois un extrait du profil d'observation modifié (figure 83a) et un extrait du méta-modèle d'observation modifié avec la nouvelle observation (figure 83b).

Ajout du template, mise à jour du fichier de mapping et ajout du point d'entrée

Maintenant que l'observation est modélisée, il faut, comme pour les observations précédentes, créer un *template*, lui associer et ajouter le point d'entrée indiquant la place à laquelle il pourra être inséré dans le code de l'application. Dans le fichier de *Mapping*, nous créons un point d'entrée spécial nommé *insertDisplayTiler* qui vérifie si le type de l'observation est bien *DisplayTilerData* et lui associe le *template* correspondant.

Nous présentons ici uniquement un extrait du fichier de génération de code dans lequel nous avons placé le code du point d'entrée spécifique : *insertDisplayTiler* à l'observation *DisplayTilerData*. On y retrouve ligne 3, 31 et 41 les points d'entrée génériques[4] et ligne 32 le point d'entrée spécifique à l'observation modélisée.

4. *InsertBefore*, *InsertInside* et *InsertAfter*.

```
1  <%script type="Pthread.TilerFunction" name="calculation" %>
2   <%push()%>
3   <%self().insertBefore(self())%>
4   <%for tilerTask.tiler.origin.vectorElem{%>
5    <%if peek().tilerTask.tiler.paving.matrixElem.vectorElem.nSize
         () > 1 {%>
6     ref[<%i()%>] = <%self%><%peek().paving (self.length() - i())%>;
7    <%}else{%>
8     ref[<%i()%>] = <%self%><%peek().paving (i())%>;
9    <%}%>
10  <%}%>

12  <%if tilerTask.direction == "in" {%>
13   <%puts.acquire("buffer_out")%>
14  <%}else{%>
15   <%get.acquire("buffer_in")%>
16  <%}%>
17  <%for patternshape.size {%>
18    for(tl[<%i()%>]=0; tl[<%i()%>] < <%self%>; tl[<%i()%>]++) {
19  <%}%>

21  <%for tilerTask.tiler.origin.vectorElem {%>
22   index[<%i()%>]= (ref[<%i()%>]<%peek().fitting (i())%>)%<%peek().
         oppositeDimension(i())%>;
23  <%}%>

25  <%affectation%>

27  <%for patternshape.size {%>
28    }
29  <%}%>

31  <%self().insertInside(self())%>
32  <%self().insertDisplayTilerData(self())%>

34  <%if tilerTask.direction == "in" {%>
35   <%puts.release("buffer_out")%>
36  <%}else{%>
37   <%get.release("buffer_in")%>
38   buffer_in= (buffer_in + 1) % <%get.buffer.repet%>;
39  <%}%>

41  <%self().insertAfter(self())%>
42  <%pop()%>
```

Listing 8.4: Extrait du *template* avec le point d'entrée spécifique à l'observation ajoutée

8.3.5 *Une chaîne de transformations pour le support des observations de la chaîne de compilation* PTHREAD

Afin de pouvoir être effectuée, l'insertion des observations s'appuie sur les modèles intermédiaires, ainsi que sur la trace globale déjà générée par une exécution de la chaîne de transformations. Pour réutiliser ces modèles intermédiaires, nous avons fourni une nouvelle chaîne de transformations que nous avons représentée en figure 84. La chaîne se compose de 3 transformations : 2 transformations de modèles à modèles et 1 transformation de génération de code. La chaîne commence par consommer la trace globale pour produire une trace réduite (transformation 1). Cette trace réduite est ensuite utilisée avec le modèle de conception UML profilé avec MARTE et le profil d'observation pour produire un modèle d'observations (transformation 2). Cette transformation créé des concepts *Observations*, relatifs aux observations modélisée dans le modèle *uml* et les lie aux éléments du dernier modèle, le modèle M8[5] dont ils ont permis la génération. La trace réduite est donc utilisée ici pour déterminer les éléments du modèle M8 qui sont liés aux éléments stéréotypés par une observation. Le modèle d'observation produit est ensuite utilisé avec le modèle M8 pour générer le code de l'application embarquant le code des observations (transformation 3). Cette dernière transformation n'embarque pas la totalité des règles de transformations de la génération de code PTHREAD originale, mais hérite de celle-ci pour pouvoir accéder à tous les *templates* originalement créés et pouvoir générer le code complet de l'application.

5. Ce modèle est généré par la transformation 8.

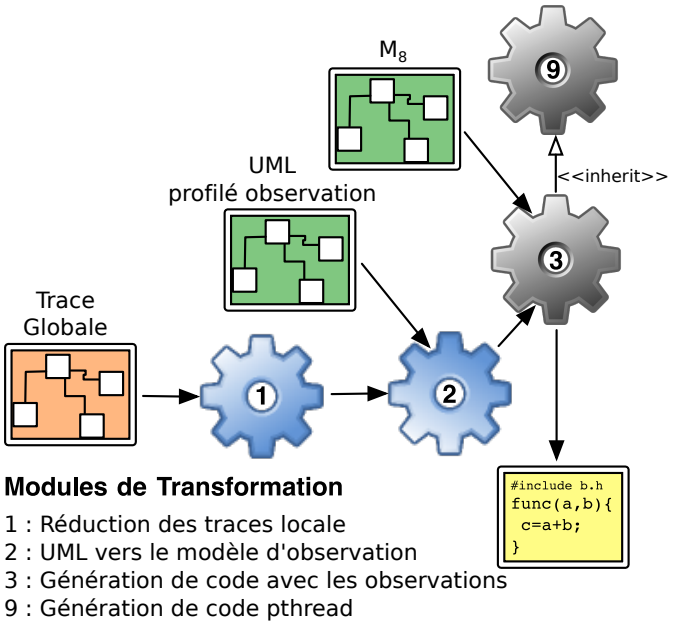


FIGURE 84: Chaîne de génération des observations

À l'issue de ces modifications sur la chaîne de compilation, le développeur a activé le support pour les observations sur la chaîne de

compilation PTHREAD. Ces opérations n'ont pas besoin d'être répétées et ne sont effectuées qu'une seule fois par chaîne. La suite du chapitre correspond au point de vue du concepteur de modèles qui utilise ce qui vient d'être présenté sans en avoir réellement connaissance.

8.4 OBSERVATION DU MODÈLE

Grâce aux modifications décrites ci-dessus, la chaîne de compilation *PThread* supporte à présent les observations. Le concepteur de modèles peut maintenant préciser sur son modèle de conception les parties qu'il souhaite observer lors de l'exécution du code correspondant. Dans cette section, nous allons voir, comment les observations demandées sur le modèle peuvent aider à identifier facilement et résoudre une erreur présente dans celui-ci.

Le code PTHREAD de l'application de multiplication de matrices est automatiquement généré à partir des modèles UML. Le code est ensuite exécuté. Pour cette multiplication, les matrices utilisées sont les suivantes :

$$A = \begin{Bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{Bmatrix} \text{ et } B = \begin{Bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \\ 7 & 8 \end{Bmatrix}.$$

Le résultat attendu pour la multiplication de matrices de A avec B est donc :

$$C = \begin{Bmatrix} 50 & 60 \\ 114 & 140 \\ 178 & 220 \end{Bmatrix}.$$

Cependant, le code généré à partir du modèle de conception produit le résultat suivant :

$$C = \begin{Bmatrix} 102 & 124 \\ 106 & 132 \\ 134 & 164 \end{Bmatrix}.$$

Ceci implique qu'une erreur a été introduite dans le modèle lors de la phase de conception.

8.4.1 *Modélisation des observations*

Afin de situer l'erreur et de comprendre le comportement de l'application, nous allons demander à observer les valeurs manipulées à l'exécution par l'application et plus particulièrement par le composant *MultTask*. Afin de modéliser les observations que nous aimerions effectuer lors de l'exécution, nous stéréotypons les éléments de ce composant grâce à notre profil d'observation.

La figure 85 montre le composant *MultTask* avec les stéréotypes d'observation, correspondants aux observations que nous avons demandées. Ici, nous avons demandé à observer les données transitant dans les 3 *Tiler* modélisés. Précisément, nous avons demandé d'insérer l'observation des données transitant dans les *Tilers*, à l'intérieur du code généré pour chaque *Tiler* (placeInCode = inside) et nous avons demandé que les retours des observations relatives à ce *Tiler* soient seulement donnés sur l'élément stéréotypé (reportOn = onlyMe). Sur le modèle, il est aussi précisé que l'exécution du programme est stoppée dès qu'une valeur est

calculée (placeInCode = after). L'arrêt du programme a été demandé après un seul calcul pour ne pas continuer inutilement l'exécution de l'application, étant donné que nous voyons bien que toutes les valeurs retournées par l'application, en l'état, sont complètement différentes de celles que nous attendions.

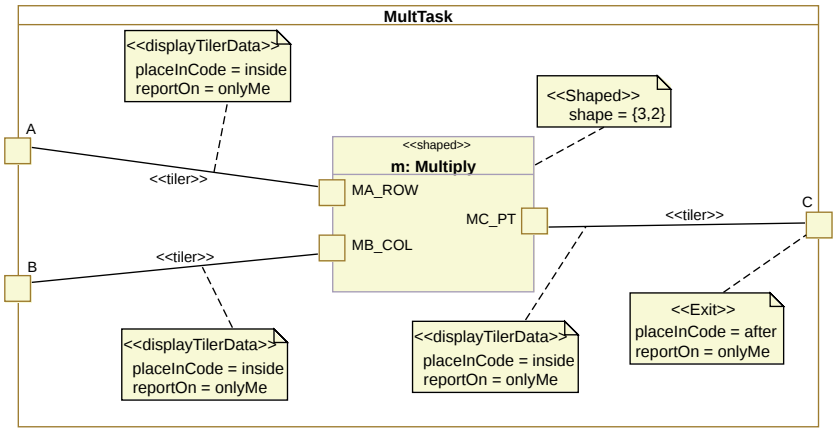


FIGURE 85: Détail du sous composant *dotProd* avec les stéréotypes d'observations

8.4.2 *Génération du code instrumenté avec le code des observations*

Une fois que les observations sont ajoutées, la chaîne spécifique à l'insertion du code des observations est lancée. La première étape de cette chaîne est de produire une trace réduite. La table 4 montre les éléments du dernier modèle auxquels les éléments du modèle de conception, que nous avons stéréotypés avec des observations, sont liés. Par exemple, le premier élément *Tiler* allant du port d'entrée *A* du composant *MultTask* au port d'entrée *MA_ROW* du sous composant *m : Multiply* a donné naissance[6] à 2 éléments de type *PortConnector*, 1 élément de type *AssemblyConnector* et 1 éléments de type *TilerFunction*.

Une fois la trace réduite générée, elle est utilisée par la seconde transformation de la chaîne de transformations[7]. Pour mémoire, le modèle d'observation est produit, à la fois, à partir de la trace réduite et à partir du modèle UML avec le profil d'observation afin de lier chacune des observations posées sur le modèle de conception aux éléments du dernier modèle qu'ils ont généré.

Le modèle d'observation produit est représenté en figure 86. On y retrouve distinctement les 4 observations que nous avons posées sur le modèles. Il est aussi possible de voir que l'observation *DisplayTilerData* en surbrillance pointe vers les éléments du dernier modèle de la chaîne de transformations récupérés grâce à la trace réduite[8].

Finalement, la dernière étape de la chaîne produit le code généré avec le code des observations inséré dans le code de l'application. Compte tenus du fait que les observations sont liées à plus d'un élément, il est raisonnable de penser que le code d'une observation apparaîtra à plusieurs endroits. Cependant, les *templates* dans lesquels ont été placés les points d'entrée pour l'injection des codes des observations ne s'appliquent pas sur tous les éléments récupérés par la trace locale. Par exemple, chaque élément *Tiler* est associé à 4 éléments dans le modèle avant la génération de code. Cependant, seul le *template* relatif aux *TilerFunctions* a été

6. Dans le dernier modèle de la chaîne de transformations avant la génération de code.

7. Générant le modèle d'observation.

8. L'ensemble total des éléments vers lesquels l'observation point n'apparaît pas sur la figure.

ÉLT. SOURCE	ÉLT. DESTINATION
Tiler (A vers MA_ROW)	noName : AssemblyConnector _A_To_MA_ROW_dev : PortConnector _A_To_A_mtask : PortConnector tiler_from_r__Tpp5_A_to_w__Tpp6_MA_ROW_m : TilerFunction
Tiler (B vers MB_COL)	noName : AssemblyConnector _B_To_MB_COL_mtask : PortConnector _B_To_B_mtask : PortConnector tiler_from_r__Tpp1_B_to_w__Tpp2_MB_COL_m : TilerFunction
Tiler (MC_PT vers C)	noName : AssemblyConnector _C_To_C_mtask : PortConnector _MC_PT_To_C_mtask : PortConnector tiler_from_r__Tpp3_MC_PT_m_to_w__Tpp4_C : TilerFunction
C : Port	C_mtask : PortPart C_mtask : Buffer C_mtask : FlowPort

TABLE 4: Extrait de la trace locale générée

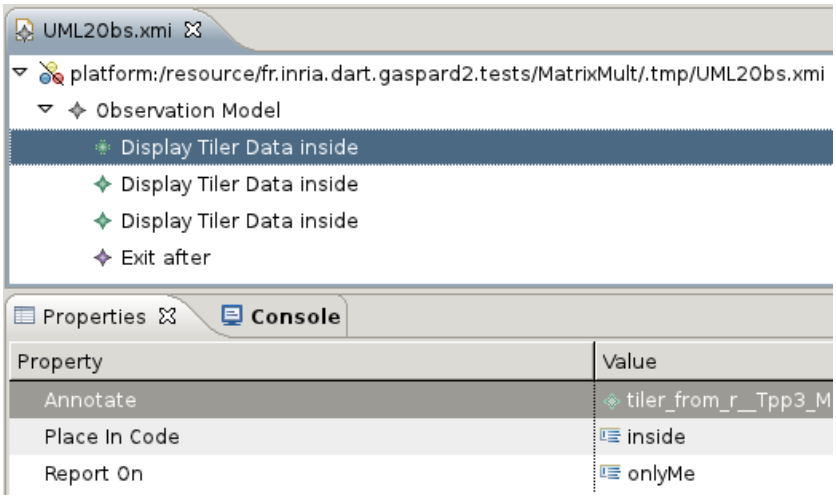


FIGURE 86: Modèle d'observations produit

surchargé, ce qui signifie que le code de l'observation sera uniquement présent dans le code généré à partir des *TilerFunctions*. Le listing 8.5 présente la fonction *tiler_from_r__Tpp3_MC_PT_m_to_w__Tpp4_C* générée pour le *Tiler* allant du port de sortie *MC_PT* du sous composant *m : Multiply* vers le port de sortie *C* du composant *MultTask*. Nous pouvons voir que le code généré pour ce *Tiler* contient aussi le code généré pour l'observation *DisplayTilerData* posée sur ce *Tiler* ainsi que le code généré pour l'observation *Exit* posée sur l'élément *C : Port*. Dans cette portion de code généré, la section relative au code généré pour l'observation *DisplayTilerData* est visible de la ligne 30 à 44 et la section relative au code généré pour l'observation *Exit* est visible de la ligne 48 à 52.

```
1 void tiler_from_r__Tpp3_MC_PT_m_to_w__Tpp4_C(int p0)
2 {
3   unsigned tilerIteration[2];
4
```

```
5   unsigned int tl[1];
6   unsigned int ref[2];
7   unsigned int index[2];
8
9   unsigned buffer_in = 0;
10
11  unsigned * w__Tpp4_C=queue_get_write(MC_mcprint[p0]);
12  for(tilerIteration[0]=0; tilerIteration[0] < 3; tilerIteration
        [0]++)
13  {
14    for(tilerIteration[1]=0; tilerIteration[1] < 2; tilerIteration
          [1]++)
15    {
16
17      ref[0] = 0+0 * tilerIteration[0]+ 1 * tilerIteration[1];
18      ref[1] = 0+1 * tilerIteration[0]+ 0 * tilerIteration[1];
19
20      unsigned * r__Tpp3_MC_PT_m=queue_get_read(MC_PT_m[buffer_in
            ]);
21      for(tl[0]=0; tl[0] < 1; tl[0]++) {
22
23    index[0]= (ref[0]+ 0 * tl[0])%3;
24    index[1]= (ref[1]+ 0 * tl[0])%2;
25
26    w__Tpp4_C[index[0] * 2 +
27        index[1] * 1]=r__Tpp3_MC_PT_m[tl[0] * 1];
28      }
29      {
30    int i = 0;
31    char res[256];
32    char temp[10];
33
34    pthread_mutex_lock(&logRight);
35    sprintf(res,"[");
36    for (i=0;i<1;i++){
37      sprintf(temp," %d ", r__Tpp3_MC_PT_m[1*i]);
38      strcat(res,temp);
39    }
40    strcat(res,"]");
41    log(logger,"_9QLdruqUEeCKSMMjmDvdzQ","DISP_TILER",res);
42    pthread_mutex_unlock(&logRight);
43      }
44      release_read(MC_PT_m[buffer_in]);
45      buffer_in= (buffer_in + 1) % 1;
46      {
47    pthread_mutex_lock(&logRight);
48    log(logger,"_9QK2luqUEeCKSMMjmDvdzQ","EXIT_BREAK","exit
          call by C_mtask");
49        close(logger);
50        exit(-1);
51      }
52    }
53  }
54  release_write(MC_mcprint[p0]);
55 }
```

Listing 8.5: Extrait du code généré avec le code des observations inséré

8.4.3 *Exécution et remontée d'informations sur le modèle de l'application de multiplication de matrices*

Le code de l'application produit est compilé, puis exécuté. Lors de son exécution, l'application produit un *log* d'observation qui est parsé en un modèle d'*Advice*[9] qui relate de l'exécution de l'application et peut être vu comme un fichier de *log*. Le modèle produit par l'exécution de l'application est montré en figure 87. Lors de l'exécution de l'application, 4 *Advices* ont été générés à partir des observations injectées. Chaque *Advice* possède un UID qui correspond à l'UID de l'élément du dernier modèle avant la génération de code qui a permis sa génération. Par exemple, l'*Advice* mis en surbrillance sur la figure nous permet de voir que c'est l'élément portant l'UID *_9QK2luqUEeCKSMMjmDvdzQ* et se nommant *C_mtask* qui a mené à sa génération. De plus, nous pouvons voir que cet *Advice* a été généré par l'exécution du code d'une observation *Exit* (l'attribut *kind* vaut EXIT_BREAK).

9. Le même modèle d'*Advice* que celui présenté au chapitre 4

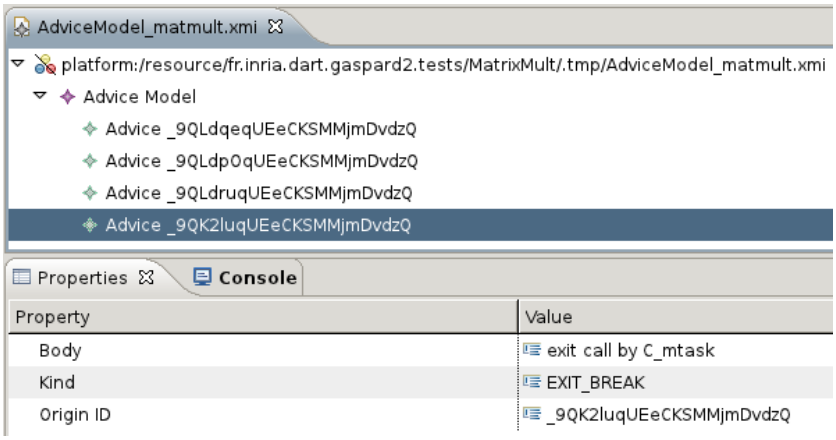


FIGURE 87: Modèle d'*Advice* généré

Ce modèle est consommé par notre algorithme de remontée[10] pour permettre la visualisation des informations directement sur le modèle de conception. L'algorithme de remontée utilise le modèle d'*Advice* généré par l'exécution de l'application ainsi que la trace réduite pour déterminer sur quels éléments l'information va être remontée. Finalement, la figure 88 montre le modèle de conception annoté avec les informations récoltées lors de l'exécution du programme. Les annotations sont représentées dans le modèle de conception comme des commentaires liés aux éléments observés. Chacun des commentaires reprend les informations obtenues à l'exécution de l'application. Par exemple, il est possible de voir que le *Tiler* entre le port *A* et le port *MA_ROW* a vu transiter une donnée de 4 éléments : $[1, 6, 11, 4]$. Le *Tiler* entre le port *B* et le port *MB_COL* a lui aussi vu transiter une donnée de 4 éléments : $[1, 3, 5, 7]$. Finalement, la valeur $[102]$ a transitée une fois par le *Tiler* situé entre le port *MC_PT* et le port de sortie du composant *C*. Le programme s'est ensuite arrêté.

10. Présenté au chapitre 5.

8.4.4 *Analyse des résultats et correction du modèle*

Ces retours, directement accessibles sur le modèle de conception, nous sont utiles pour vérifier le bon déroulement de l'algorithme. Nous observons que la ligne envoyées sur le *Tiler* d'entrée pour la matrice *A*

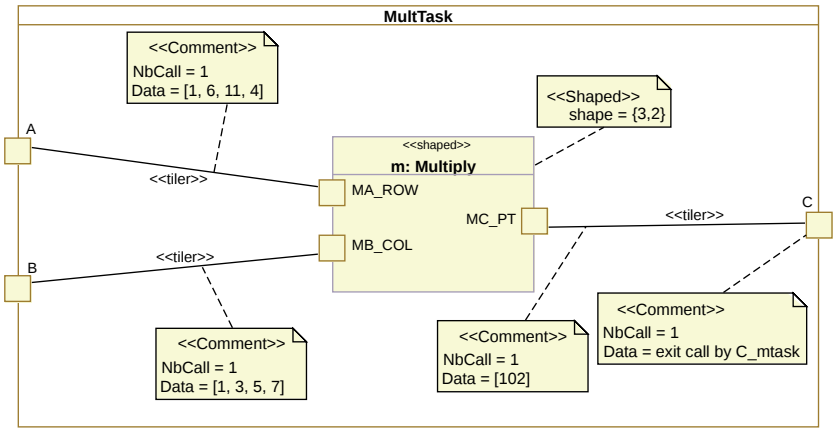


FIGURE 88: Modèle UML annoté avec les informations demandée

n'est pas la bonne. En effet, pour la matrice A, les lignes qui auraient dû être considérées sont : $[1,2,3,4]$, $[5,6,7,8]$ ou $[9,10,11,12]$ et non pas $[1,6,11,4]$ comme cela nous est reporté sur le modèle. Cela signifie que les données ne sont pas sélectionnées correctement dans le tableau passé au port d'entrée du composant, c'est donc la définition du *Tiler* qui est remise en cause. Une analyse plus précise du *Tiler* (que nous ne développons pas ici, mais qui peut être trouvée dans [32, 20]), chargé de sélectionner les données dans le tableau d'entrée permet de le modifier de sa valeur initial $\mathtt{origin} = \{0,0\}, \mathtt{paving} = \{\{1,0\},\{0,1\}\}, \mathtt{fitting} = \{\{1,1\}\}$ à $\mathtt{origin} = \{0,0\}, \mathtt{paving} = \{\{1,0\},\{0,1\}\}, \mathtt{fitting} = \{\{0,1\}\}$.

Si nous regardons maintenant la donnée qui a transité par le second *Tiler* d'entrée, chargé de récupérer une colonne de la matrice B, nous observons que celle-ci est correcte, ce qui laisse sous entendre que la seule erreur du modèle venait du *Tiler* d'entrée que nous venons de modifier.

Une fois la modification terminée, nous vérifions que celle-ci corrige bien le comportement de l'application. Le code de l'application est alors généré à nouveau, compilé, puis exécuté. Cette fois le résultat obtenu est :

$$C = \begin{Bmatrix} 50 & 60 \\ 114 & 140 \\ 178 & 220 \end{Bmatrix},$$

qui est bien celui attendu pour les données de test que nous avons utilisées. L'erreur de comportement a donc été corrigée directement depuis les modèles de conception sans avoir eu connaissance du code source généré pour l'application.

8.5 CONCLUSION

Dans ce chapitre, nous avons illustré sur un cas d'étude notre approche d'observation de modèles. Nous avons montré comment une chaîne de transformations doit être modifiée par le développeur pour supporter l'insertion d'observations dans le code source de l'application et comment le concepteur peut utiliser les observations pour corriger son programme.

L'application que nous avons détaillée comme cas d'étude dans ce chapitre est l'application de multiplication de matrices. Cette applica-

tion, a été modélisée dans l'environnement GASPARD2 en UML avec le profil MARTE. Une fois la modélisation terminée, le code de l'application a été généré et l'application a été exécutée. Les résultats initialement obtenus étant erronés, nous avons utilisé notre profil d'observations pour regarder en détail les valeurs manipulées à l'exécution. Une fois les observations posées sur le modèle, le code de l'application, embarquant des morceaux de codes additionnels, a été généré, puis compilé et exécuté.

Cette fois, l'exécution de l'application a produit un modèle contenant des mesures et des valeurs calculées et manipulées à l'exécution. Ces valeurs et mesures ont été remontées directement sur le modèle de conception pour donner au concepteur de modèles un retour sur l'exécution de son application. Les informations retournées sur le modèle ont pu indiquer que certains éléments du modèle avaient été mal modélisés et qu'ils accédaient à de mauvaises valeurs. Le modèle a donc été corrigé et le code de l'application généré à nouveau.

Notre mécanisme d'observation a donc permis d'effectuer une première activité de *debugging* directement à partir du modèle sans avoir connaissance du code de l'application généré. Le mécanisme que nous fournissons est, de plus, extensible et permet l'ajout de nouvelles observations dédiées à un langage ou un type d'application en particulier. Ce mécanisme pourrait être utilisé pour récupérer systématiquement des informations et mesures sur tous les éléments d'un programme et une analyse plus fine des informations récoltées à l'exécution pourrait permettre de déterminer certaines situations à risque comme, par exemple, des *deadlocks*. Néanmoins, quelque soit l'utilisation faite de ce mécanisme, il faut garder à l'esprit que l'instrumentation du code peut entraîner des différences de performances et, potentiellement, de comportement.

Pour utiliser ce support et ce mécanisme d'observation, nous avons émis l'hypothèse forte que la chaîne de transformations ne comporte aucune erreur. Nous nous étions donc placé dans le cadre d'un programmeur classique cherchant la faute de son application dans le code de son programme plutôt que dans le compilateur. Cependant, les chaînes de transformations peuvent introduire des erreurs et nécessitent donc d'être testée. Dans la suite de cette thèse, nous nous sommes penché sur le problème des tests de transformations pour assurer qu'aucune erreur n'est introduite automatiquement lors de la phase de compilation de l'application modélisée.

Quatrième partie

DEBUGGER ET TESTER

9 TEST DE TRANSFORMATIONS

Dans les chapitres précédents, nous avons présenté divers algorithmes et mécanismes basés sur la trace locale et la trace globale. Au chapitre 5 et 7, nous avons utilisé notre mécanisme de trace pour permettre au concepteur de modèles d'observer, de *debugger* et d'améliorer les performances des applications modélisées. L'hypothèse forte de ces travaux est que les chaînes de compilation IDM sont, à l'image d'un compilateur classique, digne de confiance et jugées sans erreurs.

Cependant, poser cette hypothèse impose de tester efficacement les différentes transformations composant une chaîne. Déterminer et comprendre quelles sont les erreurs d'une transformation est une étape importante du processus de création d'une chaîne de compilation IDM qui est nécessaire pour trois raisons principales :

- Une erreur dans une transformation peut se propager dans les modèles jusqu'à l'implémentation. Il est alors très difficile de localiser l'origine de cette erreur à partir du code.
- Une transformation de modèle a pour vocation d'être réutilisée plusieurs fois afin de justifier l'effort que l'on consacre à sa création et à sa mise au point. Si la transformation est défectueuse, alors elle risque de générer les mêmes erreurs un certain nombre de fois.
- Le système peut évoluer entraînant une modification de la transformation erronée afin de prendre en compte les nouvelles spécificités du système évoluant et conduire à des erreurs. Ceci implique la modification de plusieurs sous parties pour amener de nouveau à une configuration stable.

Une phase de test est donc nécessaire pour les transformations de modèles [46, 22].

Dans ce chapitre, nous présentons les travaux existants sur les tests de transformations de modèles en section 9.1. Afin de détailler les différents challenges donnés par le test des transformations de modèles, nous présentons en section 9.2 comment les jeux de données de test sont générés, en section 9.3, nous montrons quelles techniques sont utilisées pour la correction des transformations de modèles, puis en section 9.4 comment les oracles déterminant le succès ou non d'un test sont construits. Finalement, nous concluons en section 9.5.

9.1 PROCESSUS DE TEST DE TRANSFORMATION

Le processus de test de transformation est assez semblable à celui proposé pour le test de logiciels [133]. D'un point de vue synthétique, cela consiste à exécuter un programme avec un ensemble de données en entrée et à vérifier que les résultats obtenus sont bien ceux attendus. Si ce n'est pas le cas, ce test a détecté une erreur qu'il faut corriger. La figure 89, illustre ce processus. Le programme exécuté par les tests est : la *transformation*. Les données d'entrée de la transformation sont, par conséquent, des *modèles de test*. L'exécution de la *transformation* sur les *modèles de test* produit des *modèles de sortie* qui sont évalués par un *oracle*. C'est lui qui annonce le succès ou l'échec du test. Si le test échoue, cela signifie qu'un erreur existe dans la transformation et qu'elle doit être localisée, puis corrigée. Le processus reprend ensuite. Si le test est un succès, un *critère de test* juge si la *transformation* a été suffisamment testée. Si ce n'est pas le cas, de nouveaux *modèles de test* sont produits et le processus reprend. Si la transformation est jugée suffisamment testée, le processus termine.

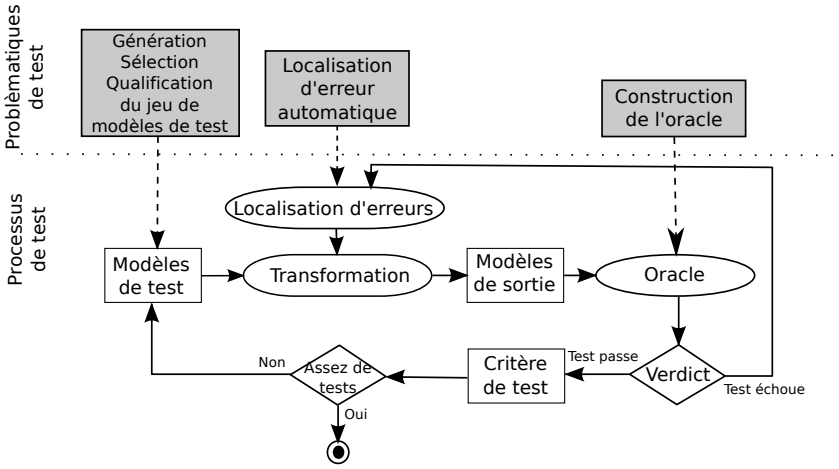


FIGURE 89: Processus de test de logiciels

Dans ce processus, plusieurs problématiques sont levées [1]. Dans les sections suivantes, nous allons présenter les différents travaux effectuer pour les problématiques de *génération, sélection et qualification du jeu de modèles de test*, de *localisation d'erreurs* et de *construction de l'oracle* dans le contexte du test de transformation de modèles.

1. Ces problématiques ne sont pas directement liées à l'IDM, mais au test de logiciel de manière générale.

9.2 GÉNÉRER UN JEU DE MODÈLES DE TEST

La génération du jeu de modèles de test correspond au premier challenge du processus de test de transformation de modèles. En effet, compte tenu des structures complexes manipulées par les transformations de modèles, c'est-à-dire des modèles conformes à des méta-modèles, la génération automatique des données de test efficace est une tâche complexe.

9.2.1 *Génération automatique de modèles de test*

Une méthode proposée dans [80], utilise un langage de *template*, défini par les auteurs, reposant sur les éléments d'entrée et de sorties, ainsi que la structure des règles de la transformation testée. Cependant, cette méthode repose sur une approche boite blanche des transformations [2] et, est fortement dépendante du langage de transformation utilisée. Évidemment, cette méthode nécessite d'être adaptée ou redéfinie entièrement lorsqu'elle est portée pour un nouveau langage de transformation.

2. L'implémentation est connue.

Dans [111], les auteurs tentent d'obtenir des modèles de tests conformes à un méta-modèle en combinant différentes contraintes exprimées en PROLOG. De cette façon, il est possible d'obtenir des modèles de test à partir du méta-modèle source d'une transformation et de contraintes complémentaires comme les pré-conditions d'une transformation. Cependant, PROLOG ne permet pas de contraindre suffisamment la création de modèles. Une alternative est donc fournie par [110] où ALLOY [66] est utilisé pour exprimer l'ensemble des contraintes de génération des modèles de test.

Afin de générer des modèles de test, d'autres travaux considèrent les modèles manipulés comme des graphes. Ainsi, dans [39], les auteurs proposent une technique pour générer automatiquement des ensembles de modèles par dérivation automatique de grammaire de graphe à partir du méta-modèle. Le méta-modèle est analysé pour générer des règles créant et associant des instances des méta-classes du méta-modèle. Cependant, cette approche souffre de nombreuses limitations, dont : la non initialisation des attributs des instances créées. Une autre approche proposée dans [29], considère des règles de transformation de graphes comme une spécification de la transformation testée. La génération de modèles de test s'effectue à partir de cette spécification.

9.2.2 *La qualification de modèles de test*

Depuis quelques années, les tests de transformations de modèles sont de plus en plus présents dans la littérature. Les thématiques abordées sont très souvent la génération des modèles de test ainsi que la construction de l'oracle. Cependant, très peu de travaux travaillent sur la qualification et l'amélioration d'un jeu de modèles de test. La qualification de modèles de test permet de mesurer l'efficacité de modèles de tests à mettre en valeur les erreurs dans une transformation. C'est une activité importante qui permet de fournir le jeu de tests le plus adapté pour le test d'un programme en particulier. Dans le cadre de l'IDM, cela revient à fournir le jeu de modèles de test le plus adapté pour tester une transformation de modèle.

Fleurey et al. proposent de qualifier un jeu de modèles de test en fonction de sa couverture du domaine d'entrée [47, 48], où le domaine d'entrée est défini avec des méta-modèles et des contraintes. La qualification du jeu de tests est donc calculée en fonction de la capacité des modèles de test à couvrir le domaine d'entrée de la transformation. Cependant, cette qualification proposée est statique et repose uniquement sur le domaine d'entrée de la transformation, alors qu'une analyse dynamique permettrait d'évincer, lors de l'exécution, certains modèles. De plus, dans le cas de certaines transformations qui ne manipulent qu'une petite partie du domaine d'entrée, cette approche peut juger

comme efficace un jeu de modèles de test contenant plus de modèles que nécessaire.

Une autre approche, publiée en [46], propose une adaptation d'un algorithme bactériologique aux transformations de modèles. L'algorithme bactériologique [12] est conçu pour automatiquement améliorer la qualité d'un jeu de données de test. L'adaptation proposée consiste à créer de nouveaux modèles de test en couvrant des parties du domaine d'entrée qui ne sont pas couvertes par le jeu de modèles de test actuel. Les auteurs utilisent l'algorithme bactériologique pour sélectionner les modèles les plus prometteurs et pour les combiner afin de créer de nouvelles données de test.

Ces deux approches reposent sur un partitionnement du domaine d'entrée. Cependant, même si le partitionnement du domaine d'entrée fournis une aide à la qualification des modèles de test, il ne procure pas assez d'informations et de méthodes pour améliorer les modèles de test lorsqu'une haute qualité est requise[3]. Une autre technique, l'analyse de mutation, est utilisée afin d'améliorer efficacement la qualité du jeu de tests.

3. Par exemple, pour des transformations manipulant des domaines d'applications critiques.

9.2.3 *L'analyse de mutation pour la qualification du jeu de modèles de test*

L'analyse de mutation est une technique pour la qualification de données de test, proposée dans [33] par DeMillo pour des programmes traditionnels, est utilisée dans deux buts : évaluer la qualité et assister la génération de tests. Cette technique repose en grande partie sur l'utilisation de versions erronées du programme sous test. Ces versions sont systématiquement créées grâce à des opérateurs de mutation qui injectent des erreurs représentant celles communément trouvées dans les programmes.

Depuis sa création, l'analyse de mutation a été beaucoup étudiée. Dans [69], les auteurs proposent une enquête sur le développement de l'analyse de mutation et ont observé une augmentation des publications sur l'analyse de mutation depuis 1978, année de sa création, jusqu'à 2009, montrant ainsi que l'analyse de mutation est une technique largement étudiée et éprouvée. D'ailleurs, plusieurs travaux [82, 116, 115], comparant différentes approches, recommandent l'utilisation de l'analyse de mutation. Ils analysent la capacité de l'analyse de mutation à qualifier des jeux de tests et les avantages qu'elle propose. Notamment sa capacité à finalement produire peu de données de test supplémentaires et à considérer ce jeu de tests comme efficace, là où les méthodes reposant sur une analyse de couverture qualifieraient le jeu de tests comme insuffisant et demanderaient la création de plus de données de test.

La plupart des travaux reposant sur l'analyse de mutation traitent de la création des opérateurs de mutations. Les opérateurs de mutations sont directement créés en fonction de la syntaxe de langage de programmation dans lequel le programme testé est écrit : C [11], ADA [105] ou encore JAVA [86]. Ces opérateurs de mutation prennent en considération les paradigmes du langage, par exemple, impératif [7], ou objet [86]. Ainsi, ils peuvent difficilement être portés vers d'autres langages de programmation.

Il existe peu de travaux sur l'analyse de mutation qui considèrent des opérateurs de mutation indépendemment de la syntaxe du langage. Dans [45] les auteurs proposent des opérateurs de mutations en fonction

des caractéristiques de la programmation par aspect. Plus récemment, dans [114], les auteurs ont proposé MuDeL, un langage dédié à la description d'opérateurs de mutations reposant sur des définitions génériques qui peuvent être réutilisées quel que soit le langage de programmation utilisé. Cependant, ces travaux font état uniquement de langages de programmation classique et ne s'intéressent pas aux transformations de modèles.

Afin de prendre en compte les spécificités des transformations de modèles, dans [90], Mottu et al. ont proposé 10 opérateurs de mutations dédiés à la mutation des transformations de modèles. Ces opérateurs ne sont pas définis en fonction d'un langage de transformation en particulier et peuvent donc être implantés pour n'importe quel langage de transformation. Par exemple, dans [50], Fraternali et al. ont proposé une implémentation de ces opérateurs pour ATL. Un autre travail [121] a représenté ces opérateurs de mutation dédiés comme un ensemble de HOTs modifiants automatiquement la transformation testée pour en produire des versions erronées. Cependant, certains langages de transformation n'utilisent pas de représentation interne à base de modèles pour la transformation, ou cette représentation n'est pas connue ou accessible et rend cette technique à base de HOTs difficile à réaliser.

Comme nous l'avons évoqué précédemment, l'analyse de mutation possède deux buts principaux : la qualification du jeu de tests et son amélioration. Parmi les différents travaux existant sur l'analyse de mutation et les moyens d'automatiser cette technique, plusieurs d'entre eux relatent de la description des opérateurs de mutation, mais, à notre connaissance, aucun ne travaille sur l'activité d'amélioration du jeu de modèles de test. Cette activité, toujours effectuée manuellement, requiert une grande expertise et possède un coût temporel non négligeable dans le processus lorsqu'il s'inscrit dans le processus de test d'une transformation de modèle.

9.3 LOCALISATION D'ERREURS DANS LES TRANSFORMATIONS DE MODÈLES

La localisation d'erreurs pour les transformations de modèles consiste à trouver la position des erreurs dans une transformation, ou dans une chaîne de transformations. C'est une activité qui est, elle aussi, coûteuse en temps, car elle est très souvent effectuée à la main. Dans ce cas, le but de cette activité est d'assister le testeur lors de son activité de recherche des règles ou instructions erronées dans la transformation.

Dans le contexte des transformations de modèles, peu de travaux tente de fournir une solution et un moyen d'assister efficacement le testeur. Dans [136, 135], les auteurs proposent de fournir un outil de *debugging* pour le langage QVT et plus particulièrement pour son sous-langage Relation. Afin de pouvoir exploiter une représentation de l'exécution de la transformation, les auteurs effectuent un changement de formalisme en transformant la transformation exprimée avec le sous-langage Relation en réseaux de pétri colorés [68]. Il est ainsi possible de bénéficier d'outils existants. Ainsi, les auteurs sont en mesure de fournir une exécution *pas-à-pas* de la transformation donnant une aide précieuse au testeur. Cependant, même si la recherche de l'erreur est simplifiée, elle n'en reste pas moins longue si les transformations sont complexes et le testeur doit avoir une idée de la partie fautive de la transformation s'il veut accélérer la recherche.

Hors du contexte de l'IDM, pour effectuer cette localisation d'erreur, la plupart des travaux existants proposent d'exploiter la trace d'exécution du programme testé. La localisation d'erreur consiste alors à déterminer un ensemble d'instructions suspectes du programme testé. Pour cela, Agrawal et al. proposent une technique reposant sur deux exécutions du programme testé : une correcte et une défaillante [1]. Toutes les instructions qui sont uniquement traversées par l'exécution défaillante sont marquées comme potentiellement porteuses de fautes. Cette technique a été améliorée par [37, 70] en classant les instructions par ordre, de celle la plus susceptible à la moins susceptible de porter une faute.

9.4 UN ORACLE POUR LE TEST DE TRANSFORMATION DE MODÈLES

Dans le processus de test de transformation de modèles, l'oracle est le composant permettant de valider si oui ou non les modèles produits par la transformation testée sont ceux attendus. Cela revient à effectuer une comparaison de modèle entre : les modèles produits par la transformation de modèle testée et les modèles attendus. Dans [85], les auteurs identifient le problème de la construction de l'oracle pour le test des transformations de modèles comme un problème de comparaison de modèle. Afin de construire l'oracle, ils utilisent un algorithme de comparaison inspiré de celui des techniques de comparaison de graphe. Ils proposent aussi un framework permettant d'organiser le test de transformations de modèles et présente cette comparaison, ainsi que les différentes étapes du test d'une transformation de modèles sur un exemple [84, 83].

La comparaison de modèles est une tâche complexe, en effet, en considérant les modèles comme des graphes d'objets, leur comparaison atteint la complexité d'un algorithme NP-complet. Afin de simplifier le problème, des algorithmes sont proposés par [138, 83] en exploitant les méta-modèles et les structures qu'ils définissent. La comparaison proposée est donc purement structurelle. Afin d'alléger la comparaison du modèle, Kolosov et al. proposent dans [77] une alternative à la comparaison *complète* du modèle attendu avec le modèle de sortie de la transformation testée. Ils définissent alors un ensemble de règles pour la comparaison, chargé d'effectuer des comparaisons seulement entre certains éléments et certains attributs pour vérifier, par exemple, l'égalité de nom. Actuellement, des outils, comme EMFCompare [40], tentent d'implémenter la comparaison structurelle de modèles.

Les retours fournis par la comparaison de modèle et donc, par l'oracle, sont des informations qui peuvent aider à la compréhension des erreurs dans la transformation de modèle. S'appuyant sur ce fait, dans [26, 25], les auteurs s'intéressent aux types de différences retournés en cas d'inégalité entre les modèles. Ils se servent de ces informations pour guider la co-évolution de modèle et proposent de montrer les résultats de la comparaison directement sur les modèles comparés. Pour ajouter ces informations sur les modèles, les auteurs proposent donc d'étendre le méta-modèle des modèles qu'ils comparent en ajoutant de nouvelles méta-classes correspondants aux différences qu'il est possible de trouver, c'est-à-dire, *changement*, *ajout* et *suppression*.

9.5 CONCLUSIONS

Dans ce chapitre, nous avons vu quelques travaux se rapportant au test de transformations de modèles ainsi que les différentes problématiques intervenant dans ce processus. Parmi les problématiques existantes, nous en avons suivi 3, à savoir : la génération du jeu de tests (génération et qualification), la localisation d'erreur et la construction de l'oracle.

De manière générale, nous pouvons constater que beaucoup de travaux se portent sur la génération du jeu de modèles de test ainsi que sur la construction de l'oracle. Ces deux problématiques sont, en effet, primordiale pour pouvoir effectuer correctement le processus de test. Cependant, actuellement, les autres problématiques qui sont la localisation d'erreurs et la qualification du jeu de tests ne présentent que très peu de travaux.

Concernant la localisation d'erreurs, nous avons vu que cette thématique est beaucoup abordée lorsque l'on traite avec des langages de programmation classique, mais quasi inexistante dès lors que l'on travaille avec des langages de transformations de modèles. Les techniques appliquées aux langages de programmations se servent de traces de l'exécution du programme pour détecter un ensemble d'instructions potentiellement porteuse de faute. Dans le cadre d'une transformation de modèle, où la transformation remplace le programme testé, l'utilisation de trace de transformation peut se révéler comme déterminante dans l'activité de localisation d'erreurs pour les transformations de modèles.

D'autre part, la même remarque sur le travail existant pour les langages de transformations classiques comparé au travail effectué sur les transformations de modèles peut aussi s'appliquer à la qualification des jeux de modèles de test. En effet, seuls quelques travaux traitent de la qualification d'un jeu de modèles de test. Parmi les méthodes proposées dans ces travaux, l'analyse de mutation est tout particulièrement intéressante, car elle propose à la fois de qualifier le jeu de modèles de test de manière dynamique et d'améliorer celui-ci si sa qualité est jugée insuffisante. La partie qualification du jeu de modèles de test de l'analyse de mutation possède quelques travaux spécifiques dédiés aux transformations de modèles. Cependant, à notre connaissance, aucun travail ne propose d'automatiser, entièrement ou en partie, l'étape d'amélioration du jeu de modèles de test. Cette activité, qui appartient à part entière à la technique d'analyse de mutation est, pour le moment, effectuée manuellement et, est très coûteuse en terme de temps.

10 LOCALISATION D'ERREURS

En utilisant les approches traditionnelles, les erreurs observées lors de l'exécution de l'application peuvent avoir été introduites par le compilateur ou par le programme source. Dans une approche IDM, la même distinction peut être établie entre les erreurs introduites lors de la construction de la transformation de modèles et les erreurs présentes dans les modèles de conception. Les erreurs présentes dans les transformations peuvent avoir d'importantes conséquences telles que la réplication d'erreurs.

Nous avons vu que dans [1, 37, 70], les auteurs utilisaient des traces d'exécution du programme testé pour fournir un ensemble d'instructions potentiellement fautives. De façon similaire, pour aider le concepteur à identifier des problèmes dans les transformations de modèles, il faut pouvoir fournir une trace de leur exécution, ce qui peut se faire en utilisant la traçabilité de transformations de modèles. En effet, la traçabilité peut être vue comme un « historique » de la transformation liant les éléments manipulés et créés par la transformation [1].

1. *C.f*, chapitre 2

Nous avons déjà défini un mécanisme de traçabilité reposant sur une trace locale et une trace globale [2]. Ces méta-modèles sont suffisamment riches pour supporter des algorithmes dédiés à la résolution du problème de localisation d'erreur. Dans ce chapitre nous proposons un algorithme basé sur la trace de transformation de modèles. À partir d'un élément généré, notre approche identifie les séquences de règles, pour chaque transformation intermédiaire d'une chaîne de transformations, qui ont permis sa génération. Nous avons couplé cet algorithme aux techniques de test pour fournir un algorithme de localisation d'erreurs.

2. *C.f*, chapitre 3

Ce chapitre est organisé comme suit. En section 10.1, nous présentons le procédé de localisation d'erreurs et comment les différents artefacts sont articulés. En section 10.2, nous montrons comment la trace locale et la trace globale sont exploitées pour pouvoir localiser les erreurs dans les transformations. Nous montrons en section 10.3 comment l'algorithme peut être amélioré pour pouvoir proposer une analyse plus fine. En section 10.4, nous proposons une étude de cas permettant de valider les algorithmes, premièrement, sur une unique transformation

avant de nous orienter vers une chaîne de transformations. Finalement, nous concluons en section 10.5.

10.1 LA LOCALISATION D'ERREURS

En automatisant les opérations critiques du développement d'un système, les transformations de modèles permettent un gain de temps et d'effort. Cependant, elles peuvent aussi introduire des erreurs additionnelles si elles contiennent, elles-mêmes, des erreurs. Par conséquent, tester systématiquement et efficacement les transformations est nécessaire pour prévenir la production de modèles erronés.

La figure 90 reprend le processus de test de transformation avec les problématiques associées tel que nous l'avons présenté au chapitre précédent. Dans ce chapitre, nous nous concentrons uniquement sur la partie encadrée du processus. En premier lieux, la transformation testée est exécutée pour les modèles de test générés. Les modèles résultats obtenus sont alors passés à un oracle permettant d'identifier les erreurs dans les modèles de sortie. Ensuite, il faut identifier ce qui, dans la transformation, à provoqué ces erreurs sur le modèle. Une fois la faute identifiée grâce à notre mécanisme de traçabilité[3], celle-ci doit être corrigée. Le processus est ensuite joué à nouveau pour vérifier que les erreurs ont bien été corrigées et que la correction de la transformation n'en a pas ajouté d'autres.

3. Présenté au chapitre 3.

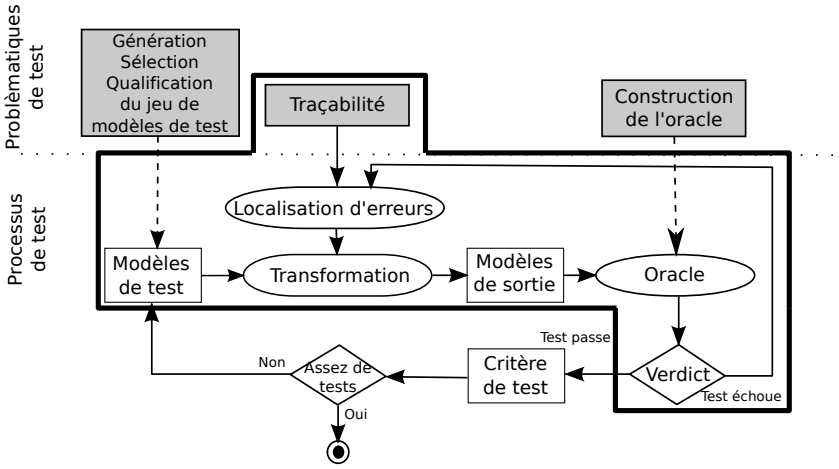


FIGURE 90: Processus de localisation d'erreurs

Les erreurs identifiées dans les modèles de sorties grâce à l'oracle peuvent concerner : des mauvaises valeurs d'attributs ou encore des classes manquantes. Ces erreurs résultent de fautes présentes n'importe où dans les transformations de modèles. La détection de ces fautes est simplifiée si le champ de recherche est réduit à la règle fautive[4]. Cependant, à cause de la non exhaustivité du test et la complexité de la construction d'oracles, le test d'une seule transformation de modèles est souvent laissé de côté au détriment du test de la chaîne de transformations considérée comme une unique entité. En effet, pour une chaîne de transformations, il est parfois plus évident de générer les modèles de test pour la première transformation et de construire l'oracle uniquement pour la dernière transformation de la chaîne plutôt que pour toutes les transformations de la chaîne. Dans cette configuration, il est évident que déterminer manuellement dans quelle transformation

4. *I.e.*, la règle qui porte la faute entraînant la génération de l'erreur dans le modèle de sortie.

se trouve l'erreur est complexe et ce d'autant plus que la chaîne compte de transformations.

10.2 IDENTIFICATION DES RÈGLES ERRONÉES

Notre algorithme de localisation d'erreurs requiert qu'une ait été identifiée dans un modèle de sortie grâce à l'oracle. Afin d'identifier la règle fautive, l'algorithme tente de réduire au maximum le champ d'investigation en mettant en avant les séquences de règles qui ont mené à la production de l'erreur identifiée.

10.2.1 *Pour une transformation de modèles*

Afin d'identifier la séquence des règles ayant modifié ou créé un élément, nous partons de l'hypothèse suivante : Si nous considérons deux éléments A et B du modèle de sortie, créés, respectivement, par les règles *toA()* et *toB()*, si A référence B grâce à une association classique ou une composition, cela suppose que la règle *toA()* a appelé la règle *toB()* ou que la règle *toA* a effectué une opération pour référencer B [5].

En suivant cette hypothèse et en utilisant la trace locale produite pour une transformation, il est possible de réduire le champ de recherche de la totalité des règles de la transformation à un sous-ensemble de règles de la transformation en récupérant l'ensemble des séquences de règles qui ont potentiellement pu produire l'erreur. L'algorithme que nous proposons est composé de 7 étapes pour récupérer le sous-ensemble des règles potentiellement fautives.

1. Sélection de l'élément fautif à étudier
2. Sélection de la trace locale relative à la transformation testée, c'est-à-dire, la trace locale dont le modèle contenant l'élément précédemment identifié est spécifié comme étant modèle de sortie
3. Recherche de l'élément *ElementRef* dans le *destContainer* de la trace locale
4. Identification des règles [6] associées à l'élément par navigation des liens de trace [7]
5. Enregistrement de la règle et de l'objet sur lequel on travail
6. Recherche, dans le *destContainer* de la trace locale, des *ElementRef* qui ont leur *eObject* possédant une référence vers l'élément sur lequel on travail
7. Application récursive de l'algorithme à partir de l'étape 3 sur chaque élément identifié en étape 5

L'appel récursif s'arrête lorsqu'aucun *eObject* ne peut être trouvé à l'étape 6. Cela signifie que la règle créant cet élément est appelée par le point d'entrée de la transformation. Techniquement, cette règle est représentée par le stockage d'un pointeur *null*.

Afin d'illustrer l'algorithme, nous présentons un exemple sur lequel nous allons montrer le sous-ensemble des règles récupérées lorsqu'une erreur est détectée par l'oracle. Cela correspond aux étapes 4 à 6 de l'algorithme de localisation d'erreurs.

La figure 91 présente le modèle de sortie d'une transformation. Dans ce modèle, 6 éléments sont présents : A à F, créés, respectivement à partir de règles toA() à toF(). En supposant que l'oracle retourne une

5. Par exemple, une opération de résolution d'objet en QVT.

6. Élément *RuleRef* de la trace locale.

7. Élément *Link* de la trace locale.

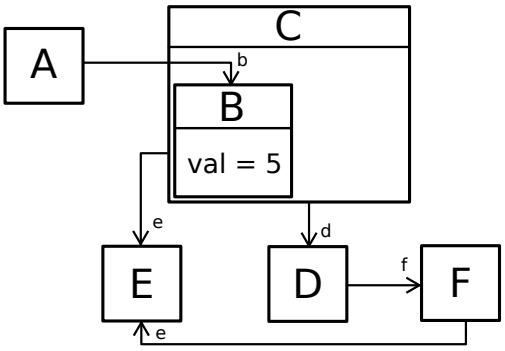


FIGURE 91: Exemple de modèle de sortie

erreur sur l'élément D, l'algorithme commence par identifier et enregistrer la règle créant D, soit toD() (étape 4 et 5). Ensuite, les éléments directement liés à D sont cherchés, soit C (étape 6). Puis la règle créant C : toC() est identifiée, puis enregistrée. Le sous-ensemble des règles recherchées devient donc {toD, toC}. De la même façon, les éléments liés à C sont recherchés. Aucun ne peut être trouvé, ainsi, un pointeur null est stocké. Le sous-ensemble des règles de la transformation est donc {toD, toC, null}. Ainsi, si une erreur est trouvée sur D, il faut chercher l'erreur dans les trois règles retournées par l'algorithme, à savoir : toD(), toC() et *main*, en supposant que *main* est le point d'entrée de la transformation.

10.2.2 *Dans une chaîne de transformations*

L'algorithme que nous venons de présenter est dédié à la localisation d'erreurs dans une simple transformation, mais nous en avons développé une variante adaptée aux chaînes de transformations. Cette fois, ce n'est pas uniquement la succession de règles de transformations qui est sauvegardée par l'algorithme, mais aussi tous les éléments du modèle d'entrée qui ont mené à la création de l'élément fautif dans le modèle de sortie. L'algorithme est de nouveau appliqué sur chacun de ces éléments. Le résultat final est donc l'ensemble des règles potentiellement fautives sur l'ensemble de la chaîne et pour chaque transformation. Cet algorithme travaille en 6 étape importantes :

1. Sélection de l'élément fautif à étudier
2. Sélection de la trace locale relative à la transformation testée
3. Recherche des éléments potentiellement fautif grâce à l'algorithme de localisation d'erreur pour une transformation
4. Enregistrement des éléments récupérés
5. Navigation vers la trace locale précédente dans la chaîne de transformations
6. Réapplication de l'algorithme à partir de l'étape 3 sur chaque élément identifié) l'étape 4

10.3 AMÉLIORATION DE L'ALGORITHME DE LOCALISATION D'ERREURS

Afin d'améliorer l'algorithme de localisation d'erreur pour une transformation de modèles, nous nous sommes penchés sur les retours

fournis par l'oracle. Ces retours doivent être analysés pour pouvoir déterminer plus précisément où la faute se situe dans la transformation. Comme nous l'avons vu, les résultats de l'oracle donnent déjà un point d'entrée à notre algorithme de localisation d'erreurs. Cependant, lorsque les éléments des modèles sont fortement connexes entre eux, la recherche peut entraîner une récupération d'un sous-ensemble de règles contenant presque l'ensemble des règles de la transformation. Dans cette section, nous allons montrer comment le type d'erreurs données par l'oracle permet de restreindre ce sous-ensemble et d'aider au *debug* de la transformation en plus de la localisation plus précise de l'erreur.

10.3.1 *Les types d'erreurs rencontrées*

Les type d'erreurs reportées par l'oracle sont relatifs à la structure du modèle produit. Exactement 5 types d'erreurs ont été identifiés à partir des travaux menés sur les oracles [83, 77] :

1. élément manquant ; indiquant qu'un élément devant apparaître dans le modèle de sortie n'y est pas,
2. élément ajouté ; indiquant qu'un élément ne devant pas apparaître dans le modèle de sortie est présent,
3. élément déplacé ; indiquant qu'un élément ne se trouve pas à la place où il devrait être, structurellement parlant, cela signifie que cet élément n'appartient pas à la bonne relation de composition,
4. valeur d'attribut modifié ; indiquant qu'une valeur d'attribut est différente de celle attendue, voir inexistante,
5. référence modifiée ; indiquant qu'une référence pointe vers un élément inattendu (*i.e.*, vers aucun élément, vers un élément en trop dans une collection ou vers un mauvais élément).

L'oracle que nous avons utilisé nous reporte les erreurs sous forme d'un couple, $(\texttt{type d'erreur}, \{\texttt{éléments}\})$. Dans ce couple, l'ensemble d'éléments sont les éléments sur lesquels l'erreur est relevée et dépendent du type d'erreur rencontré :

- Dans le premier cas, l'oracle renvoie, à la fois, l'élément censé contenir l'élément manquant ainsi que le type de l'élément manquant.
- Dans le troisième cas, l'oracle renvoie, à la fois, l'élément déplacé et l'élément qui aurait dû contenir l'élément déplacé.
- Dans les autres cas, l'oracle renvoie uniquement l'élément sur lequel l'erreur a été relevée.

10.3.2 *Réduction du champ de recherche des règles potentiellement fautives*

Selon les types d'erreurs, nous essayons de « découper » la séquence des règles trouvées par l'algorithme précédent en deux ensembles, celui des règles les plus potentiellement fautives et celui des règles les moins potentiellement fautives. Il est à noter que les propositions que nous faisons ici font état des différentes observations que nous avons effectuées lors des expérimentations. Nous émettons l'hypothèse suivante : de manière générale, dans la plus part des cas, l'erreur provient : soit de la règle créant l'élément pointé elt par l'oracle, soit de la règle créant les éléments du modèle de sortie qui sont liées à

l'élément pointé par l'oracle. Afin d'illustrer les différents cas et sur quels éléments l'algorithme est lancé, nous utilisons le modèle présenté en figure 91, représenté à nouveau en figure 92, comme modèle de sortie attendu d'une transformation hypothétique. Nous considérons que ces éléments sont contenus par un élément root qui est racine du modèle.

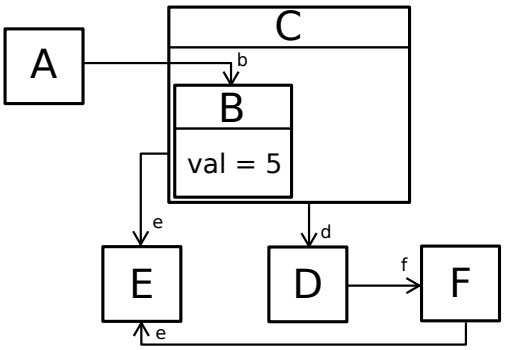


FIGURE 92: Modèle de sortie exemple

Nous considérons aussi que les règles qui ont créé les éléments A, B, C, D, E et F sont les règles toA(), toB(), toC(), toD(), toE() et toF() respectivement. Pour les cas suivants, nous appelons elt l'élément problématique provenant du couple retourné par l'oracle et parent tout élément directement lié à elt.

ÉLEMENT MANQUANT : la figure 93 reprend le modèle exemple avec l'élément B manquant. Il faut noter que dans cet exemple, la référence b est aussi modifiée, mais que pour l'explication, nous ne nous concentrons ici que sur l'élément manquant.

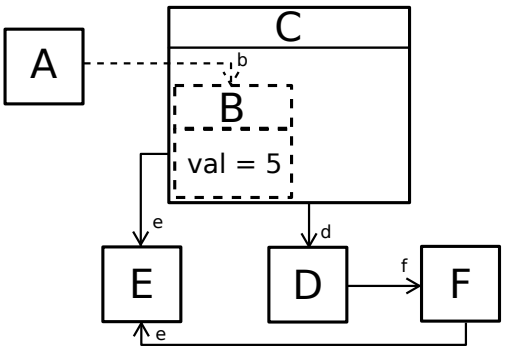


FIGURE 93: Modèle de sortie avec élément manquant

L'élément elt (B) n'existant pas, l'oracle renvoie l'élément parent censé contenir l'élément manquant (ici C). L'algorithme est donc lancé à partir de C. Les règles récupérées sont donc la règle créant C, puis le point d'entrée de la transformation, ce qui donne la séquence {toC, null}. L'élément manquant peut impliquer un filtrage un peu trop agressif, une garde trop exigeante ou même un oubli d'appel de règle[8]. Quoi qu'il en soit, c'est la règle censée contenir l'élément qui effectue, normalement, l'appel de la règle. L'ensemble est donc découpé en sous-ensembles avec la règle de l'élément parent contenant l'élément dans le sous-ensemble des règles les plus potentiellement fautives. L'ensemble final obtenu est donc le suivant : {{toC}, {null}}.

8. Détaillé en section 3.

ÉLEMENT AJOUTÉ : la figure 94 reprend le modèle exemple avec un élément B en plus dans la collection b.

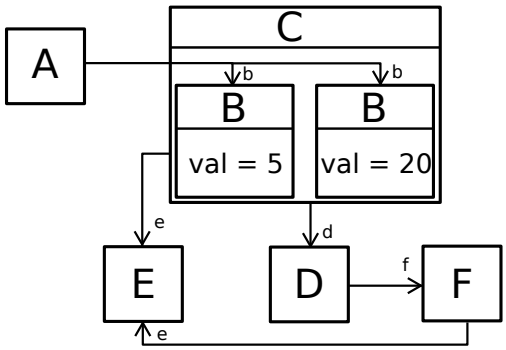


FIGURE 94: Modèle de sortie avec élément supplémentaire

L'élément elt (l'élément B possédant la valeur de son attribut *val* à 20) est en trop dans le modèle, cela veut dire qu'il provient, probablement, d'une exécution de règle en trop. Ceci peut provenir d'un filtrage manquant ou d'une garde de règle manquante. L'algorithme est lancé à partir de l'élément B reporté par l'oracle et le sous-ensemble récupéré est $\{toB, toC, toA, null\}$. Cette fois, le sous-ensemble des règles est réordonnée avec la règle créant l'élément B et les règles de ses *parents* (ici, C et A) comme les règles les plus potentiellement fautives. Le sous-ensemble des règles potentiellement fautives est donc restreint à $\{toB, toC, toA\}$ et l'ensemble final retourné est $\{\{toB, toC, toA\}, \{null\}\}$

ÉLEMENT DÉPLACÉ : la figure 95 reprend le modèle exemple avec l'élément B déplacé à la racine du modèle.

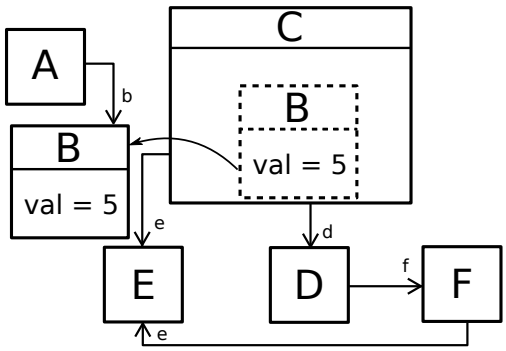


FIGURE 95: Modèle de sortie avec un élément déplacé

L'élément elt (B) appartient au mauvais élément (ici, root). L'appel à la règle créant l'élément est donc mal placé, ou une opération référençant l'élément est manquante. C'est donc dans la règle contenant l'élément mal placé qu'il faut chercher l'erreur. L'algorithme est lancé à partir de B et de C, l'ensemble trouvé est donc $\{toB, toA, null, toC\}$. Cette fois, il est possible de réorganiser les règles trouvées avec deux règles jugées comme potentiellement plus fautives : soit la règle créant l'élément contenant B, soit null représentant le point d'entrée de la transformation et à la règle créant l'élément censé contenir B, soit toC. L'ensemble final est donc : $\{\{toB, null\}, \{toA, toC\}\}$.

VALEUR D'ATTRIBUT MODIFIÉ : la figure 96 reprend le modèle exemple avec la valeur *val* de l'élément B modifiée.

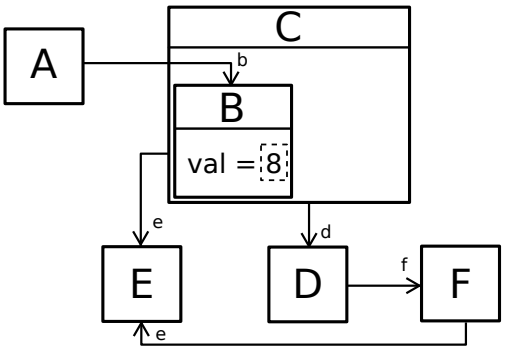


FIGURE 96: Modèle de sortie avec une valeur d'élément modifiée

La recherche est lancée à partir de l'élément elt (B). L'ensemble retrouvé est donc {toB, toA, null}. Cependant, si la valeur de l'attribut est erronée, c'est donc dans la règle créant l'élément et modifiant/créant l'attribut qu'il faut chercher l'erreur. L'ensemble final donné est donc : {{toB}, {toA, null}}. Évidemment, si l'attribut est calculé par une fonction tierce, c'est cette fonction qu'il faut vérifier.

RÉFÉRENCE MODIFIÉE : la figure 97 reprend le modèle exemple avec la référence *e* de F supprimée.

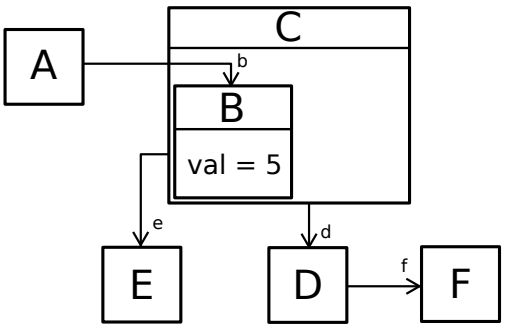


FIGURE 97: Modèle de sortie avec une référence d'élément modifiée

Les erreurs portant sur les références sont semblables à celles portant sur les attributs et le sous-ensemble récupéré par l'algorithme est réduit de la même façon. L'algorithme est donc lancé à partir de elt (F) et le sous-ensemble {toF, toD, toC, null} est retourné. Cette fois, de même que pour lorsque l'attribut de l'élément est modifié, c'est dans la règle créant l'élément portant la référence qu'il faut chercher l'erreur, soit le nouveau sous-ensemble : {{toF}, {toD, toC, null}}.

Dans cette sous-section, nous avons vu qu'il est possible d'affiner la localisation d'erreurs et, ainsi contraindre le sous-ensemble de règles retourné par l'algorithme. Cependant, les exemples montrés n'ont fait état, à chaque fois, que d'une seule erreur dans le modèle de sortie, or plusieurs erreurs peuvent intervenir sur un même modèle de sortie. Dans ce cas, il suffit de prendre les erreurs les unes après les autres tout en relançant la transformation et l'oracle à chaque modification

de la transformation, car il se peut que certaines erreurs identifiées par l'oracle soient intervenues par effet de bord sur la transformation. Effectivement, il n'est pas rare de trouver des retours de l'oracle indiquant plusieurs erreurs provenant d'une seule faute dans la transformation. Si l'on reprend le modèle exemple de la figure 93, comme nous l'avions évoqué, l'oracle reporte deux erreurs. La première sur l'élément C du modèle en indiquant que l'élément B est manquant (donc une faute sur la règle toC), mais aussi une erreur sur l'élément A en précisant que la référence a est modifiée (ce qui implique une faute sur la règle créant A, soit : toA). Dans ce cas, la résolution de la première erreur entraîne une résolution de la seconde erreur par effet de bord, à moins que, effectivement, une faute existe dans toA.

10.4 ÉTUDE DE CAS

L'étude de cas que nous montrons ici s'inscrit dans une des chaînes de compilation proposée par la plate-forme GASPARD2. La chaîne que nous visons est la même que celle déjà décrite au chapitre 6. Cette chaîne de compilation produit du code OPENCL pour GPUs à partir de modèles UML profilés MARTE. Dans cette section, nous allons montrer l'efficacité de notre algorithme de localisation d'erreur à déterminer la règle contenant la faute menant à l'erreur dans le modèle de sortie.

Dans un premier temps, nous allons appliquer l'algorithme sur une unique transformation en essayant d'y corriger les erreurs. Nous procéderons à l'exécution de l'algorithme sur une chaîne de transformations de modèles complète pour tenter d'y situer les fautes. Pour les deux cas présentés, nous vérifierons à chaque fois que la faute est bien présente les règles retournées par notre algorithme de localisation d'erreur.

10.4.1 *Exemple de localisation d'erreurs avec la transformation* « UML *vers* MARTE »

Comme étude de cas, nous utilisons la transformation de modèle UML vers MARTE déjà présentée au chapitre 3. Comme modèle d'entrée, nous avons réutilisé le modèle du *DownScaler* que nous avons aussi présenté au chapitre 6. Pour l'étude, nous avons pris une version erronée de la transformation créée lors de son développement. La version actuelle et corrigée de la transformation sert de transformation témoin. C'est à partir de cette transformation que nous obtenons le modèle de sortie attendue avec lequel va être comparé le modèle de sortie de la transformation erronée. La transformation corrigée nous sert aussi à vérifier que la faute est bien contenue dans les règles potentiellement fautives retournées par notre algorithme. Mis à part l'utilisation de la transformation corrigée comme transformation témoin, pour les expérimentations, nous avons ensuite suivi le procédé classique de test de transformation décrit en figure 90, à savoir :

1. exécuter la transformation avec un modèle de test (ici le modèle du *downscaler* présenté au chapitre 6 et 8),
2. identifier l'erreur grâce à l'oracle,
3. localiser la règle contenant l'erreur dans la transformation et la corriger,
4. exécuter à nouveau la transformation pour vérifier si l'erreur a bien disparu.

La version de la transformation UML vers MARTE que nous utilisons pour notre expérimentation est une version ancienne conservée dans notre *repository* et possédant des erreurs qui ont été identifiées et corrigées dans les versions actuelles des chaînes de compilation. La transformation UML vers MARTE contient 84 règles de transformation [9] écrites en QVTo. Dans cette transformation, exactement, 5 erreurs comprises dans 4 règles ont été corrigées à partir de la version que nous étudions ici pour arriver à la version correcte actuellement utilisée. La table 5 résume les règles portant les fautes en précisant pour chacune leur nom et le nombre de fautes qu'elles contiennent.

9. Sans compter les *query*.

NOM DE LA RÈGLE	NOMBRE DE FAUTES
toPortImplements	2
toMainInstance	1
resolveAllocateMapping	1
toDeployModel	1

TABLE 5: Résumé des règles fautives

Nous allons voir comment les champs d'investigation sont réduits en utilisant notre algorithme pour ne sélectionner que les règles potentiellement fautives. Dans un premier temps, nous avons exécuté la transformation contenant les fautes avec le modèle UML de *DownScaler* en entrée. La transformation a produit, à la fois, le modèle MARTE de sortie et un modèle de trace locale. Le modèle MARTE est ensuite donné à un oracle pour déterminer s'il contient des erreurs dans le modèle de sortie. Comme oracle, nous avons utilisé l'outil de comparaison de modèle EMFCOMPARE et nous avons comparé le modèle produit par la transformation fautive avec le modèle produit par la transformation corrigée. Les résultats de l'oracle sont donnés dans la table 6. La table donne pour chaque entrée le type d'erreur rencontré, l'élément sur lequel l'erreur a été reportée et le nombre d'erreurs reportées sur l'élément [10].

10. Par exemple, la première erreur et reportée deux fois sur deux références différentes du même élément.

TYPE D'ERREUR	ÉLEMENT PORTANT L'ERREUR	QUANTITÉ
Modification Référence	fromip_ytovip_y : PortImplement	
	attribut : target	1
	attribut : source	1
Manque	Downscaler : DeploymentModel	
	PortImplement	28
Modification Référence	Fromifctoactuator : Distribute	
	référence : target (elt. en sus.)	1
Modification Référence	Fromifgtosensor : Distribute	
	référence : target (elt. en sus.)	1
Modification Référence	Fromrhftoproc_unit1 : Distribute	
	référence : target (elt. en sus.)	1

TABLE 6: Résumé du nombre, du types d'erreurs et de l'élément sur lequel l'erreur est trouvée

En utilisant le modèle de *DownScaler* comme modèle d'entrée, l'oracle a décelé 33 erreurs dans le modèle MARTE de sortie. Parmi ces 33 erreurs :

- 28 élément de type *PortImplement* ont été identifiés comme manquant sur l'élément *Downscaler* de type *DeploymentModel*,
- 2 références : *source* et *target* ont été identifiées comme modifiées sur l'élément *fromip_ytovip_y* du type *PortImplement*,
- 3 références *target* ont été identifiées comme modifiées pour des éléments de type *Distribute*.

Afin de déterminer les ensembles de règles potentiellement fautives, nous appliquons l'algorithme présenté en page 173 sur chacun des éléments porteur d'erreur. Nous commençons, par la première erreur reportée, à savoir l'erreur reportée sur l'élément *Fromip_ytovip_y : PortImplement*. L'élément en question est donc sélectionné dans le modèle de sortie et l'algorithme est lancé en prenant en paramètre la trace locale générée et l'élément fautif. L'ensemble de règles retourné est le suivant : $\{\text{toPortImplements}, \text{toDeployModel}, \text{toModel}, \text{main}\}$, ce qui restreint le champ d'investigation à 4 règles plutôt qu'à 84. L'ensemble est ensuite découpé en sous-ensembles pour placer les règles les plus susceptibles de porter la faute de côté. Les sous-ensembles obtenus sont, dans ce cas : $\{\{\text{toPortImplements}\}, \{\text{toDeployModel}, \text{toModel}, \text{main}\}\}$. C'est donc la règle toPortImplements qui est la plus susceptible de porter la faute. Le listing 10.1 présente la règle en question.

```
1  mapping UML::Dependency::toPortImplements() : DeployMarte::
       PortImplements
2  {
3    name := self.name;
4    result.target := self.source.resolveone(GCM::FlowPort);
5    result.source := self.target.late resolveone(GCM::FlowPort);
6  }
```

Listing 10.1: Règle toPortImplement

Si l'on regarde les lignes 4 et 5, on peut se rendre compte que la référence *target* est calculée à partir de la *source* d'un élément d'entrée (ligne 4) et que la référence *source* est calculée à partir de la *target* d'un élément d'entrée. L'erreur provient de ces deux affectations qui ont été inversées. Nous les inversons donc pour que la référence *target* soit calculée en fonction de la référence *target* de l'élément d'entrée et que la référence *source* soit calculée en fonction de la référence *source* de l'élément d'entrée.

Une fois que la modification est effectuée, la transformation est lancée de nouveau et l'oracle est interrogé pour vérifier que la modification de la transformation a bien corrigé l'erreur et qu'aucune autre erreur n'a été introduite. Comme précédemment, le résultat de l'oracle est présenté dans la table 7. Nous pouvons voir que l'erreur portant sur le *PortImplement* a bien été corrigée, mais qu'aucune autre des erreurs reportées n'a été impactée par notre modification.

Nous nous occupons maintenant d'un des *Distribute* dont la référence *target* a été modifiée et nous lançons l'algorithme à partir de l'élément *Fromifctoactuator : Distribute* dans le modèle de sortie. L'ensemble des règles retourné est le suivant : $\{\text{resolveAllocateMapping}, \text{toModel}, \text{main}\}$, ce qui, cette fois, réduit le champ d'investigation à 3 règles plutôt qu'à 84. Comme pour le cas précédent, après découpage des règles en

TYPE D'ERREUR	ÉLEMENT PORTANT L'ERREUR	QUANTITÉ
Manque	Downscaler : DeploymentModel PortImplement	28
Modification Référence	Fromifctoactuator : Distribute référence : target (elt. en sus.)	1
Modification Référence	Fromifgtosensor : Distribute référence : target (elt. en sus.)	1
Modification Référence	Fromrhftoproc_unit1 : Distribute référence : target (elt. en sus.)	1

TABLE 7: Résumé du nombre, du types d'erreurs et de l'élément sur lequel l'erreur est trouvée

sous-ensembles, la règle donnée comme étant la plus susceptible d'être la règle fautive est la règle $resolveAllocateMapping$. Le listing 10.2 présente une partie du code de la règle en question. La ligne 10 montre le code relatif à la référence *target*. Cependant, en regardant juste le code, il est difficile de savoir si la règle contient bien la faute et comment la corriger. En regardant plus en détail le résultat de l'oracle, il est précisé que la référence *target*, qui est une collection, possède un élément en trop. Afin de réduire le nombre d'éléments assignés à la collection *target*, le mot clef *resolve* (ligne 10) est remplacé par *resolveOne* qui effectue la même action que *resolve*, mais en ne retournant qu'un seul élément plutôt que plusieurs.

```
1   mapping UML::Abstraction::resolveAllocateMapping() : Alloc::
        Allocate
2   {

10    result.target+=self.target.resolve().oclAsType(MM::
          ModelElement);

12  }
```

Listing 10.2: Règle $resolveAllocateMapping$

Une fois que la modification est effectuée, la transformation est ensuite rejouée et l'oracle interrogé. Les retours de l'oracle sont donnés dans la table 9. Toutes les erreurs sur les *Distributes* ont disparues et il reste uniquement les erreurs de type *manque*. En regardant plus en détail les retours de l'oracle, il est précisé que les éléments manquant sont des éléments de type *PortImplement*. L'algorithme

TYPE D'ERREUR	ÉLEMENT PORTANT L'ERREUR	QUANTITÉ
Manque	Downscaler : DeploymentModel PortImplement	28

TABLE 8: Résumé du nombre, du types d'erreurs et de l'élément sur lequel l'erreur est trouvée

est donc lancé à partir de l'élément *Downscaler : DeploymentModel* du modèle de sortie. L'ensemble des règles retournées par l'algorithme est : $\{toDeployModel, toModel, main\}$. Le champ de recherche

a donc été réduit à 3 règles potentiellement fautives. Après séparation des règles en plusieurs sous-ensemble, l'ensemble final retourné est : $\{\{\texttt{toDeployModel}\}, \{\texttt{toModel}, \texttt{main}\}\}$. L'erreur est donc cherchée, en priorité, dans la règle toDeployModel. Le listing 10.3 présente le code de la règle. La ligne 8 indique qu'un seul élément est pris en entrée de la règle toPortImplement, règle créant des éléments de type *PortImplement*. En effet, l'opérateur '!' (ligne 8) sélectionne un élément d'un type particulier. Dans cette assignation, il est précisé qu'un seul élément de type *UML : :Dependency* est sélectionné et transformé en *PortImplement*. La ligne est donc tout simplement modifiée en enlevant l'opérateur '!'.

```
1   mapping UML::Package::toDeployModel() : DeployMarte::
        DeploymentModel {

8     ownedElement += self.ownedElement![UML::Dependency]->map
          toPortImplements();

18  }
```

Listing 10.3: Règle toDeployModel

Une fois encore, la transformation est rejouée et l'oracle interrogé pour vérifier l'efficacité de la modification. Les erreurs relevées par l'oracle sont présentées dans la table 9. L'erreur concernant les 28 manques d'éléments de type *PortImplement* a, effectivement, disparue, mais 41 éléments de type *PortImplement* sont marqués comme étant en surplus dans le modèle de sortie. L'algorithme de localisa-

TYPE D'ERREUR	ÉLEMENT PORTANT L'ERREUR	QUANTITÉ
Ajout	Fromin_yhftomem : PortImplement	1
Ajout	Fromcons_gtomemory : PortImplement	1
...	...	...
Ajout	Fromout_utomem : PortImplement	1
Ajout	Fromcons_rtomemory : PortImplement	1
Total		41

TABLE 9: Résumé du nombre, du types d'erreurs et de l'élément sur lequel l'erreur est trouvée

tion d'erreur est donc lancé à partir du premier élément retourné : *Fromin_yhftomem : PortImplement*. L'ensemble de règle retourné est le suivant : $\{\texttt{toPortImplement}, \texttt{toDeployModel}, \texttt{toModel}, \texttt{main}\}$. Sur cet ensemble de 4 règles, la séparation en sous-ensemble de règles donne : $\{\{\texttt{toPortImplement}, \texttt{toDeployModel}\}, \{\texttt{toModel}, \texttt{main}\}\}$, ce qui place la règle toPortImplement et toDeployModel comme étant les règles les plus suspectes. Les raisons pour laquelle des éléments peuvent être « en surplus » dans le modèle de sortie sont, soit une garde manquante [11] dans la règle toPortImplement, soit un filtrage manquant dans toDeployModel. Si nous regardons la transformation corrigée, nous voyons que la correction a été effectuée dans la règles toPortImplement où une garde a été ajoutée. Étant donné la complexité de la garde, nous ne la détaillons pas ici, mais nous la considérons comme une modification que nous effectuons sur la transformation fautive. Une fois la modification terminée, la transformation est, à nouveau, exécutée. Cette fois, l'oracle ne retourne plus d'erreurs sur le modèle de sortie, la transformation est, potentiellement, corrigée.

11. Une section *when* dans un *mapping*.

À travers cet exemple, nous avons montré que grâce à notre algorithme de localisation d'erreurs reposant sur l'utilisation de la trace locale, nous avons pu réduire, pour une erreur donnée, les champs d'investigation de la règle fautive à 3 ou 4 règles sur les 84 règles que contient la transformation. Les modifications à effectuer sont alors plus simples à déterminer pour le testeur et le temps de recherche des erreurs est raccourci. Néanmoins, pour notre expérimentation, nous avions sélectionné une transformation contenant exactement 5 fautes. Or, nous n'en avons corrigé ici que 4 et pourtant, l'oracle n'a pas observé d'erreurs dans le modèle de sortie. Cela est dû au modèle d'entrée utilisé pour les tests qui n'est pas assez sensible pour mettre en évidence une erreur dans le modèle de sortie, c'est-à-dire qu'il ne possède pas les caractéristiques suffisantes pour pouvoir entraîner une erreur dans les modèles de sortie. Pour assurer la bonne correction de la transformation, il faudrait alors utiliser un nouveau modèle de test.

10.4.2 *Localisation d'erreurs dans une chaîne de transformations*

Dans la sous-section précédente, nous avons montré en détail les ensembles de règles potentiellement fautives retrouvés lorsqu'une erreur est relevée par l'oracle dans le modèle de sortie. Nous allons voir ici quel sont les ensembles de règles considérées comme potentiellement fautives qui sont récupérées pour une chaîne de transformations complète lorsque l'oracle détecte des erreurs dans le modèle de sortie de la chaîne et si la règle fautive appartient bien à ces ensembles.

Pour les expérimentations, nous utilisons la chaîne de transformations GASPARD2 ciblant le langage OPENCL que nous avons déjà présenté au chapitre 6. Évidemment, nous ne nous intéressons pas à la transformation de modèle vers code. La chaîne de transformations est représentée à la figure 98.

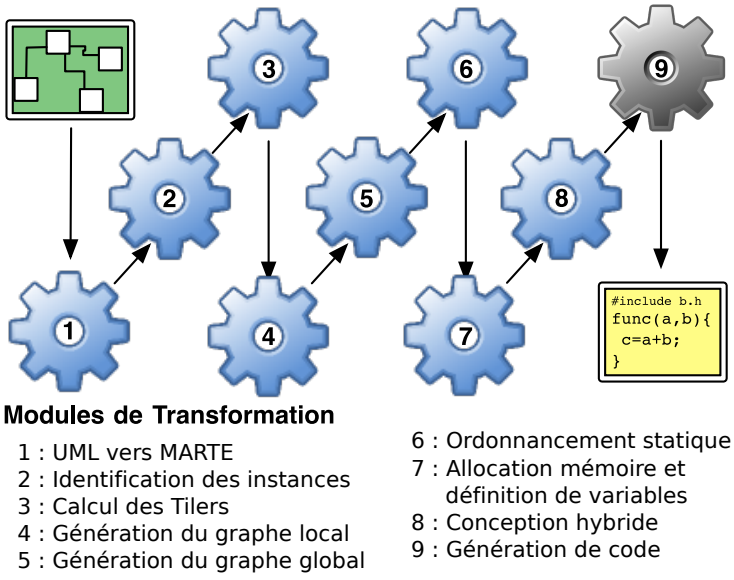


FIGURE 98: Chaîne de compilation GASPARD2 vers du code OPENCL

Pour les expérimentations nous avons récupéré une ancienne version de la transformation numéro 5 dans notre *repository* qui servira de transformation témoin. Cette transformation, calculant les dépendances de

tâches sur la globalité de l'application, possède exactement 1 faute sur la règle toTaskIntraDependency, corrigée dans les versions les plus récentes. Cette faute entraîne l'initialisation de la mauvaise référence pour les éléments de type *TaskIntraDependency*. La conséquence pour la suite de la chaîne de transformations est un mauvais ordonnancement des tâches par la transformation numéro 6.

La chaîne de compilation est donc lancée, toujours en utilisant le modèle de *DownScaler* comme modèle d'entrée. Pendant l'exécution de la chaîne, plusieurs traces locale sont générées, ainsi qu'une trace globale. Comme nous l'avons détaillé dans ce chapitre, ces traces sont utilisées par l'algorithme de localisation d'erreur. Une fois la chaîne exécutée, le modèle de sortie de la chaîne est donné à l'oracle qui va déterminer quelles sont les erreurs présentes et leurs types. Le résumé de ces erreurs est reporté dans la table 10. Au total, 22 erreurs ont été reportées sur 11 éléments. Les erreurs sont, à chaque fois, des modifications de références.

TYPE D'ERREUR	ÉLEMENT PORTANT L'ERREUR	QUANTITÉ
Modification Référence	TaskIntraDependency Référence : from Référence : to	 1 1
Modification Référence	TaskIntraDependency Référence : from Référence : to	 1 1
...	...	...
Modification Référence	TaskIntraDependency Référence : from Référence : to	 1 1
Total		22

TABLE 10: résumé du nombre, du types d'erreurs et de l'élément sur lequel l'erreur est trouvée (Extrait)

L'algorithme de localisation d'erreurs est lancé à partir d'un élément *TaskIntraDependency*. Le résultat de l'algorithme est donné dans la table 11 qui résume le nombre de règles identifiées comme potentiellement fautives dans chaque transformation de la chaîne, le nombre de règles retournées comme étant les plus potentiellement fautives ainsi que le nombre de règles pour chaque transformation de la chaîne.

Afin de situer précisément dans quelle transformation de la chaîne la faute se trouve, les règles sont inspectées en partant de la transformation 6. En effet, il est plus simple et plus rapide de démarrer la recherche précise de la faute par les transformations contenant le moins de règles potentiellement fautives car cela permet de rapidement évincer certaines transformations. Dans notre cas, rien de suspect n'est relevé dans la transformation 6 et la recherche se poursuit dans la transformation 5.

Si nous vérifions l'ensemble de règles retourné pour la transformation numéro 5 nous avons {{toTaskIntraDependency, toGlobalGraph}, {addEndDependency, addGlobalGraph, toGlobalTask, main}} et plus particulièrement {toTaskIntraDependency, toGlobalGraph} pour l'ensemble des règles les plus potentiellement fautives qui contient bien la règle toTaskIntraDependency que nous savions fautives grâce à

NUM. DE TRANSFO.	RÈGLES TROUVÉES	+ POT. FAUTIVE	RÈGLES TOTAL
1	11	4	84
2	6	4	8
3	7	3	10
4	2	2	7
5	6	2	13
6	2	1	2
7	0	0	14
8	0	0	15
Total	34	16	150

TABLE 11: Résumé du nombre, du types d'erreurs et de l'élément sur lequel l'erreur est trouvée

la transformation témoin. Nous observons que le champ de recherche a été réduit de 150 règles réparties sur 8 transformations à 34 règles réparties sur 6 transformations et finalement à 16 règles réparties sur 6 transformations. En effet, l'algorithme n'a relevé aucune règle potentiellement fautive dans les transformations 7 et 8. C'est l'utilisation des transformations localisées qui explique ce résultat. Comme nous l'avons vu aux chapitres 1 et 3, les transformations localisées sont implémentées par des transformations *sur place*. Dans une chaîne, cela signifie que lorsque des éléments sont créés dans un modèle à une étape de la chaîne, si ceux-ci ne sont pas supprimés, ils seront toujours présent dans le modèle transformé jusqu'à la fin de l'exécution de la chaîne de transformation. Ceci explique la présence des éléments erronés dans le dernier modèle de la chaîne. Notre algorithme appliqué sur une chaîne de transformations retourne bien, parmi l'ensemble des règles les plus fautives de la chaîne, la règle portant effectivement la faute.

10.5 CONCLUSIONS

Dans ce chapitre, nous nous sommes intéressés à l'activité de localisation d'erreur dans les transformations de modèles à base de règles, pour le test de transformation de modèles. Afin d'aider le développeur des transformations à déterminer les fautes dans sa transformation, nous avons proposé un algorithme reposant sur une hypothèse forte et sur l'utilisation de la trace locale permettant d'obtenir un sous ensemble de règles potentiellement fautives de la transformation. Nous avons décliné ce premier algorithme de localisation d'erreurs pour l'utiliser sur une chaîne de transformations. Cette fois, l'algorithme renvoie par transformation les sous ensembles de règles potentiellement fautives.

Nous avons ensuite proposé une manière de réorganiser le sous ensemble obtenu pour fournir en première position les règles étant plus à même de porter la faute. Cette réorganisation dépend des capacités de l'oracle et des informations qu'il transmet lorsqu'il trouve des erreurs. De cette façon, il est possible de cibler plus rapidement certaines règles de transformation plutôt que d'autres.

Afin d'éprouver notre approche, nous l'avons utilisé sur un cas d'étude. Nous avons dans un premier temps appliqué notre approche sur une transformation de modèles contenant 5 erreurs. À chaque étape de la correction de la transformation, l'algorithme de localisation d'er-

reur nous fournissait un sous ensemble de règles dans laquelle la règle effectivement fautive se trouvait. Nous avons ensuite réutilisé l'algorithme sur une chaîne de transformations dont une des transformations contenait une erreur. Nous avons montré que pour une chaîne de 8 transformations contenant au total 150 règles de transformations, nous étions capables de réduire le sous ensemble de règles potentiellement fautives à 34 à vérifier sur 6 transformations plutôt que 8.

Néanmoins, pour le cas d'étude sur une transformation de modèles, sur les 5 erreurs existant dans la transformation, l'oracle et l'algorithme de localisation d'erreur n'ont permis de trouver et corriger que 4 fautes. C'est le modèle de test utilisé qui est à mettre en cause ici. En effet, celui-ci n'est pas suffisamment sensible pour pouvoir mettre en évidence une erreur dans le modèle de sortie. Ceci montre l'importance de la qualification des données de test pour assurer que celles-ci sont suffisantes pour assurer le test complet de la transformation.

11 ANALYSE DE MUTATION

Au chapitre 9, nous avons vu que les transformations de modèles peuvent être assimilées à des programmes et peuvent être testées en tant que tels. Cependant, les structures des données qu'elles manipulent (des modèles conformes à des méta-modèles) impliquent aussi l'utilisation d'opérations spécifiques que l'on ne retrouve pas en programmation traditionnelle, comme les navigations dans les modèles ou les filtrages sur des collections d'éléments. Ainsi, les erreurs qui peuvent être trouvées dans les transformations peuvent aussi bien être des erreurs classiques que des erreurs spécifiques. Par exemple, le développeur de la transformation peut avoir navigué une mauvaise association entre deux classes et, ainsi, manipuler les mauvaises instances de classes. L'émergence du paradigme objet a impliqué une évolution des techniques de vérification [87]. De manière similaire, ces techniques de vérification doivent être adaptées aux spécificités des transformations de modèles. Cependant, cela implique la rencontre de nouveaux

problèmes relatifs à la génération, la sélection et la qualification des données d'entrée [1] des techniques de test.

1. des modèles d'entrée

Comme nous l'avons vu au chapitre 9, il existe différentes techniques de test. Dans ce chapitre, nous nous concentrons uniquement sur l'analyse de mutation [33]. Cette technique vise la qualification d'un ensemble de données de test pour détecter les fautes dans un programme sous test. L'analyse de mutation repose sur l'hypothèse suivante : si un ensemble de données de test peut révéler les fautes dans un programme volontairement erroné, il est capable de détecter des fautes involontaires. Pour mettre en œuvre cette technique, des versions erronées du programme, appelés *mutant*, sont systématiquement créées en injectant une seule faute par version. Ces fautes volontairement injectées doivent, bien entendu, être pertinentes et refléter celles qu'un développeur pourrait avoir, involontairement, introduites. L'efficacité d'un ensemble de données de test à révéler les fautes de ces programmes erronés est ensuite évaluée. Si la proportion des fautes détectées [131] est considérée trop basse, de nouvelles données de tests doivent être introduites [92].

Dans [69], les auteurs affirment que peu de travaux s'occupent de l'amélioration de l'ensemble de données de test d'entrée. Dans ce chapitre, nous présentons nos travaux concernant cette étape. Le reste du processus de mutation est seulement rapidement évoqué. En section 11.1, nous commençons par présenter comment l'analyse de mutation a été adaptée aux transformations de modèles. En section 11.2, nous expliquons ensuite comment utiliser la trace pour récupérer de l'information et aider à sélectionner des modèles de test pertinents. En section 11.3, nous montrons comment il est possible de tendre vers une automatisation du processus d'analyse de mutation et plus précisément de l'étape d'amélioration du jeu de tests. Par la suite, en section 11.4, nous présentons notre approche en pratique sur un cas d'étude. Finalement, nous concluons en section 11.5.

11.1 LE PROCESSUS D'ANALYSE DE MUTATION POUR QUALIFIER UN JEU DE DONNÉES DE TEST

Assurer qu'un programme ne contient, sans doute possible, aucune erreur est une tâche difficile et requiert beaucoup de temps et d'expertises. Qualifier un jeu de tests est plus simple et permet de facilement estimer la pertinence du jeu de tests produit. Parmi les techniques existantes, l'analyse de mutation permet de calculer une estimation de la pertinence du jeu de modèles de test utilisé. Si cette estimation est considérée trop basse, le jeu de tests doit être amélioré. Dans ce chapitre, nous présentons les travaux que nous avons publiés dans [5] et [6] et leur extension. Ces travaux relatant de l'aide apportée par les traces de transformations de modèles dans le processus d'analyse de mutation. Dans les sous section suivantes, nous décrivons brièvement le processus d'analyse de mutation [33] et nous présentons pourquoi et comment ce processus est adapté aux transformations de modèles.

11.1.1 *Le processus d'analyse de mutation*

Le processus d'analyse de mutation pourrait être divisé en quatre activités principales comme cela est présenté en figure 99. L'étape préliminaire (activité (a)) correspond, à la fois à la définition d'un jeu

de tests initial que le testeur veut qualifier et à la création de variations du programme, appelé *mutants*, (P_1, P_2, ..., P_k) du programme P testé en lui injectant, à chaque fois, un seul changement atomique. En pratique, chaque changement correspond à l'application d'un seul opérateur de mutation sur P. Ensuite, P et tous les mutants sont successivement exécutés avec chaque donnée du jeu de tests qui doit être qualifié (activité (b)).

Si l'exécution d'un mutant P_i avec une donnée de test diffère de celle de P avec la même donnée de test, ce mutant est dit *tué*. La faute introduite dans ce P_i a été effectivement mise en évidence par la donnée de test. À l'inverse, si un mutant P_j retourne le même résultat que P quelque soit les données de test, ce mutant est indiqué comme étant *vivant*. Une fois tous les mutants exécutés sur toutes les données de test, l'activité (c) calcule le ratio de mutants tués, aussi appelé score de mutation. Si ce ratio est considéré trop bas, cela signifie que le jeu de tests n'est pas assez sensible pour pouvoir mettre en valeur les fautes injectées dans le programme. Dans ce dernier cas, le jeu de tests doit être amélioré (activité (d)) jusqu'à ce que le ratio soit dépassé, que tous les mutants soient tués, ou qu'il reste uniquement les mutants équivalents au programme P [2] [33].

2. Aucune donnée de test ne pourra distinguer P des mutants restés vivant (par exemple, une faute est insérée dans du code mort).

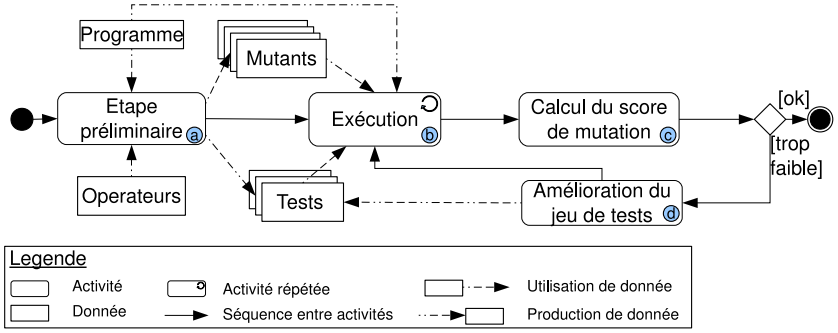


FIGURE 99: Processus d'analyse de mutation

11.1.2 *Un procédé à grande partie manuelle*

Même si une partie du processus d'analyse de mutation est automatique, cela reste un travail complexe et long pour le testeur. La création des mutants peut être automatisée. Cependant, très souvent, les opérations de mutations sont spécifiques à un langage utilisé par le programme testé. Ainsi, pour chaque nouveau langage, les opérateurs de mutation doivent être définis et implémentés. L'exécution du programme P et de ses mutants associés avec les données de test ainsi que la comparaison des sorties du programme, est, évidemment automatique. Cependant, ces tâches sont les seules qui sont automatisées. L'analyse des mutants vivant et l'amélioration du jeu de tests sont, actuellement, manuels. D'une part, l'identification automatique des mutants équivalents est un problème indécidable [33, 104] et, d'autre part, l'amélioration du jeu de tests peut être très complexe. En effet, les fautes injectées qui ne sont pas révélées par les données de test doivent être analysées statiquement et dynamiquement pour pouvoir créer une nouvelle donnée de test qui pourra tuer le mutant considéré.

Le but de ce chapitre est d'aider à l'automatisation de l'amélioration du jeu de tests lorsque le programme testé est une transformation de modèle.

11.1.3 *Adaptation aux transformations de modèles*

Les transformations de modèles peuvent être considérées comme des programmes et, par conséquent, la technique d'analyse de mutation précédemment expliquée peut être utilisée. Cependant, la complexité et les spécificités induites par les structures de données manipulées par les transformations de modèles [3] implique des modifications dans le processus d'analyse de mutation expliquée en section 11.1.1.

3. C'est-à-dire, des modèles conformes à leurs méta-modèles.

Chaque étape du processus d'analyse de mutation doit être adaptée aux transformations de modèles. Cela commence par la génération des modèles de test, abordée en [110]. Comme énoncé précédemment, les opérateurs de mutations doivent être définis et implémentés pour le paradigme de modèles. Dans [90], Mottu et al. ont défini des opérateurs de mutations dédiés aux transformations de modèles indépendamment de tout langage de transformation. Ces opérateurs reposent sur trois opérations d'abstraction liées aux traitements de base d'une transformation de modèles à savoir : la navigation d'une classe à l'autre par leurs relations, le filtrage des collections d'objet et la création/modification d'éléments de modèle. Ces nouveaux opérateurs de mutation sont utilisés pour produire des mutants à partir d'une transformation de modèles. L'application de ces opérateurs est manuelle, mais peut être parfaitement effectuée automatiquement si un langage de transformation de modèle est choisi. L'exécution d'une transformation testée T et de ses mutants T_1, T_2,...,T_k diffère légèrement de l'exécution d'un programme, mais le principe reste sensiblement le même que dans le processus d'analyse de mutation classique. La comparaison des modèles de sortie produits par la transformation de modèle testée T et les mutants T_i peut être effectuées en utilisant des outils adéquats tel que EMFCompare [40]. Si une différence est observée, le mutant est considéré comme *tué*, sinon il est considéré comme *vivant*. Une fois que les mutants sont exécutés sur tous les modèles de test, le score de mutation est calculé. Comme pour le processus d'analyse de mutation classique, si le score de mutation est trop faible, de nouvelles données de test sont produites pour tuer les mutant (non équivalent) *vivant*.

Finalement, la figure 100 présente le processus d'analyse de mutation pour les transformations de modèles. On retrouve les phases d'analyse de mutation décrites sur la figure 99. L'analyse des résultats du pro-

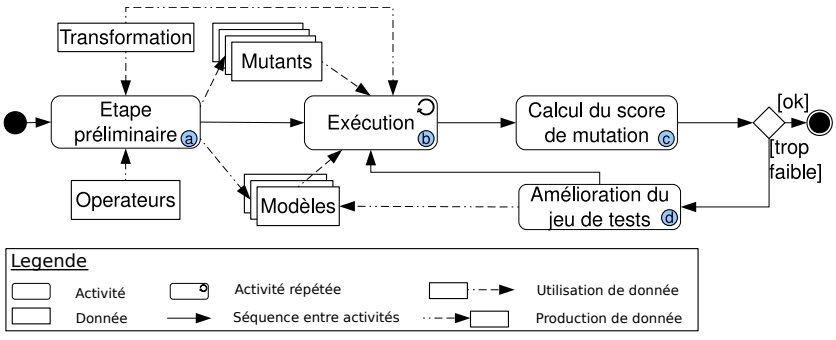


FIGURE 100: Processus d'analyse de mutation adapté aux modèles

cessus d'analyse de mutation et l'amélioration du jeu de tests restent cependant manuel.

Dans la suite de ce chapitre, nous nous concentrons uniquement sur l'amélioration du jeu de tests (activité (d)) dans le processus d'analyse de mutation dédiée aux transformations de modèles. Notre proposition repose sur l'hypothèse suivante : construire de nouveaux modèles de test à partir de rien peut être un travail complexe, alors que bénéficier de l'analyse des modèles déjà existants pourrait aider à la construction de nouveaux. Ainsi, nous avons développé une approche qui permet d'aider à la création de nouveaux modèles de tests à partir de l'adaptation de modèles existants pertinents.

11.2 LA TRAÇABILITÉ, UN MOYEN DE COLLECTER DE L'INFORMATION

En considérant que la création de nouveaux modèles de test est plus simple lorsqu'elle est effectuée à partir d'un modèle déjà existant, les problèmes qu'ils se posent pour l'amélioration du jeu de tests peuvent se résumer en trois questions :

- Parmi tous les couples existants (modèle de test, mutant), lesquels sont les plus intéressants à étudier ?
- À quoi devrait ressembler le modèle de sortie si un mutant était tué ? C'est-à-dire, quelle doit être la différence que nous voulons faire apparaître dans le modèle de sortie ?
- Comment modifier le modèle de test pour produire le modèle de sortie attendu et, ainsi, tuer le mutant ?

Pour aider le testeur à répondre à ces trois questions, nous avons développé une méthode basée sur notre mécanisme de trace locale [4].

4. *C.f*, chapitre 3, page 57.

11.2.1 *Traces de transformation et matrices de mutations*

Les résultats de l'exécution des mutants avec les données de test sont conservés dans une matrice, portant le nom de : matrice de mutation. Cette matrice de mutation permet d'indiquer si le mutant a été *tué* ou non par une donnée de test en particulier. L'analyse des résultats contenus dans la matrice de mutation et des informations de trace sont combinées pour automatiser une partie du processus d'amélioration du jeu de tests. La figure 101 présente un aperçu général de comment la trace est produite pendant l'étape d'exécution des mutants et de la transformation testée avec les différentes données de test. Cette figure détaille l'étape d'exécution présentée en figure 100(b).

Cette étape d'exécution requiert plusieurs paramètres tel que : la transformation testée T, l'ensemble des mutants produits T_i et l'ensemble des modèles de test. Pour chaque exécution d'une transformation [5], une trace locale est générée (activité 1). Le résultat intermédiaire, à savoir, *tué* ou *vivant* pour un couple (modèle de test, mutant) est conservé dans une cellule d'une matrice de mutation. Cette matrice sert à la définition du statut final des mutants. Ainsi, si aucune cellule relative à un mutant n'est marquée comme *tué*, le mutant est considéré comme étant *vivant*. À chacun de ces couples, on associe, également, la trace locale produite lors de l'exécution du mutant.

5. Initiale ou mutante

La matrice de mutation devient donc un pivot entre les mutants, les modèles de test et leur trace locale. Chaque cellule d'un modèle de la matrice de mutation correspond à une abstraction de l'exécution

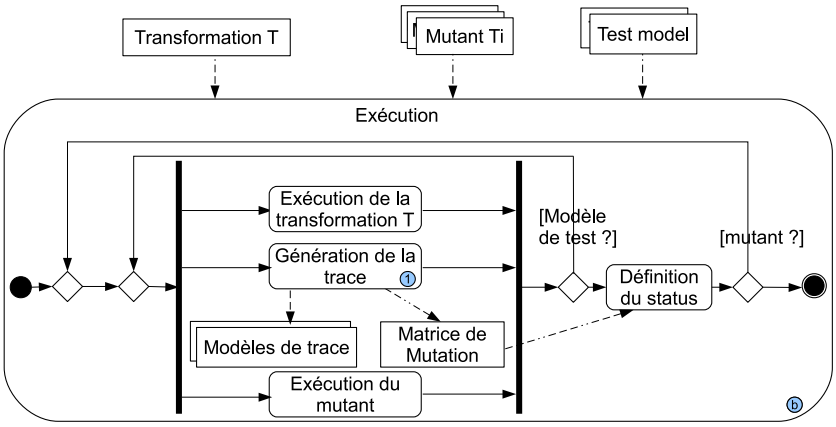


FIGURE 101: Génération des modèles de trace locale

d'un mutant T_i avec une donnée de test D_k. En représentant la matrice de mutation sous forme de modèle, la navigation entre les différents mondes est facilitée. En pratique, le modèle de matrice de mutation est produit à partir des résultats de la comparaison entre les modèles produits par la transformation initiale et ceux produits par un mutant T_i.

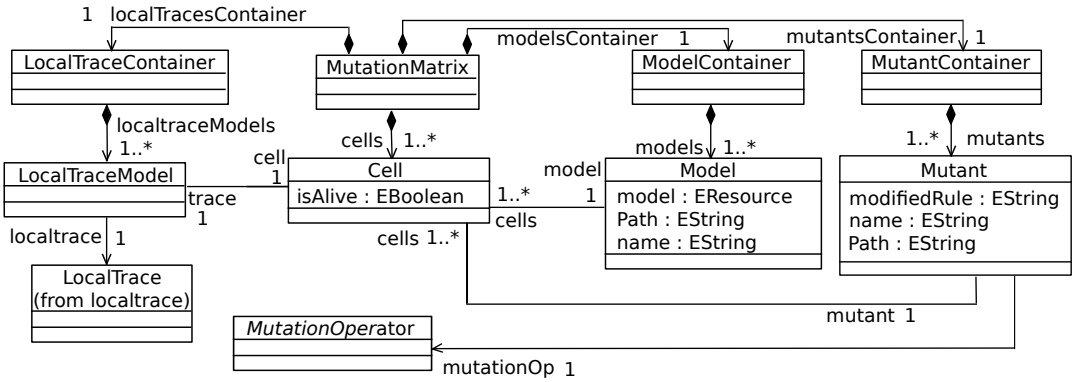


FIGURE 102: Extrait du méta-modèle de matrice de mutation

Le méta-modèle de matrice de mutation, présenté en figure 102, est organisé autour de trois concepts. Le concept *Mutant* fait référence aux mutants créés à partir de la transformation originale. Un mutant possède une de ses règles (*modifiedRule*) en fonction d'un opérateur de mutation qui est représenté par le concept *MutationOperator*[6]. Le concept *Model* représente les modèles de test d'entrée. Les cellules *Cell*, quant à elles, correspondent chacune à une abstraction d'un couple (modèle de test D_k, mutant T_i). La valeur de l'attribut *isAlive* (*true* ou *false*) précise l'état du *Mutant* T_i (respectivement, *vivant* ou *tué*) lorsqu'il est exécuté avec un modèle de test spécifique D_k. La trace locale générée à l'exécution d'un mutant T_i avec un modèle D_k est, également, associée à une cellule *Cell*.

6. Ce concept est abstrait, les concepts concrets qui en héritent seront détaillés en section 11.3.

11.2.2 *Amélioration du jeu de modèles de test*

La figure 103 présente l'activité d'amélioration du jeu de modèles de test en détail (figure 100(d)). La matrice de mutation est directement utilisée pour sélectionner un mutant *vivant* (activité (1)). Une fois qu'un

mutant *vivant* est choisi, l'activité d'identification d'un modèle de test pertinent est lancée en prenant en compte les modèles de trace générés pendant l'activité d'exécution (figure 101(1)). Le modèle sélectionné est ensuite modifié pour produire un nouveau modèle de test tuant le mutant (activité (3) qui sera détaillée en section 11.2.3).

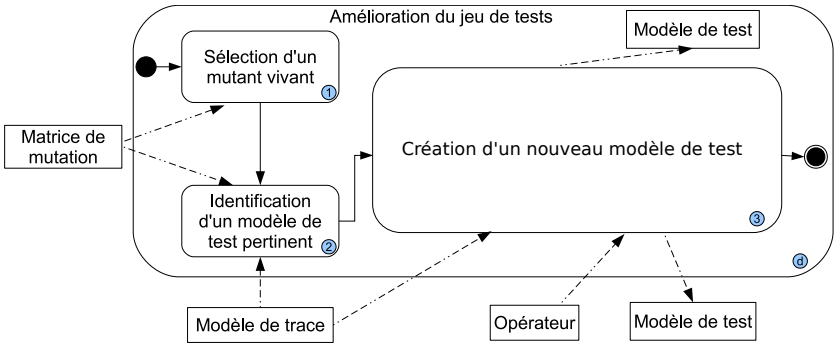


FIGURE 103: Processus d'amélioration des modèles de test

Sélection d'un mutant vivant

La sélection d'un couple pertinent débute par le choix d'un mutant *vivant* sur lequel travailler[7]. Les mutants *vivants* peuvent être facilement et automatiquement identifiés en explorant les cellules de la matrice de mutation. Les cellules relatives à un mutant donné sont parcourues. Si, pour chacune d'elles, l'attribut *isAlive* possède la valeur *true*, alors le mutant est considéré comme étant *vivant*. Le choix du mutant *vivant* à étudier parmi tous les mutants *vivant* est laissé au testeur.

7. Rappel : un mutant est *vivant* si aucun modèle de test ne l'a *tué*.

Nous avons précédemment énoncé trois questions que le testeur est en mesure de se poser lorsqu'il tente d'améliorer le jeu de modèles de test. Nous allons, dans cette sous-section, donner un élément de réponse à la première question, à savoir : « Parmi tous les couples existants (modèle de test, mutant), lesquels sont les plus intéressants à étudier » ? Pour répondre à cette première question, nous nous appuyons sur les modèles de trace générés ainsi que sur le modèle de matrice de mutation.

Identification d'un modèle de test pertinent

Identifier un modèle de test comme bon candidat, parmi tous les modèles de test, pour tuer un mutant *vivant* donné est une tâche beaucoup plus complexe. Notre approche repose sur le principe que les modèles de test qui ont permis la traversée de la règle fautive du mutant sont de bons candidats. En effet, les éventuelles conditions entraînant l'exécution de la règle ont été satisfaites. Notre mécanisme de traçabilité nous permet d'identifier ces modèles et, pour chacun d'eux, de mettre en valeur les éléments consommés et créés par la règle erronée. L'algorithme 4 implémente cette partie du processus d'amélioration du jeu de modèles de test.

L'algorithme prend en entrée un mutant *vivant* de la matrice de mutation. À partir de la matrice, il est possible d'accéder aux modèles de trace correspondant à ce mutant. Les cinq premières lignes de l'algorithme correspondent à l'initialisation des différentes variables. La variable *trace* garde la trace associée à l'exécution du mutant T_i pour

Algorithme 4 Récupération d'information pour un mutant

```
Require: mutant : Mutant
 1: trace ← null
 2: rule ← null
 3: modifiedRule ← mutant.modifiedRule
 4: modelsHandled ← ∅
 5: eltsHandledSrc ← ∅
 6: eltsHandledDest ← ∅
 7: for each mutant.cells do
 8:    trace ← cell.trace
 9:    rule ← trace.findRule(modifiedRule)
10:    if rule ≠ null then
11:       modelsHandled += cell.model
12:       tempEltsSrc ← ∅
13:       tempEltsDest ← ∅
14:       for each rule.links do
15:          tempEltsSrc += link.srcElements
16:          tempEltsDest += link.destElements
17:       end for
18:       eltsHandledSrc += tempEltsSrc
19:       eltsHandledDest += tempEltsDest
20:    end if
21: end for
```

Algorithme 5 Recherche d'une règle dans la trace locale

```
Require: trace : LocalTrace, ruleName : String
1: for each rule ∈ trace.ruleContainer do
2:    if rule.name = ruleName then
3:       return rule
4:    end if
5: end for
6: return null
```

un modèle de test D_k donné. La variable *rule* fait référence à une règle dans le modèle de trace locale[8]. La variable *modifiedRule* est une chaîne de caractère initialisée avec le nom de la règle modifiée du mutant T_i. La variable *modelsHandled* est un ensemble de modèles contenant les modèles de test pour lesquels l'exécution d'un mutant a entraîné la traversée de la règle modifiée. Les variables *eltsHandledSrc* et *eltsHandledDest* contiennent les listes des éléments consommés (respectivement produits) par la règle fautive.

8. Pour mémoire, la règle dans la trace locale est donné par le concept *RuleRef*.

L'algorithme inspecte chaque cellule relative au mutant étudié T_i. Les traces correspondant à l'exécution de ce mutant sur un modèle de test D_k sont récupérées (ligne 8). La trace est ensuite naviguée pour vérifier si la règle modifiée a été traversée pendant l'exécution du mutant. Cette recherche est effectuée par la méthode *findRule*[9]. Cette méthode explore le container de règle *RulesContainer* d'une *LocalTraceModel* (ligne 1 à 5, listing 5) jusqu'à ce qu'une instance de *RuleRef* possédant le même nom que celui de la règle modifiée soit trouvée, c'est-à-dire une règle ayant son attribut *name* avec la même valeur que l'attribut *modifiedRule* du concept *Mutant*. Cette méthode retourne une instance de *RuleRef* ou *null* si la règle n'apparaît pas dans la trace. Le résultat est sauvegardé dans la variable *rule* (ligne 9 de l'algorithme 4). Si le contenu de la variable

9. Détaillé au listing 5

rule est *null*, l'analyse s'arrête ici pour cette cellule et continue avec la suivante. Si celle-ci n'est pas vide, le modèle D_k est sauvegardé dans la variable *modelHandled* (ligne 10). Pour chaque lien de trace *Link* associé à cette règle, la liste des éléments d'entrée (*srcElements*) est gardé dans la variable *eltsHandledSrc* grâce à la variable temporaire *tempEltsSrc*. La gestion du modèle de sortie et de ses éléments est effectuée de la même façon (ligne 12 à 17).

Pour un mutant donné T_i, cet algorithme fournit :

1. des modèles de test candidats à la modification (*modelsHandled*),
2. leurs éléments consommés par la règle fautive du mutant (*eltsHandledSrc*),
3. les éléments produits par cette règle (*eltHandledDest*).

Les variables *eltsHandledSrc* (et *eltsHandledDest*) sont composés de différents ensembles où chacun d'eux représente les éléments consommés (respectivement produits) par la règle fautive pour un modèle de test.

Si le contenu de la variable *modelHandled* est vide, cela veut dire que la règle modifée du mutant n'a jamais été appelée quel que soit la donnée de test utilisée. Un nouveau modèle de test doit donc être créé, potentiellement à partir de rien, contenant les types d'éléments requis par cette règle pour pouvoir entraîner son exécution. À l'inverse, si la variable *modelHandled* n'est pas vide, cela signifie que la règle fautive a été appelée au moins une fois. Cependant, puisque le mutant est vivant, cette règle n'a jamais produit de résultat différent de celui produit par la transformation originale T. Un nouveau modèle de test est donc créé en adaptant les modèles de test candidats identifiés.

11.2.3 *Création d'un nouveau modèle de test*

Pendant cette étape, un nouveau modèle de test est produit en utilisant les informations récupérées par l'algorithme précédemment présenté. Cette étape d'amélioration du jeu de tests est composée de trois sous étapes comme le montre la figure 104, proposant une vue plus détaillée de l'activité d'amélioration du jeu de tests, présentée en figure 103(1). Les plus importantes sont les 1(a) et 1(c) qui modifient un modèle précédemment identifié comme pertinent. Comme ces sous-étapes modifient délibérément un modèle de test choisi parmi ceux existant, la sous étape 1(b) copie le modèle de test considéré pour le conserver inchangé dans le nouveau jeu de modèles de test.

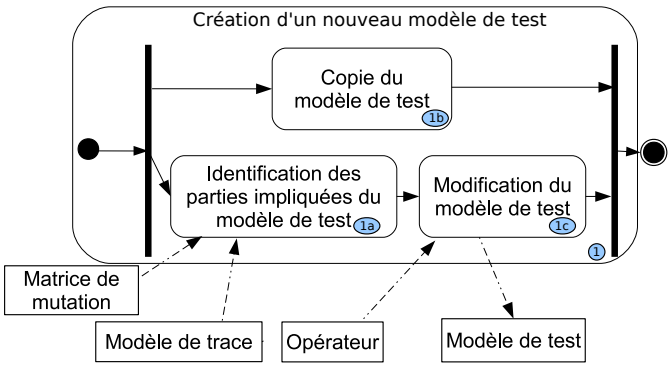


FIGURE 104: Processus de création d'un nouveau modèle de test

La modification du modèle de test utilise à la fois les éléments précédemment identifiés et l'opérateur de mutation appliqué. En effet, en se reposant sur une analyse statique, le testeur doit comprendre pourquoi le mutant reste *vivant* alors que la règle mutée a été appelée sur les éléments identifiés. Il doit ensuite modifier le modèle en conséquence.

11.3 VERS UNE AUTOMATISATION DE L'AMÉLIORATION DU JEU DE TESTS

La création d'un nouveau modèle de test est, actuellement, effectuée manuellement. Afin de tendre vers une automatisation du processus, il faut pouvoir identifier pourquoi un mutant est resté *vivant*. Nous avons observé que pour chaque opérateur de mutation, il y a, finalement, quelques configurations du modèles de test qui n'entraînent pas obligatoirement la *mort* d'un mutant. Nous avons listé, pour chaque opérateur ces situations et identifié des modifications associées pouvant entraîner la mort d'un mutant. Nous associons donc à l'algorithme de sélection de modèles pertinents un moyen de détecter automatiquement ces situations dans les modèles de test et d'effectuer automatiquement, dans la mesure du possible, les modifications sur un modèle en particulier.

11.3.1 *Exemple*

À titre d'exemple, nous prenons l'opérateur *MSNA* [90]. Cet opérateur ajoute une navigation inutile à une séquence de navigation déjà existante tout en respectant les contraintes du méta-modèle impliqué dans la transformation. Ainsi, par exemple, une transformation originale navigant la séquence `self.a.b` est mutée en `self.a.b.c`. Nous avons identifié trois cas pour lesquels le mutant reste *vivant* :

1. la séquence de navigation originale et celle mutée pointent finalement vers la même instance,
2. la séquence de navigation originale et celle mutée pointent finalement vers *null*,
3. la valeur de l'attribut de l'élément pointé par la séquence de navigation originale et celle mutée sont les mêmes.

Évidemment, la façon de modifier les modèles de test diffère pour chacun des trois cas.

Premier cas

Le premier cas intervient lorsque la navigation ajoutée est du même type que la dernière navigation de la séquence originale. Une telle situation est possible si la navigation ajoutée correspond à une référence réflexive [10] dans le méta-modèle. La navigation mutée est donc `self.a.b.b` pour une séquence de navigation originale `self.a.b`. Le mutant peut être tué si la séquence de navigation originale et celle mutée pointent vers deux instances différentes de la même méta-classe. Une nouvelle instance de cette méta-classe contenant des valeurs d'attribut différentes est donc ajoutée au modèle de test avec les références du modèle mises à jour en conséquence.

10. C'est-à-dire une référence d'une classe vers elle-même.

Second cas

Le second cas intervient si une navigation intermédiaire ne peut être effectuée. Dans notre exemple, si la référence a (ou la référence b) pointe vers *null*, ni la séquence de navigation originale self.a.b , ni la séquence de navigation mutée self.a.b.c ne peut être entièrement naviguée. L'élément récupéré par les deux séquences est *null*. Le modèle de test sélectionné peut alors être modifié en mettant à jour les références vide, c'est-à-dire en faisant pointer les références *null* vers un élément du bon type.

Troisième cas

Le troisième cas intervient lorsque l'élément récupéré par la séquence originale self.a.b et la séquence récupérée par la séquence mutée self.a.b.c possèdent un attribut avec le même nom [11]. Le mutant reste vivant si, dans le modèle de test, les deux attributs des instances récupérées [12] possèdent la même valeur. Pour résoudre ce problème, le testeur peut modifier un des attributs en changeant sa valeur.

11. Par exemple, un attribut *label*.

12. Potentiellement de deux méta-classes différentes.

En étendant ce travail sur tous les opérateurs identifiés en [90], la création de nouveaux modèles de test peut être encore plus facilement appréhendée par le testeur. Cependant, pour effectuer cette automatisation, les algorithmes ont besoin de travailler directement avec le langage de transformation utilisé. Pour pouvoir généraliser ce travail à n'importe quel langage de transformation, il semble inévitable de devoir utiliser une définition plus générique pour définir les opérateurs de mutation. Comme nous l'avons présenté au chapitre 9, dans [114], les auteurs ont proposé MuDeL, un langage permettant de décrire les opérateurs de mutation indépendamment de tout langage. Ainsi, un opérateur générique donné peut être réutilisé avec plusieurs langages. Cependant, les opérateurs MuDeL sont dédiés aux langages de programmation classique et ne sont pas utilisables pour les transformations de modèles. Une représentation générique des opérateurs de mutation définis en [90] est donc nécessaire pour pouvoir faire bénéficier notre approche à n'importe quel langage de transformation.

11.3.2 *Modéliser les opérateurs de mutation*

Dans [90], les auteurs identifient les opérateurs de mutation en s'appuyant sur les 3 phases constituant les transformations de modèles : *navigation*, *filtrage* et *création ou modification* des éléments des modèles d'entrée et de sortie. Les phases de navigation permettent la sélection d'éléments en fonction de relations définies sur les méta-modèles d'entrée. Celles de filtrage réduisent un ensemble d'éléments manipulés par la transformation par l'application d'une condition. Les éléments de modèles de sortie sont créés à partir d'éléments des modèles d'entrée. Dans le cas d'une transformation *sur place*, les éléments des modèles d'entrée pourraient être modifiés. Les phases de navigation, filtrage, création et modification sont séquentiellement dépendantes tant que la navigation retourne des éléments. En effet, une fois les collections naviguées, elles sont souvent filtrées avant de créer ou modifier des éléments de modèles de sortie. Ces phases constituent un cycle basique qui est répété pour composer une transformation de modèle complète.

La décomposition d'une transformation de modèles en de telles phases élémentaires fournit une vue abstraite utile pour l'injection de fautes. Ainsi, dans [90], dix opérateurs de mutation, appliqués sur ces phases, ont été définis. Ils correspondent à l'injection d'une opération de navigation, de filtrage, de création ou de modification erronée. Ces opérateurs ont été définis sans prêter attention aux détails d'un quelconque langage de transformation. En effet, même si QVT a été défini comme le standard pour les transformations de modèles en 2004, actuellement, encore peu de transformations l'utilisent. Beaucoup d'autres langages de transformation existent. Si certains de ces langages de transformation utilisent une représentation intermédiaire de la transformation sous forme de modèle, les méta-modèles utilisés sont différents pour chaque langage. Afin d'être le plus générique possible, nous avons donc décidé de définir les opérateurs de mutation non pas comme des modifications opérées sur une transformation de modèle, mais comme une opération de modification de navigation, de filtrage ou de création sur les méta-modèles d'entrée ou de sortie de la transformation. De cette façon, les opérateurs de mutation sont représentés comme des méta-modèles faisant référence aux méta-classes et relations des méta-modèles d'entrée et de sortie manipulés par une transformation de modèles. Ainsi, les opérateurs de mutation appliqués aux transformations de modèles sont toujours définis indépendemment de tout langage de transformation, mais ils sont facilement manipulables.

Dans notre approche pour l'amélioration automatique du jeu de tests, l'analyse de l'opérateur de mutation nous permet de prendre en compte ses spécificités pour détecter automatiquement le cas problématique dans un modèle. Une fois le cas problématique identifié, l'analyse de l'opérateur de mutation permet aussi de proposer un modèle solution ou une modification à effectuer pour obtenir un nouveau modèle de test.

Les sous-sections suivantes présentent trois méta-modèles d'opérateurs de mutation ; pour chaque opérateur de mutation, nous décrivons :

- sa définition proposée en [90],
- son méta-modèle,
- un exemple d'application de l'opérateur sur un méta-modèle.

La totalité des opérateurs, c'est-à-dire, ceux présentés dans ces sous-sections et ceux manquants sont détaillés en annexe A. Les exemples d'application de ces opérateurs sont présentés sur le méta-modèle montré en figure 105. Ce méta-modèle sera considéré comme étant méta-modèle d'entrée ou de sortie de la transformation selon l'opérateur de mutation. Il contient six méta-classes liées avec différentes références et relations d'héritage.

Mutation de navigations : l'opérateur Remplacement d'une relation vers une même classe (RRMC)

Les opérateurs de mutation relatifs à la phase de *navigation* peuvent être représentés en fonction du méta-modèle d'entrée de la transformation. En effet, cette phase travaille seulement avec le méta-modèle d'entrée pour sélectionner des concepts sur lesquels les règles de transformation sont appelées. La figure 106 présente l'opérateur de mutation *RRMC*. La partie haute du méta-modèle vient du noyau EMOF. Il est à noter que pour chaque méta-modèle d'opérateur, plusieurs concepts abstraits, héritant les uns des autres, apparaissent en plus du concept représentant l'opérateur. Cette hiérarchie est utile pour pouvoir, factoriser

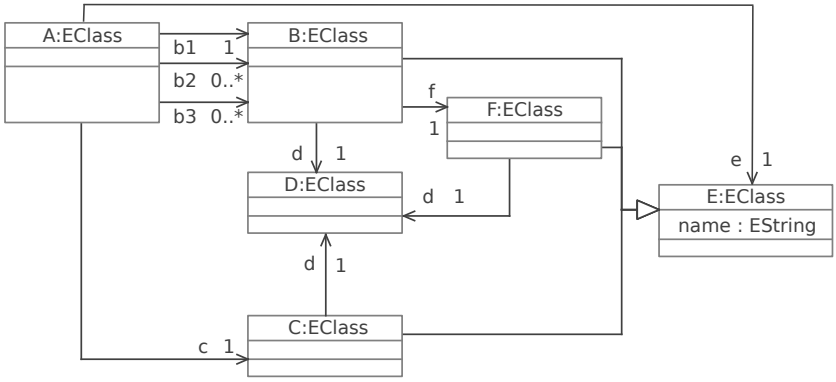


FIGURE 105: Méta-modèle support pour les examples de modélisation des opérateurs de mutation

et réutiliser certaines relations. Par exemple, la différence entre les opérateurs *RRAC* [13] et *RRMC* est très mince, ces deux opérateurs partagent donc des relations communes qui appartiennent à leurs super classes. L'opérateur de mutation est décrit par les paragraphes ci-dessous.

13. *C.f*, Annexe A

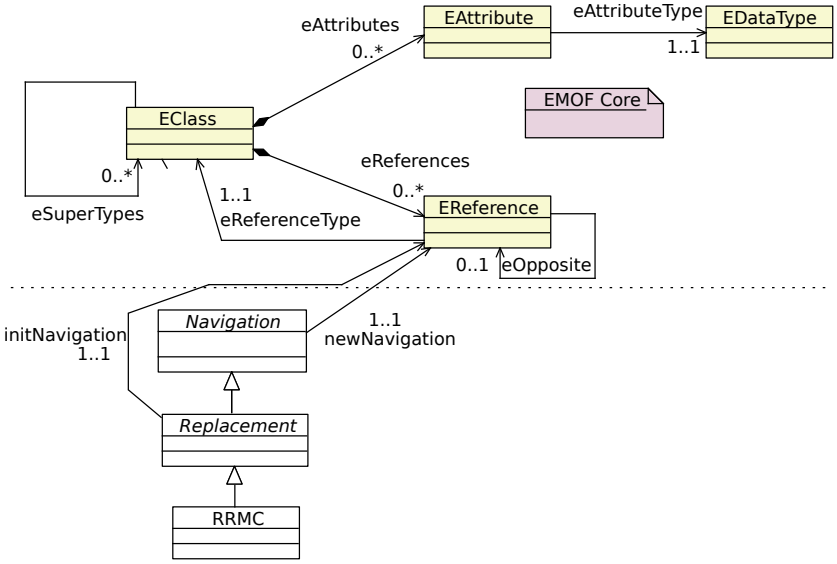


FIGURE 106: Méta-modèle de l'opérateur *RRMC*

DÉFINITION « L'opérateur *RRMC* remplace la navigation d'une association vers une classe, par la navigation d'une association vers la même classe. »

DESCRIPTION DU MÉTA-MODÈLE La table 12 regroupe les méta-classes et les références spécifiques de l'opérateur *RRMC* représenté en figure 106.

EXEMPLE L'opérateur *RRMC* modélisé en figure 107 précise que la transformation originale navigue la référence *b1* alors que la référence mutée navigue la référence *b2*.

Mutation de filtrages : L'opérateur Modification d'un filtrage d'une collection avec ajout (MFCA)

Les opérations de filtrage manipulent des collections pour ne sélectionner seulement que quelques éléments utiles pour la transformation.

MÉTA-CLASSE/RELATION	DESCRIPTION
RRMC	L'opérateur de mutation
initNavigation	Référence initialement naviguée par la transformation
newNavigation	Référence naviguée après la mutation

TABLE 12: Description du méta-modèle de l'opérateur *RRMC*

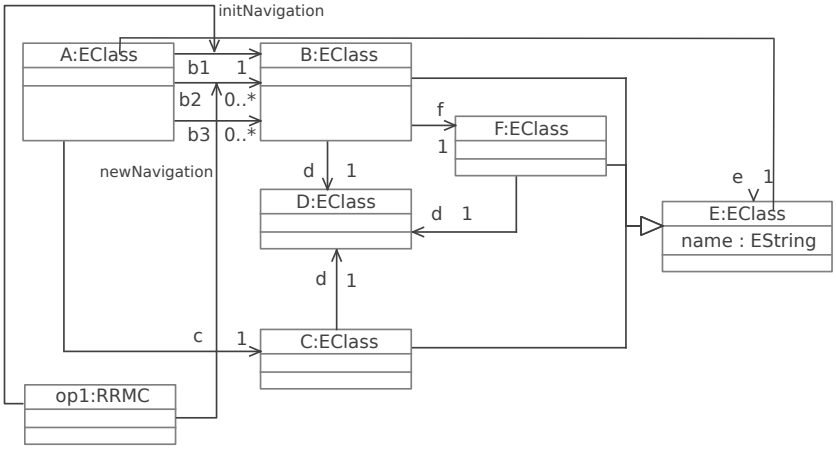


FIGURE 107: Exemple d'application de l'opérateur *RRMC*

De manière générale, un filtrage peut être considéré comme une garde sur une collection dépendant de critères spécifiques. Ces gardes peuvent être altérées de plusieurs façons, chacune relative à une opération de mutation. La figure 108 présente le méta-modèle de l'opérateur de mutation sur filtrage *MFCA*.

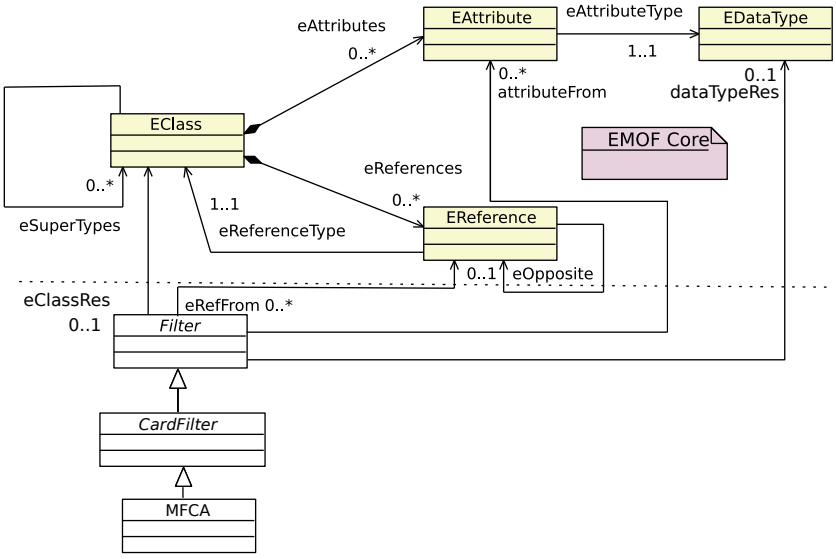


FIGURE 108: Méta-modèle de l'opérateur *MFCA*

DÉFINITION « L'opérateur *MFCA* utilise une collection et procède un filtrage inutile sur celle-ci. Cet opérateur pourrait retourner un

nombre infini de mutants et nous devons le contraindre. Nous avons utilisé une collection et nous retournons un seul élément sélectionné arbitrairement. »

DESCRIPTION DU MÉTA-MODÈLE La table 13 regroupe les méta-classes et les références spécifiques de l'opérateur *MFCA* représenté en figure 108.

MÉTA-MODÈLE / RELATION	DESCRIPTION
MFCA	L'opérateur de mutation
eRefFrom	Collection de références sur laquelle l'opérateur est appliqué
attributeFrom	Collection d'attribut sur laquelle l'attribut est appliqué
eClassRes	Type de la collection de références manipulées
dataTypeRes	Type de la collection d'attributs manipulés

TABLE 13: Description du méta-modèle de l'opérateur *MFCA*

Le filtrage est appliqué soit sur une référence (*eRefFrom*), soit sur une collection d'attributs (*attributeFrom*). Dans les deux cas, le type des éléments contenus dans la collection est spécifié par soit *eClassesRes*, soit *dataTypesRes*.

EXEMPLE L'opérateur *MFCA* modélisé en figure 109 précise que la collection originalement filtrée est la collection de référence *b3* et que le filtrage muté implique la sélection d'un seul élément retournant ainsi une collection d'un seul élément de type *B*.

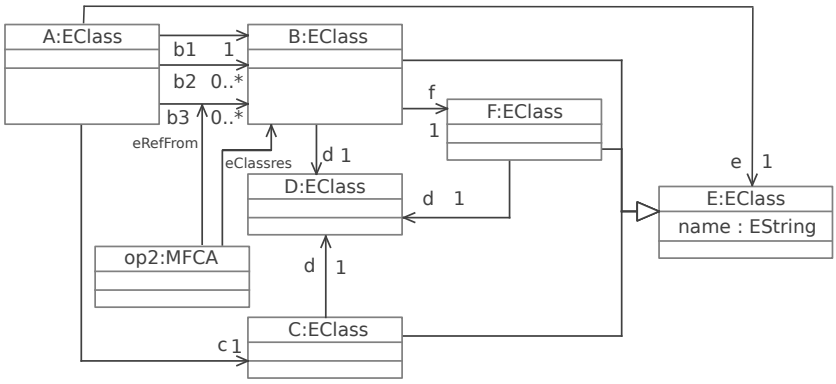


FIGURE 109: Exemple d'application de l'opérateur *MFCA*

Mutation de création d'éléments : l'opérateur Modification d'une mise en relation de classe avec ajout(MRCA)

Les opérateurs de mutation de *création* sont relatifs à la dernière phase du processus de transformation[14]. Ils peuvent être exprimés sur le méta-modèle de sortie puisque cette phase de création travaille uniquement avec les éléments du modèle de sortie. Le méta-modèle de l'opérateur *MRCA* est présenté en figure 110.

14. C'est-à-dire, relatif à la création ou à la modification d'éléments appartenant aux méta-modèles de sortie.

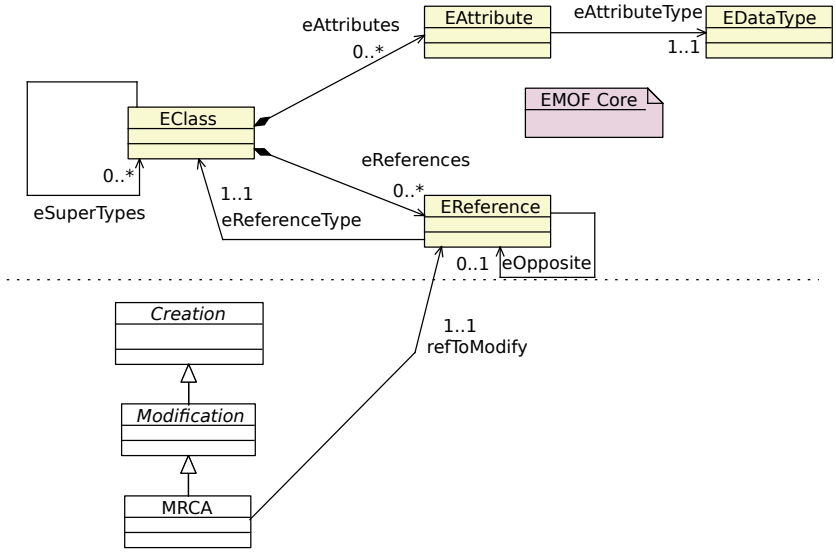


FIGURE 110: Méta-modèle de l'opérateur *MRCA*

DÉFINITION « L'opérateur *MRCA* ajoute la création inutile d'une relation entre deux instances de classe du modèle de sortie lorsque le méta-modèle le permet. »

DESCRIPTION DU MÉTA-MODÈLE La table 14 regroupe les méta-classes et les références spécifiques de l'opérateur *MRCA* représenté en figure 110.

MÉTA-CLASSE/RELATION	DESCRIPTION
MRCA	L'opérateur de mutation
refToModify	Relation ajoutée par l'opérateur

TABLE 14: Description du méta-modèle de l'opérateur *MFCA*

EXEMPLE L'opérateur *MRCA* modélisé en figure 111 précise qu'une relation *c* d'une instance de A vers une instance de C est inutilement ajoutée au modèle de sortie. Avec cet opérateur, si la relation existe déjà [15], la relation est tout simplement écrasée.

15. La relation c est de cardinalité 1 et peut déjà avoir été initialisée.

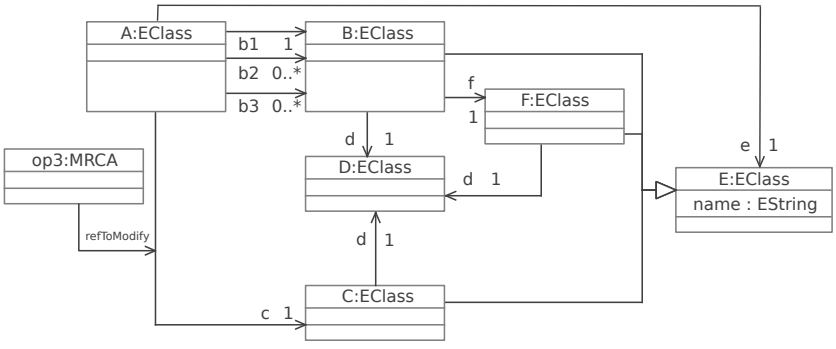


FIGURE 111: Exemple d'application de l'opérateur *MRCA*

Dans cette sous-section, nous avons présenté trois opérateurs de mutations relatifs aux trois grandes catégories énoncées dans [90]. La

modélisation de ces opérateurs de mutation est intéressante, car elle peut amener une analyse plus automatique du mutant. Dans la section suivante, nous allons voir les cas qui laissent les mutants présentés dans cette sous-section comme vivants et comment produire, plus ou moins automatiquement, un nouveau modèle de test permettant de tuer le mutant étudié.

11.3.3 *Liaison avec la matrice de mutation*

Dans la section 11.2, nous avons proposé un méta-modèle pour la matrice de mutation. La matrice de mutation est utilisée comme un pivot entre les mutants, les modèles de test et les traces générées lors de l'exécution des mutants sur les modèles de test. Lors de sa description, nous avions laissé volontairement de côté la méta-classe *MutationOperator* qui représente l'opérateur de mutation qui a permis de créer le mutant. La figure 112 présente un extrait du méta-modèle de matrice de mutation [16].

16. Déjà montré à la figure 102.

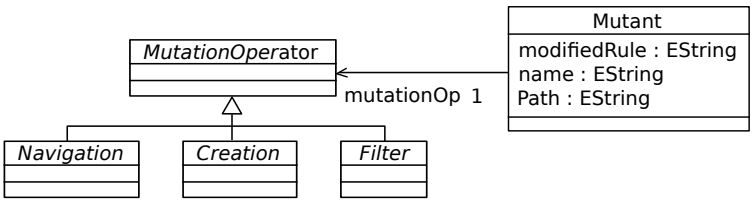


FIGURE 112: Extrait du méta-modèle de matrice de mutation

Sur la figure, seul les concepts se rapportant à la modélisation des opérateurs de mutation sont reportés. La méta-classe *Mutant* est associée à une méta-classe abstraite *MutationOperator* dont les méta-classes abstraites *Creation*, *Navigation* et *Filter* héritent. Ces méta-classes abstraites sont celles présentées en figure 106, 108 et 110. Ainsi, à partir d'un mutant dans la matrice de mutation, il est possible d'accéder à une représentation de l'opérateur qui lui est appliqué.

11.3.4 *Cas problématiques et mutants vivant*

Comme nous l'avons expliqué en sous-section 11.3.1, la traversée de la règle fautive n'implique pas forcément la mort du mutant. Dans certains cas, la règle modifiée dans le mutant peut s'exécuter de la même façon que sa version originale dans la transformation à tester, et ce, quelque soit les modèles d'entrée. Il faut donc analyser les modèles de test utilisés, en fonction de l'opérateur de mutation appliqué sur le mutant. L'algorithme de sélection de modèles pertinents que nous avons précédemment présenté [17] nous permet d'analyser le jeu de modèles de test et d'identifier deux ensembles : les modèles de test pertinents ainsi que leurs éléments pertinents. Ces deux sous-ensembles peuvent servir de base au testeur pour la production manuelle d'un nouveau modèle de test. Nous avons représenté sur la figure 113 le processus de création d'une nouvelle donnée de test présentée à la figure 104.

17. *C.f.* algorithme 4, page 196.

Pour pourvoir créer une nouvelle donnée de test automatiquement (activité (1c)), nous proposons dans cette section une analyse de l'opérateur de mutation et une modification de l'un des modèles identifié par notre algorithme. Cette analyse utilise à la fois les opérateurs de mutation modélisés ainsi que les retours de notre l'algorithme d'identi-

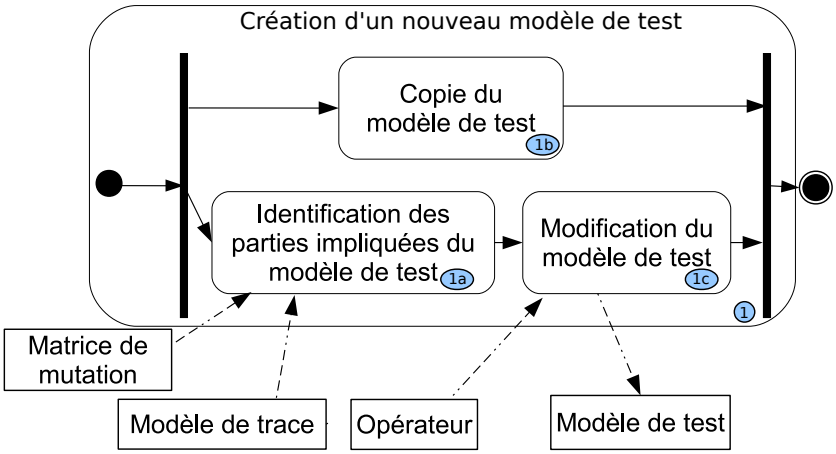


FIGURE 113: Processus d'amélioration des modèles de test

fication des données de test pertinentes. Cette analyse est lancée lorsque les retours proposés par notre algorithme est différent d'un ensemble nul, c'est-à-dire lorsqu'une donnée de test au minimum a été jugée pertinente.

Dans la sous-section 11.3.1, nous avions détaillé les cas problématiques qu'il est possible de trouver pour l'opération de mutation *MSNA* [90]. Nous avions identifié trois cas qui pouvaient laisser identique le résultat de la transformation originale et mutée. Dans cette sous-section, nous allons montrer les différents cas laissant le mutant *vivant* que nous avons identifiés pour les trois opérateurs de mutation précédemment présentés, à savoir : *RRMC*, *MFCA* et *MRCA*.

Les cas problématiques de l'opérateur RRMC

Les cas posant problèmes sont les mêmes que ceux présentés pour l'opérateur *MSNA*. On dénote alors 3 cas :

1. l'instance récupérée par la navigation originale et celle mutée est la même,
2. la navigation originale et celle mutée pointent vers *null*,
3. la valeur de la propriété récupérée par la navigation originale et celle mutée sont les mêmes.

Les solutions sont les mêmes que celles mises en oeuvre pour l'opérateur *MSNA*.

DÉTECTION AUTOMATIQUES DES CAS : la modélisation de l'opérateur de mutation est utilisé avec les retours de l'algorithme pour détecter les différents cas. Pour chacun des éléments récupérés par l'algorithme, les relations pointées par *initNavigation* et *newNavigation* sont naviguées. Les instances récupérées sont ensuite comparées pour détecter si elles sont identiques (cas 1) ou *null* (cas 2). Si les instances sont différentes, les attributs contenus par les instances sont comparés pour ainsi détecter les valeurs communes (cas 3). Une fois que le cas est identifié, le modèle peut être modifié en conséquence.

SOLUTIONS POUR LE PREMIER CAS : l'idée est d'ajouter une nouvelle instance. Une première solution est d'ajouter une nouvelle instance dans le nouveau modèle de test complètement différente

de celle récupérée par navigation de la référence originale et mutée. Cette solution, certes simple, peut ne pas tuer un mutant *vivant*. Il peut donc être plus pertinent d'ajouter une instance en faisant varier les éléments des références manipulées par le mutant :

- Si au moins une des deux références (originale ou mutée) a sa borne supérieure strictement supérieure à 1 [18], il suffit d'ajouter une nouvelle instance à la collection. En effet, l'ajout d'une nouvelle instance dans une collection manipulée par le mutant ou la transformation originale peut largement influer sur le résultat produit.
- Si au moins une des deux relations possède sa borne inférieure égale à 0, il est possible de supprimer cette référence vers l'élément pointé. Ainsi, la transformation originale ou le mutant [19] ne pourra pas récupérer d'instance et donnera un résultat différent.

18. Références $1 - n$ et $1 - *$, il s'agit donc de collections.

19. Selon la référence supprimée.

SOLUTION POUR LE SECOND CAS : une fois encore, l'idée est d'ajouter une nouvelle instance, comme pour le cas précédent, dans un nouveau modèle de test et de mettre à jour la référence naviguée par la transformation originale pour qu'elle pointe vers ce nouvel objet.

SOLUTION POUR LE TROISIÈME CAS : cette fois, ce sont les valeurs des propriétés qu'il faut modifier. Ainsi, une fois les instances manipulées par la règle fautive identifiées, il suffit de modifier la valeur de ses attributs pour donner naissance à un nouveau modèle de test.

Les cas de l'opérateur MFCA

Les cas relatifs aux opérateurs de filtrage proviennent de problématiques différentes. En effet, ces opérateurs reposent sur la sélection d'éléments dans une collection en fonction d'une condition. Le résultat de la collection filtrée est donc obtenu lors de l'exécution de la transformation. La modélisation que nous avons proposée pour cet opérateur [20] ne prend pas en compte la description de la condition de filtrage. Il serait, en effet, difficile de modéliser cette condition, puisque son expression dépend des possibilités du langage de transformation utilisé.

20. *C.f*, figure 108

Cependant, la modification de la transformation testée par l'opérateur *MFCA* implique la sélection d'un seul élément dans la collection. Si aucune différence n'est notée, il est raisonnable de supposer que le filtrage original n'a retourné aucun ou un seul élément. En effet, la différence de comportement de la transformation mutée par rapport à celle originale tient du fait que la collection filtrée peut contenir plusieurs éléments. Si aucun élément n'est retourné par le filtrage original, le filtrage muté retournera lui aussi aucun élément et les deux transformations vont continuer leur exécution. Il en est de même si un seul élément est retourné par le filtrage original. Dans les deux cas, le mutant et la transformation originale vont se comporter de la même façon.

Nous sommes conscients que sans modélisation complète de la condition de filtrage, il est difficile de fournir une solution sans faille pour produire un nouveau modèle de test tuant les mutants construits avec cet opérateur. Néanmoins, aux vues de suppositions que nous avons

donnée, il est possible de considérer deux cas spéciaux pouvant laisser le modèle inchangé :

- aucun élément n'existe dans la collection filtrée,
- un seul élément existe dans la collection filtrée.

DÉTECTION AUTOMATIQUE DES CAS : afin de trouver automatiquement le nombre d'éléments de la collection, la modélisation du mutant et les résultats de l'algorithme d'identification de modèles de test pertinent sont utilisés. Les éléments passant par la règle mutée et possédant la collection *eRefFrom* (respectivement *attributeFrom* si la collection est une collection d'attributs) sont cherchés. Une fois que ceux-ci sont trouvés, le nombre d'éléments de la collection *eRefFrom* (respectivement *attributeFrom*) est calculé. Les cas sont déterminés en fonction du nombre d'éléments trouvés dans la collection.

SOLUTION POUR LE PREMIER CAS : si aucun élément n'est présent dans la collection sur laquelle s'applique le filtrage, il serait raisonnable de penser que l'ajout d'un élément pourrait mettre en évidence une différence. Cependant, cela reviendrait à considérer le second cas. Afin de donner une solution à ce cas, il faut créer deux instances différentes satisfaisant la condition du filtrage de la collection et ajouter ces instances à la collection filtrée.

SOLUTION POUR LE SECOND CAS : si un seul élément est présent dans la collection, comme pour le cas précédent, il suffit d'ajouter une instance satisfaisant la condition du filtrage de la collection et ajouter cette instance à la collection filtrée.

Évidemment, la collection peut posséder n éléments et le filtrage n'en sélectionne aucun, ou juste un. Dans ce type de cas, le testeur doit aller regarder en détail les modèles de test d'entrée et vérifier la condition du filtrage pour ajouter de nouvelles instances à la collection filtrée.

Pour les cas non identifiés, les éléments des modèles de test passant par la règle mutée sont obtenus grâce à la trace et sont fournis au testeur pour que celui-ci puisse regarder plus en détail le mutant étudié.

Les cas de l'opérateur MRCA

Les opérateurs de *créations* supposent un ajout, un retrait ou un remplacement, directement dans le modèle de sortie, ce qui augmente grandement les chances de mettre en évidence une différence entre le modèle de sortie du mutant et celui de la transformation originale.

Concernant l'opérateur de mutation *MRCA*, les cas problématiques que nous avons recensés sont relatifs à la cardinalité de la référence manipulée par le mutant ainsi qu'aux instances qu'elle référence. La description de l'opérateur présenté dans [90] précise que si la borne supérieur de la référence mutée est à 1, la succession de deux instructions de mise en relation entraîne l'écrasement de la référence mutée. Pour les références dont la borne supérieur est $*$, l'exécution de l'instruction mutée entraîne l'ajout d'un élément de plus dans la collection. Étant donné que les modifications impliquées par les opérateurs de mutation sont directement faites sur les modèles de sortie et non pas par « effet de bord » [21], les cas problématiques ne viennent pas directement de configurations des modèles de test d'entrée. Il est donc difficile d'expliciter des cas problématiques à l'image de ceux que nous avons présentés pour les opérateurs précédents.

21. Par altération d'une opération sur les modèles d'entrée entraînant une modification dans les modèles de sorties de la transformation.

Cependant, pour les cardinalités 0..1, il est possible d'appliquer une modification ad-hoc sur un des modèles d'entrée identifié par l'algorithme que nous avons présenté en section 11.2. En effet, dans ce cas, l'exécution du mutant entraîne un « écrasement » de la référence de sortie pour ajouter la référence inutile vers une classe dans le modèle de sortie. L'idée ici est d'éviter que la référence ajoutée dans la transformation mutante écrase la référence originale. Pour arriver à ce résultat, il suffit de supprimer dans le modèle de sortie l'élément pointé par la référence originale. De cette façon, l'initialisation de la référence originale s'effectue vers un objet inexistant. Dans le modèle de sortie de la transformation originale, la référence ne pointera vers rien alors que la référence du modèle créé à partir du mutant pointera vers un objet et l'oracle pourra ainsi mettre en évidence une différence entre les deux modèles. Pour empêcher la création de cet élément dans le modèle de sortie originale, il suffit de supprimer un élément du modèle d'entrée menant à sa création grâce à la trace locale. Pour détecter ce cas, les éléments du modèle de sortie créés par la règle mutée sont collectés. La relation *refToModify* [22] est naviguée et l'instance qu'elle pointe est récupérée. Finalement, les éléments du modèle d'entrée ayant créés cette instance du modèle de sortie sont ensuite identifiés et le choix de l'élément du modèle d'entrée à supprimer est laissé au testeur.

22. Récupérés à partir de la modélisation du mutant.

Pour les autres cas, c'est-à-dire les cardinalités 1..1, 0..$*$ et 1..$*$, cette modification est : soit inapplicable (pour la cardinalité 1..1) ou potentiellement inefficace. Le testeur doit aller regarder le mutant et les modèles de tests en détail pour comprendre pourquoi le mutant est resté vivant.

Dans cette sous-section, nous avons donné plusieurs cas laissant indubitablement le mutant vivant. Pour chacun de ces cas, nous avons proposé une modification d'un des modèles d'entrée existant pour fournir un nouveau modèle de test tuant le mutant étudié. Cependant, dans certains cas, comprendre pourquoi aucune différence n'a été révélée par l'oracle requiert une analyse plus fine de la transformation qui peut être allégée grâce aux retours effectués par notre algorithme d'identification de modèles pertinents. Afin de résumer les différentes stratégies appliquées et leurs ordres, la figure 114 montre un diagramme d'activité correspondant aux choix effectués par l'algorithme d'assistance à la création des nouveaux modèles de tests.

Le processus démarre lors de l'*identification des modèles et des éléments pertinents*. Si l'algorithme n'a identifié aucun modèle pertinent, cela signifie que la règle mutée n'a jamais été traversée. Ainsi, la règle mutée doit être regardée pour déterminer les éléments nécessaires pour activer sa traversée et un nouveau *modèle de test* est produit en conséquence. En revanche, si la règle mutée a bien été traversée et que le mutant reste vivant, il faut aller analyser la situation plus en détail pour pouvoir créer un nouveau modèle de test. Le processus continue donc en tentant d'identifier si les modèles de test correspondent à un des cas spéciaux que nous avons présentés. Pour effectuer cette tâche, l'*opérateur* de mutation modélisé est utilisé ainsi que le *modèle de trace*.

À l'issue de cette activité, si aucun cas n'a pu être identifié, le testeur doit analyser en détail le mutant. Afin de l'aider dans sa tâche, la trace produite ainsi que les retours de l'algorithme d'identification des modèles pertinents lui est donné. Une fois qu'il y a analysé le mutant, le testeur peut ensuite créer le nouveau modèle de test.

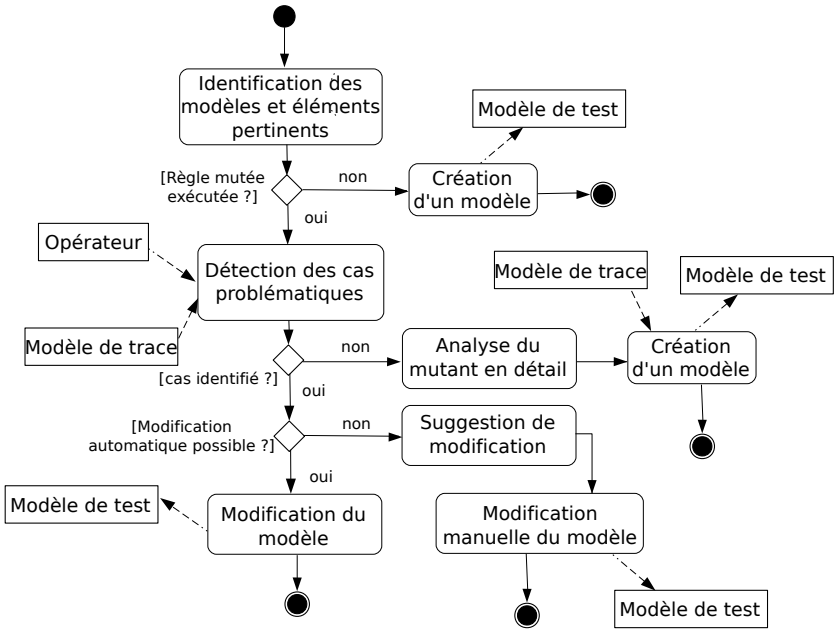


FIGURE 114: Choix effectués lors de la création des nouveaux modèles de test

Cependant, si un cas a bien pu être identifié et si la solution que nous proposons peut être mise en œuvre automatiquement, un nouveau modèle de test est créé. Dans le cas où la modification ne peut pas être effectuée automatiquement, une suggestion de modification est faite au testeur pour que celui-ci modifie le modèle de test en conséquence et qu'il produise la nouvelle donnée de test.

11.4 ÉTUDE DE CAS

Nous illustrons notre approche avec la transformation appelée *class2rdbms* d'un diagramme de classe simple [23] vers un système de base de donnée relationnelle [24]. Dans ce cas d'étude, nous essayons d'améliorer le jeu de modèles de test pour passer du score de mutation initial que nous obtenons à un score de mutation de 90%. Nous montrons aussi l'efficacité de notre approche pour rapidement proposer un nouveau modèle de test possédant la capacité de *tuer* un mutant sans analyse manuelle du mutant ou du jeu de modèles de test.

23. Simple Class Diagram SIMPLECD

24. Relational Database Managment Systems (RDBMS)

Dans un premier temps, nous décrivons rapidement la transformation *class2rdbms*, puis nous expliquons pourquoi cette transformation est représentative et intéressante pour l'étude du test de transformation de modèles. Par la suite, nous illustrons l'approche présentée dans ce chapitre pour améliorer le jeu de modèles de test.

11.4.1 *Description générale*

La transformation *class2rdbms* produit un modèle *RDBMS* à partir d'un modèle *SimpleCD*. Nous adoptons la version de la spécification proposée au workshop MTIP de la conférence MODELS 2005 [17]. Les figures 115 et 116 présentent les méta-modèles SIMPLECD et RDBMS respectivement.

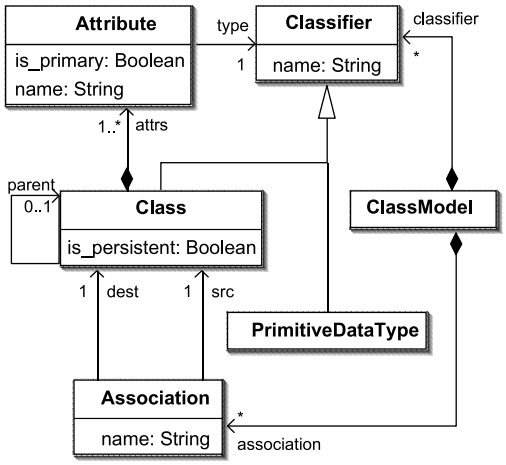


FIGURE 115: Méta-modèle de classe simple [17]

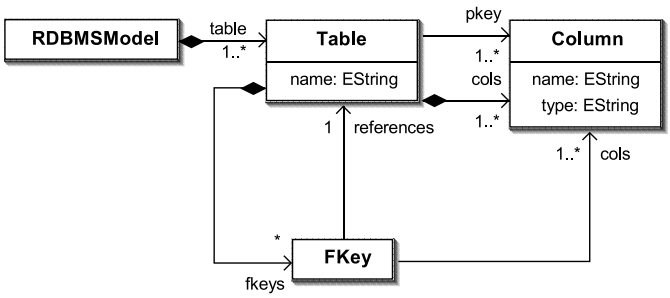


FIGURE 116: Méta-modèle RDBMS [17]

11.4.2 *Un cas d'étude efficace*

La transformation *class2rdbms* est un cas d'étude représentatif pour plusieurs raisons. Premièrement, cette transformation est le benchmark proposé pour le workshop MTIP lors de la conférence MODELS 2005 [17] pour exprimer et valider des possibilités de langages de transformations. Cette transformation a été très largement réutilisée dans le domaine des transformations de modèles. De plus, selon [15], la transformation *class2rdbms* est classifiée comme étant d'un niveau de complexité 6 [25]. Les méta-modèles d'entrée et de sortie couvrent les majeurs concepts de méta-modélisation tel que l'héritage, la composition et les cardinalités finies et infinies. La transformation *class2rdbms* définie un ensemble de règles qui s'appellent entre elles. Elles spécifient, par exemple, des règles de navigation à travers les modèles d'entrée et de sortie. Finalement, la transformation *class2rdbms* possède une taille raisonnable et rend possible l'amélioration de la technique d'analyse de mutation avec une expérimentation significative.

25. Le niveau de complexité maximal est 7.

11.4.3 *Implémentation*

Nous avons choisi de travailler sur une implémentation de la transformation KERMETA, un langage impératif, orienté objet, avec un système de typage orienté modèle. En effet, le travail que nous proposons ici est une extension et une amélioration des travaux initiés en [90]. En utilisant ici le même cas d'étude, nous pouvons comparer et évaluer notre approche par rapport au travail initialement proposé en [90]. L'aperçu

de l'implémentation KERMETA de la transformation est extraite de [91]. La transformation est implémentée en trois étapes :

CRÉATION DES TABLES Les *Tables* du modèle RDBMS sont créées à partir de chaque *Class* marquées comme persistantes dans le modèle d'entrée.

CRÉATION DES COLUMNS Pour chaque *Class* persistante, il faut traiter tous les attributs et les associations sortantes pour créer les colonnes correspondantes. Lors de cette étape, les *FKey* (*Foreign Keys*) sont aussi créées, mais les *Columns* associées ne peuvent pas être créées tant que toutes les *Tables* ne sont pas traitées.

MISE À JOUR DES FKEY Les *Columns* associées à une *FKey* sont créées dans les *Tables* qui contiennent les *Fkeys* et la référence *cols* de la *FKeys* est mise à jour.

La transformation a été implémentée par une classe appelée *Class2RDBMS*. Comme KERMETA n'est pas un langage dédié aux transformations de langages, ce langage manipule des méthodes de classe appelées *operation* plutôt que des règles de transformation. Ainsi, la classe *Class2RDBMS* fournit une méthode *transform* qui prend un modèle d'entrée comme paramètre et retourne le modèle créé correspondant. Le listing 11.1 présente un extrait du code KERMETA de la transformation.

```
1  package Class2RDBMS;
2  require kermeta // The kermerta standard library
3  require "LocalTrace.kmt" // The local trace framework

5  // Input metamodel in Ecore
6  require "../ClassMM.ecore"

8  // Output metamodel in Ecore
9  require "../RDBMSMM.ecore"
10 [...]

12 class Class2RDBMS
13 {
14   /** The trace of the transformation */
15   reference class2table : LocalTrace

17   /** Set of keys of the output model */
18   reference fkeys : Collection<FKey>

20   operation transform(inputModel : ClassModel)
21             : RDBMSModel is
22   do
23     // Initialize the trace
24     class2table := LocalTrace.new
25     class2table.create
26     fkeys := Set<FKey>.new
27     result := RDBMSModel.new

29     // Create tables
30     getAllClasses(inputModel)
31     .select{ c | c.is_persistent }.each{ c |
32       var table : Table init Table.new
33       table.name := c.name
34       class2table.storeTrace(c, table)
35       result.table.add(table)
```

```
36     }

38     // Create columns
39     getAllClasses(inputModel)
40     .select{ c | c.is_persistent }.each{ c |
41       createColumns(class2table
42       .getTargetElem(c), c, "")
43     }

45     // Create foreign keys
46     fkeys.each{ k | k.createFKeyColumns }
47   end
48  [...]
49 }
```

Listing 11.1: Extrait de la transformation *class2RDBMS*

Les trois étapes de la transformation apparaissent clairement dans le corps de l'*operation* (ligne 29 à 46). Premièrement les *Tables* sont créées pour chaque *Class* persistantes (lignes 30 à 35). Ensuite, les *Columns* sont créées dans les tables (ligne 39 à 42) et, finalement, les *FKeys* sont mises à jour (lignes 46). Il est aussi possible d'observer la création de la trace locale et la création de liens de traces (par exemple, ligne 34). Les autres *operation*, par exemple, *createColumns* ou *createFKeyColumns* ont été implémentées, mais ne sont pas représentées ici. L'implémentation Kermeta de la transformation *class2rdbms* est composée de 113 lignes de code et de 11 *operation*.

En utilisant les opérateurs de mutation dédiés aux transformations de modèles, 200 mutants ont été créés. Le nombre de mutants créés par opérateur de mutation est résumé dans la table 15. Seul l'opérateur *RCCC* n'est pas appliqué sur la transformation *class2rdbms* puisqu'il n'existe aucune relation d'héritage dans le méta-modèle de sortie. On peut ainsi voir par exemple que 12 mutants ont été créés en appliquant l'opérateur *RRAC* sur la transformation initiale.

TYPE D'OPÉRATEUR	OPERATEUR DE MUTATION	NOMBRE DE MUTANTS
Navigation	RRAC	12
	RRMC	9
	MSNA	72
	MSNM	12
Filtrage	MFCP	38
	MFCM	18
	MFCA	19
Création	MMRR	11
	MRCA	9

TABLE 15: Nombres de mutants par opérateurs de mutation créés pour *class2rdbms*

Pour les expérimentations, le processus d'analyse de mutation est lancé sur les 200 mutants avec un jeu de 8 modèles de test manuellement produits. Nous avons utilisé la version 0.4.1 de Kermeta. Cette version est relativement ancienne mais est la même que celle utilisée dans [90]. Dans la suite du chapitre, nous présentons en détail la création d'une

nouvelle donnée de test avec notre algorithme, dédié à *tuer* un mutant produit à partir de l'opérateur de mutation *RRMC*.

11.4.4 *Modélisation des mutants : exemple du mutant RRMC_mutant_33*

Parmi tous les mutants, nous allons nous focaliser sur le mutant numéro 33, produit à partir de l'opérateur de mutation *RRMC* et portant le nom *RRMC_mutant_33* [26]. La figure 117 montre le modèle de l'opérateur qui a permis sa construction. Ce modèle précise, grâce à sa référence *initNavigation*, qu'initialement, une référence *dest* d'un élément était naviguée et, grâce à sa référence *newNavigation*, que sa mutation entraîne la navigation de la référence *src* à la place. Si nous regardons en détails le méta-modèle d'entrée de la transformation, nous pouvons voir que ces références appartiennent à la classe *Association*.

26. La transformation associée à ce mutant est la transformation *RRMC_class2RDBMS_33*.

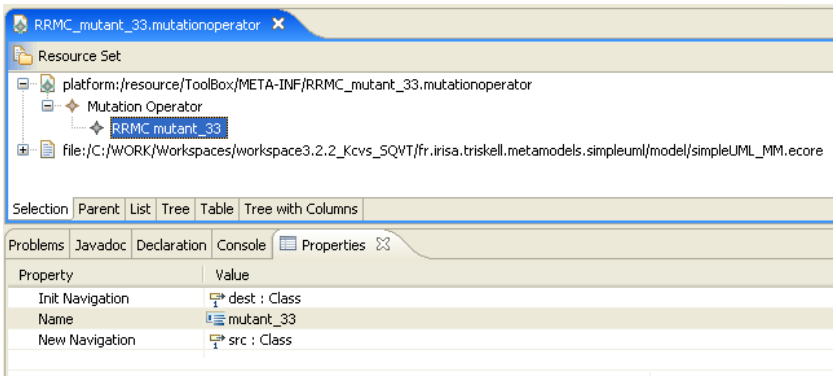


FIGURE 117: Modélisation du mutant *RRMC_mutant_33*

Ce modèle correspond, dans la transformation mutée, au morceau de code présenté au listing 11.2. La ligne 2 présente l'instruction initiale et la ligne 3 l'instruction mutée par l'application de l'opérateur de mutation *RRMC*. On retrouve sur cet extrait de code l'information donnée par le modèle, à savoir que, initialement, la référence naviguée est *dest* et que la référence naviguée après la mutation est *src*.

```
1  operation createColumnsForAssociation(table : Table, asso :
       Association, prefix : String) is do
2    // if isPersistentClass(asso.dest) - initial
3    if isPersistentClass(asso.src) then // mutant

15   else

17   end
18 end
```

Listing 11.2: Extrait du mutant *RRMC_class2RDBMS_33*

11.4.5 *Première exécution du processus d'analyse de mutation*

Pour débuter le processus d'analyse de mutations, les modèles de test sont successivement exécutés sur la transformation testée ainsi que sur l'ensemble des mutants créés. Une fois les exécutions terminées, un oracle indique, pour tous les mutants existants, quels sont les modèles de test qui les ont *tués* et ceux qui les ont laissé *vivants*. Les différents

résultats donnés par l'oracle sont conservés dans un modèle de matrice de mutation. La figure 118 présente un extrait du modèle de matrice de mutation produit. L'extrait présenté met en évidence les retours de l'oracle obtenus pour le mutant *RRMC_mutant_33*. On y retrouve les informations concernant le mutant, c'est-à-dire :

- le nom de la règle qu'il modifie : `createColumnsForAssociation`,
- le nom de la transformation mutante : *RRMC_class2RDBMS_33*,
- le chemin vers la transformation,
- un lien vers la modélisation du mutant proposé.

Finalement, on trouve le résultat annoncé par l'oracle, à savoir si le mutant a été *tué* ou non par un modèle de test, par le biais de la référence *Cells*. Par exemple, pour ce mutant, on peut voir que chacune des *Cell* auxquelles il est associé possède la valeur *true*. Cela signifie que chaque modèle de test a laissé le mutant *vivant*. Comme aucun modèle de test n'a *tué* le mutant, celui-ci est considéré comme étant *vivant*.

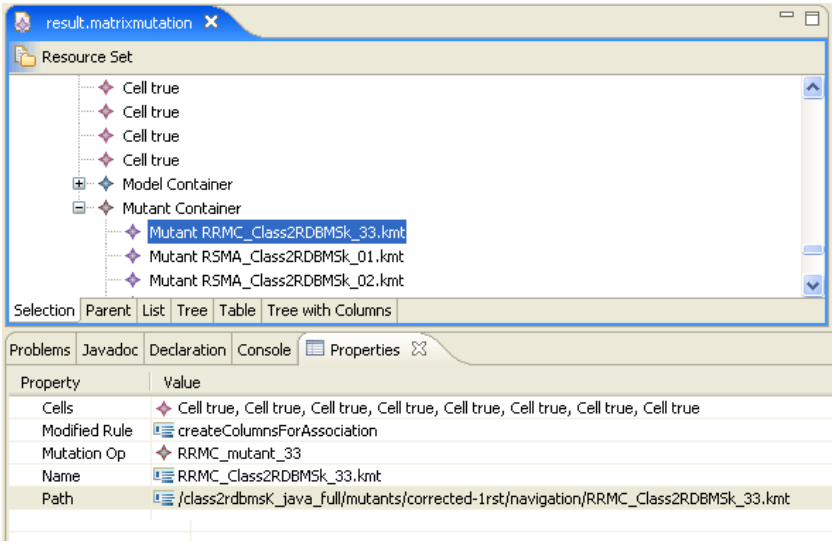


FIGURE 118: Extrait de la matrice de mutation produite

L'analyse de la matrice de mutation indique que 144 mutants ont été *tués* et que 56 sont restés *vivant*, ce qui donne un score de mutation de 72%[27], ce qui peux être suffisant selon le niveau d'exigence du testeur, mais qui ne l'est pas dans notre cas. Pour fournir une nouvelle donnée de test et atteindre un meilleur score de mutation, nous allons essayer de fournir un modèle *tuant* le mutant *RRMC_mutant_33*, choisi arbitrairement.

27. Pour les 200 mutants et les 8 modèles de test exécutés.

11.4.6 *Identification des modèles et des parties des modèles pertinent*

Après avoir choisi un mutant vivant, nous utilisons notre algorithme d'identification de modèles pertinents. L'algorithme est exécuté en prenant en entrée un *Mutant* de la matrice de mutation. Dans notre exemple, c'est le *Mutant RRMC_class2RDBMS_33*, en surbrillance à la figure 118, sur lequel l'algorithme est appliqué.

Le résultat de l'algorithme est donné dans la table 16 résumant les modèles et les éléments des modèles d'entrée et de sortie qui sont identifiés comme pertinents.

MODÈLE DE TEST	ÉLEMENTS SOURCE	ÉLEMENTS DESTINATION
ClassModel02	*next : Association*	*next : FKey*
ClassModel04	*a : Association*	*a : FKey*
ClassModel05	*b : Association*	*b : FKey*
ClassModel06	*a : Association* *b : Association*	*a : FKey* *b : FKey*

TABLE 16: Modèles et éléments pertinent pour le mutant *RRMC_mutant_33*

Sur l'ensemble de 8 modèles de test présent initialement dans le jeu de tests, 4 ont été identifiés comme pertinents : les modèles *ClassModel02*, *ClassModel04*, *ClassModel05* et *ClassModel06*. Ces modèles possèdent des éléments qui sont effectivement passés par la règle problématique, mais qui ne *tue* pas le mutant considéré.

11.4.7 *Analyse des modèles de test*

Afin de comprendre pourquoi le mutant n'est pas tué, il faut analyser plus en détail les modèles d'entrée retenus par l'algorithme. À partir des éléments pertinent identifiés, la détection automatique des cas problématiques pour l'opérateur de mutation *RRMC* est initiée. Les résultats de l'algorithme, comprenant le numéro du cas problématique est résumé dans la table 17.

MODÈLE DE TEST	NUMÉRO DU CAS PROBLÉMATIQUE
ClassModel02	1

TABLE 17: Cas problématiques identifiés pour le mutant *RRMC_mutant_33*

Cette fois, un seul modèle est identifié. Le cas posant problème détecté est le cas numéro 1 de l'opérateur de mutation *RRMC*. Ce cas précise que l'instance récupérée par la navigation originale de la transformation et celle mutée par l'opérateur *RRMC* est la même. Si l'on croise les informations obtenues avec celles retournées par notre algorithme d'identification des modèles pertinents, cela signifie, pour le modèle de test *ClassModel02*, que l'*Association next* possède sa référence *src* et sa référence *dest* qui pointent vers la même instance.

11.4.8 *Création d'une nouvelle donnée de test*

Après identification du cas problématique pour le modèle de test conservées, celui-ci est copiée, puis modifiée. La modification que nous proposons pour ce cas est la suivante : il suffit d'ajouter une nouvelle instance du type *Class* dans le modèle, radicalement différents de la *Class* récupérée par le mutant et la transformation originale. Puis, il faut mettre à jour, soit la référence *src*, soit la référence *dest* pour pointer vers la nouvelle instance créée.

La classe récupérée par l'association originale et l'association mutée est la classe nommée *customer* dont l'attribut *is_persitent* vaut *true*. Nous créons donc une nouvelle instance de *Class*, que nous appelons *newInstance1*. Compte tenu des contraintes structurelles exprimées sur

le modèle, nous devons obligatoirement ajouter à cette nouvelle *Class* un *Attribute* qui doit être différent de celui contenu par la *Class customer*. La figure 119 représente le modèle que nous venons de créer. La nouvelle instance *NewInstance1* possède son attribut *is_persistent* mis à *false* et contient un *Attribute newAttribut* de type *customer*.

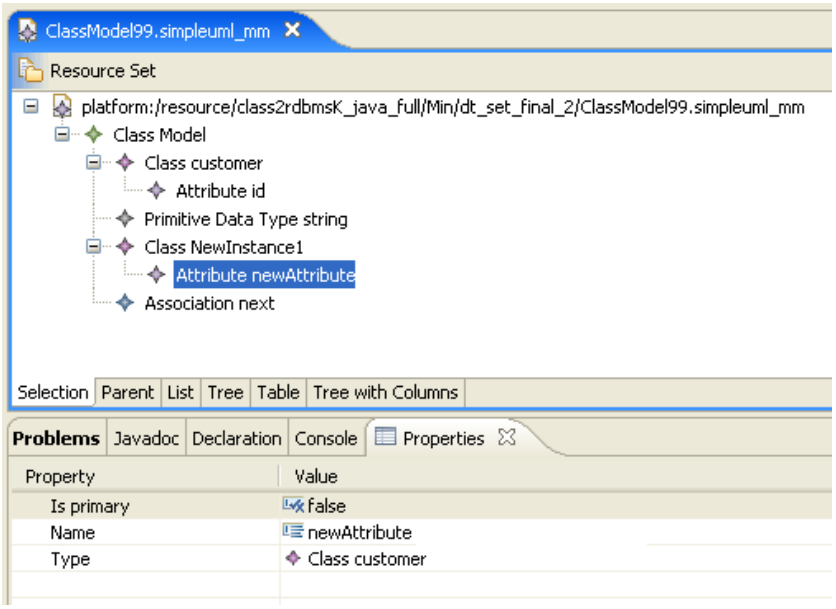


FIGURE 119: Nouveau modèle de test produit

Une fois la nouvelle donnée de test ajoutée au jeu de modèles de test, le processus d'analyse de mutation est de nouveau lancé. Cette fois, le score de mutation atteint est de 78.5%, ce qui est encore inférieur à la barrière des 90% que nous nous sommes fixée. Toutefois, lorsque nous regardons à nouveau la matrice de mutation, partiellement représentée à la figure 120, nous voyons cette fois que le *Mutant RRMC_mutant_33* ne possède pas l'ensemble de ses *Cell* à *true*. En effet, une de ses *Cell* est placée à *false*, ce qui indique que le mutant que nous considérions est, désormais, *tué*.

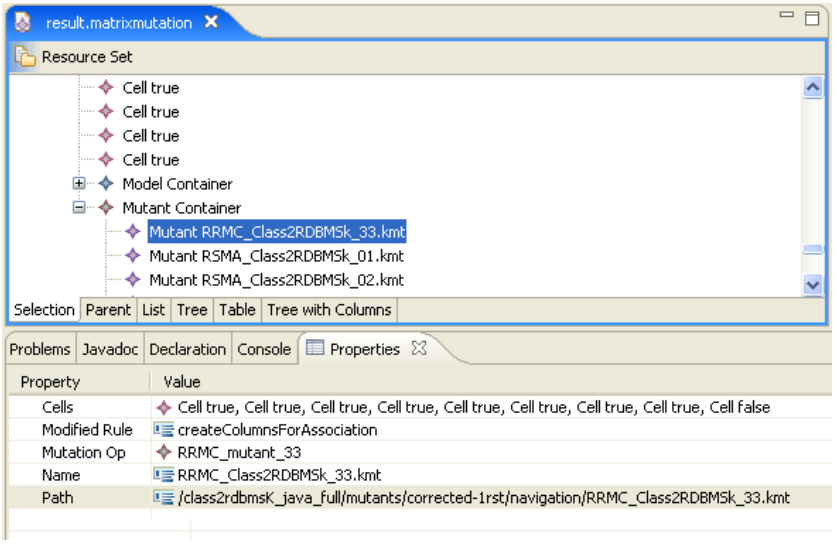


FIGURE 120: Extrait de la matrice de mutation produite

11.4.9 *Production de nouveaux modèles de test*

Cette fois, nous avons choisi d'étudier le mutant *MFCA_mutant_10* obtenu par application de l'opérateur de filtrage *MFCA* sur la référence *attrs* du méta-modèle d'entrée. Le résultat de l'analyse de notre assistant nous est donné dans la table 19 indiquant, pour le mutant étudié, le nombre de modèles jugé pertinents et, parmi les modèles pertinents, le nombre de modèles comportant un cas problématique pour l'opérateur *MFCA*.

MUTANT	#MODÈLES PERTINENTS	#CAS PROBLÉMATIQUES
MFCA_mutant_10	5	5

TABLE 18: Nombre de modèles pertinent et nombre de cas problématiques observés pour le mutant *MFCA_mutant_10*

La table indique que parmi les 9 modèles de test, 5 modèles sont pertinents et que tous ces modèles possèdent une configuration problématique qui empêche de *tuer* le mutant. Exactement, la configuration problématique qui nous est retournée par notre assistant d'analyse de mutant est le second cas identifié en section 11.3.4 pour les éléments de type *Attribute*. Pour mémoire, ce cas indique que les éléments des collections filtrée ne possèdent qu'un seul élément. La solution que nous mettons en œuvre consiste à ajouter un élément dans la collection filtrée, soit la collection *attrs*. Nous copions donc un de ces modèles, puis nous le modifions pour produire un nouveau modèle de test que nous ajoutons au jeu de modèles de test avant de lancer, à nouveau, le processus d'analyse de mutation. Avec 10 modèles de test, le score de mutation atteint est de 89%, ce qui est proche du seuil que nous nous sommes fixé, mais encore insuffisant. Cette fois aussi, nous observons que le mutant que nous considérions *MFCA_mutant_10* est effectivement tué grâce au nouveau modèle de test que nous avons introduit.

Nous passons alors à l'étude d'un nouveau mutant et nous tentons de produire un nouveau modèle de test. Nous choisissons d'étudier le mutant *MSNA_mutant_09* obtenu par application de l'opérateur de mutation *MSNA*, ajoutant une navigation inutile *parent* à une navigation *dest* initiale. Comme pour l'opérateur précédent, le résultat de l'analyse est fourni dans la table 19.

MUTANT	#MODÈLES PERTINENTS	#CAS PROBLÉMATIQUES
MSNA_mutant_09	4	3

TABLE 19: Nombre de modèles pertinent et nombre de cas problématiques observés pour le mutant *MFCA_mutant_10*

Cette fois, parmi les 10 modèles de test, 4 sont pertinents et parmi ces 4 modèles, seulement 3 possèdent une configuration qui pose problème. La configuration problématique qui est trouvé pour les 3 modèles est la configuration numéro 3 [28] indiquant que les instances finalement navigué possèdent un attribut de même valeur. L'assistant précise que l'attribut équivalent trouvé est l'attribut *is_persistent*. La solution prodiguée dans ce cas est la modification d'une de la valeur d'un des deux attributs atteint par la navigation. En suivant ces indications,

28. *C.f*, section 11.3.1.

un des deux modèles est copié puis modifié pour fournir un 11ème modèle de test. Pour vérifier l'utilité de ce nouveau modèle de test, le processus d'analyse de mutation est lancé à nouveau. Cette fois, le score de mutation obtenu est de 91.5%. Le modèle de test que nous avons produit *tue* bien le mutant considéré et le seuil que nous nous étions fixé est dépassé, le processus d'analyse de mutation se termine et le jeu de modèles de test que nous avons obtenu possède 11 modèles. Finalement, dans cette section, nous avons présenté sur un cas d'étude notre approche pour permettre un premier pas vers une automatisation de la production de nouveaux modèles de test.

11.4.10 *Comparaison avec les résultats obtenus dans [90]*

Afin de nous comparer aux travaux proposé dans [90], où l'étape d'amélioration du jeu de modèles de test était effectuée manuellement, nous avons établi deux critères. Le premier est le temps nécessaire à la création d'un ième modèle de test. Le second est le nombre de modèles créés pour dépasser le ratio de 90%.

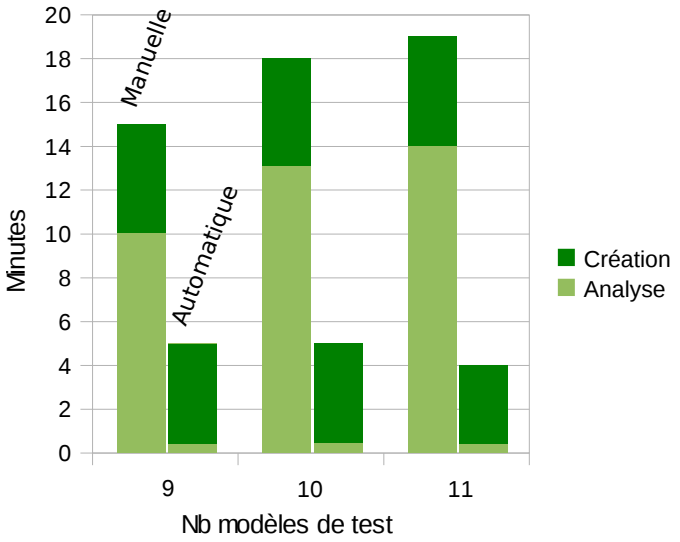


FIGURE 121: Temps nécessaire pour créer une nouvelle donnée de test

L'histogramme présenté en figure 121 résume, approximativement, les temps que nous avons obtenus lors de la création d'un nouveau modèles de tests de manière manuelle, puis en utilisant notre assistant pour passer du jeu de 8 modèles de test initial à 9, puis du jeu de 9 modèles de test à 10 et finalement de 10 modèles de test à 11. Le temps pris en compte est relatif à la création d'une nouvelle donnée de test à partir de la lecture de la matrice de mutation. Cela revient à considérer une phase d'analyse, c'est-à-dire, la récolte des données et l'analyse du mutant et une phase de création du nouveau modèle de test.

Pour passer d'un jeu de 8 modèles de test à un jeu de 9 modèles de test, l'approche manuelle a pris, environ, 15 minutes, avec notre assistant, 5 minutes ont suffit pour créer le 9ème modèle de test [29]. Nous pouvons voir que l'activité la plus chronophage de l'approche manuelle est l'analyse du mutant. Environ 10 minutes lui sont attribuées. Cette partie d'analyse devient de plus en plus complexe lorsque le nombre de modèles de test grandi et prend de plus en plus de temps. Grâce à l'utilisation de notre assistant, la partie d'analyse est accélérée et

29. Le nouveau modèle de test créé en utilisant notre assistant n'est pas exactement le même que celui introduit par l'analyse manuelle, mais il présente des similitudes dans sa structure.

passe sous la barre des 1 minutes [30]. Finalement, pour dépasser le ratio de 90% de mutants *tués* avec l'approche manuelle, il faut environ 52 minutes contre, environ, 14 minutes en utilisant notre assistant [31].

On peut se rendre compte que la création de la nouvelle donnée de test est l'activité la moins vorace en temps lorsque l'approche manuelle est utilisée, mais cette tendance s'inverse avec l'utilisation de notre assistant. Dans les deux cas, nous observons une moyenne de 5 minutes pour la création d'une nouvelle donnée de test [32]. Cette phase de création d'un nouveau modèle de test en fonction des retours de notre assistant est donc la phase à améliorer pour optimiser cette étape d'amélioration du jeu de modèles de test du processus d'analyse de mutation.

Nous avons aussi comparé notre approche par rapport à celle présentée dans [90] en fonction du nombre de nouveaux modèles de test créés pour passer la barrière des 90%. En utilisant l'approche manuelle et notre assistant, dans les deux cas, nous avons créé 3 nouveaux modèles de test. Ce nombre n'a pas été amélioré par notre assistant, car son but est d'aider le testeur à créer un nouveau modèle de test pour *tuer* un mutant en particulier, entraînant parfois la mort d'autres mutants par effet de bord. Cela peut laisser présager qu'en utilisant notre assistant, le jeu de modèles de test final pourrait contenir autant de modèles de test que de mutants et qu'il serait plus important que celui obtenu manuellement. En considérant les travaux, aussi proposés dans [90], les auteurs proposent d'effectuer automatiquement une réduction du jeu de modèles de test par réduction de la matrice de mutation en partant du principe que si les mutants *tués* par un modèle de test le sont aussi par un autre, ce modèle de test peut être retiré de l'ensemble du jeu de modèles de test. En utilisant cette algorithme de réduction, il est donc possible de réduire le jeu de modèles de test obtenus en utilisant notre approche et ainsi fournir un jeu de tests minimal.

30. Le nombre croissant de modèles dans le jeu de tests ne viens pas sensiblement augmenter ce temps.

31. Nous ne comptons ici que les temps d'analyse et de création. La phase d'exécution des mutants durant sensiblement le même temps dans les deux cas.

32. La dernière création prend moins de temps, car la modification à effectuer est mineure.

11.5 CONCLUSION

Dans ce chapitre, nous avons présenté l'ensemble des travaux que nous avons mené sur l'analyse de mutation adapté aux transformations de modèles et plus particulièrement sur l'étape d'amélioration du jeu de modèles de test. Nous avons notamment fourni un moyen de modéliser les mutants indépendamment de tout langage de transformation et deux algorithmes dédiés. Nous avons montré comment l'utilisation de la trace locale, couplée à ces deux différents algorithmes, peut aider à la production de nouveaux modèles de test à partir des modèles de test déjà existant.

Pour assister le testeur lors de la phase de création de nouveaux modèles de test du processus d'analyse de mutation, nous avons proposé deux algorithmes, tout deux reposant sur l'utilisation de la trace locale. Les deux algorithmes sont utilisés l'un après l'autre pour produire un modèle de test *tuant* un mutant *vivant*. Le premier algorithme proposé s'occupe de sélectionner des modèles de test pertinents parmi ceux présents dans le jeu de modèles de test initial et permet ainsi de faire un premier tri dans les modèles de test. Le second algorithme propose une analyse plus fine, reposant sur la modélisation du mutant étudié, pour fournir un moyen de modifier un modèle de test existant et produire un nouveau modèle de test tuant le mutant considéré.

Afin de présenter notre approche, nous avons détaillé l'activité d'amélioration du jeu de modèles de test pour la transformation *class2rdbms* écrite en KERMETA, sur un jeu de 6 modèles de test pour atteindre un score de mutation de 90%. Le jeu de tests initial, possédant un score de 72%, a nécessité l'ajout de 3 modèles de test pour dépasser le seuil fixé de 90%. La création des nouveaux modèles de test s'est faite grâce aux retours de nos algorithmes qui ont permis à chaque fois de produire rapidement de nouveaux modèles de test.

Nous avons ensuite comparé notre approche à celle manuelle en terme de temps passé pour arriver au seuil de 90% et du nombre de nouveaux modèles de test créés. Nous avons observé qu'en utilisant notre approche, pour dépasser les 90%, 14 minutes ont été nécessaires contre 52 minutes en utilisant l'approche manuelle.

L'approche que nous avons proposée ici pour l'activité d'amélioration du jeu de modèles test est un premier pas vers une automatisation complète de l'analyse de mutation. Actuellement, nous n'avons pas pu fournir à tous les coup une modification automatique des modèles de test existant pour créer un nouveau modèle de test. En effet, pour pouvoir créer automatiquement un nouveau modèle de test par modification, il faut tenir compte des contraintes structurelles du méta-modèle d'entrée. Les challenges soulevés pour obtenir une génération automatique de modèles de test sont alors les mêmes que ceux que l'on peut retrouver pour les problématiques de génération de jeu de modèles de test.

CONCLUSION GÉNÉRALE

BILAN

Les travaux présentés dans ce document s'inscrivent dans le cadre des recherches menées en IDM et plus particulièrement pour la correction et l'optimisation de modèles, ainsi que pour le test de transformations de modèles et de chaînes de transformations. Il en résulte une proposition pour la création d'outils pour les développeurs et les utilisateurs de chaînes de compilation IDM. Leurs mises en œuvre ont nécessité des contributions dans différents domaines de l'IDM.

Mécanisme de trace

Les travaux que nous avons présentés dans cette thèse reposent tous sur le concept de traçabilité. Pour remédier aux manques identifiés dans les mécanismes de trace existants, nous en avons proposé un nouveau, reposant sur deux méta-modèles spécifiques : un méta-modèle de trace locale et un méta-modèle de trace globale. Le premier méta-modèle est utilisé pour garder les liens de trace pour une unique transformation de modèles à modèles et satisfait les 6 exigences suivantes :

- approche orientée marquage de trace,
- capture des règles de transformation,
- lien de trace $m - n$,
- trace des attributs,
- indépendant de tout langage de transformation.

En complément de ce méta-modèle de trace locale, le méta-modèle de trace globale permet de gérer la trace dans les chaînes de transformations et satisfait les 3 exigences suivantes :

- séparation des préoccupations,
- approche dédiée uniquement à la trace,
- indépendance de toute plate-forme.

Optimisation et correction de modèles

Lorsque une application est modélisée, il est crucial de s'assurer que le comportement de cet application et ses performances sont en adéquation avec les attentes du concepteur. Pour déterminer les fautes ou les problèmes de performance dans les modèles, nous avons proposé deux approches reposant sur la récupération d'informations obtenues à l'exécution de l'application générée. Ces informations sont ensuite traitées et remontées, grâce à la trace, sur les modèles de conception pour fournir un retour au concepteur directement sur les artefacts qu'il manipule.

Même si ces deux approches que nous avons proposées tirent partie d'un même mécanisme de remontée d'informations sur les modèles de conception, elles diffèrent sur le traitement des informations et le moyen de les obtenir. D'un côté, pour l'optimisation de performances, les informations sont obtenues par un outil de *profiling* existant et sont traitées par un système expert dans le but de fournir un conseil sur la manière de modifier les modèles de conception. D'un autre côté, pour la

correction de modèles, les informations d'exécution sont obtenues à la demande du concepteur grâce à un mécanisme d'instrumentation et ces informations sont retournées en l'état sur les modèles de conception.

Localisation d'erreurs

Dans la quatrième partie de cette thèse, nous avons levé l'hypothèse forte qui avait guidé nos précédents travaux, à savoir que les chaînes de transformations sont dignes de confiance.

Lors d'une activité de correction de transformations de modèles, il est souvent difficile d'identifier la place d'une faute. Ce problème est encore plus perceptible lorsque l'erreur est perdue dans une chaîne de transformations. Nous avons alors proposé un algorithme de localisation d'erreurs reposant sur la trace. Il permet de donner une aide rapide au développeur afin de déterminer les règles d'une transformation ou les règles dans une chaîne de transformations qui sont potentiellement fautives et qui méritent d'être vérifiées.

Analyse de mutation

Nous avons également menés des travaux sur la technique d'analyse de mutation et plus particulièrement sur l'activité d'amélioration du jeu des modèles de test.

Nous avons automatisé en grande partie cette activité préalablement toujours effectuée manuellement. Nous avons alors proposé un assistant, reposant sur notre mécanisme de trace et une modélisation des opérateurs de mutation, pour aider le développeur lors de cette activité fastidieuse qui est bien souvent un frein à l'utilisation de la technique d'analyse de mutation.

Validations expérimentale

Tout au long de cette thèse, nous avons eu l'opportunité d'implémenter et de mettre à l'épreuve nos outils pour l'optimisation et la correction de modèles ainsi que pour le test des transformations de modèles à modèles et les chaînes de compilation IDM.

Plus particulièrement, nous nous sommes intéressé :

- à l'application de *DownScaler* avec la chaîne de compilation OpenCL,
- à l'application de la multiplication de matrices avec la chaîne de compilation PThread,
- à la chaîne de compilation OpenCL,
- à la transformation *SimpleUml* vers *RDBMS*.

Les trois premiers cas d'études, effectué dans l'environnement Gaspard2, nous ont permis d'éprouver nos approches pour les chaînes de compilations IDM, aussi bien du point de vue du concepteur, que du point de vue du développeur.

Le dernier cas d'étude a été effectué avec le langage Kermeta pour bénéficier d'un support déjà existant pour la technique d'analyse de mutation. Nous nous sommes alors appuyé sur un cas d'étude proposée dans [90] pour nous permettre de tester à large échelle et de comparer nos résultats.

PERSPECTIVES

Les perspectives aux travaux que nous avons proposé dans cette thèse sont relatives à trois contextes :
- le contexte de la traçabilité,
- le contexte de l'optimisation de modèles,
- le contexte du test de transformations.

Au même titre que nos contributions qui ont été implémentées au sein de GASPARD2, les perspectives que nous proposons ici peuvent, elles aussi, venir enrichir l'environnement.

Prise en compte de la traçabilité de modèles vers texte

Dans ce document, le mécanisme de trace que nous avons proposé est relatif uniquement aux transformations de modèles. Ce mécanisme de trace, associés aux UIDs est largement suffisant pour l'optimisation et la correction de modèles.

Concernant l'algorithme de localisation d'erreurs, il travaille uniquement sur les traces de transformations de modèles à modèles. Or s'il aide à rendre fiables les transformations de modèles à modèles, des erreurs peuvent persister dans la chaîne puisque les transformations de modèles vers texte n'ont pas été testées. De plus, il est difficile pour un développeur de localiser manuellement ces erreurs dans les transformations de modèles vers texte.

Afin de prendre en compte la traçabilité de modèles vers texte, il faudrait pouvoir proposer pour la trace locale un mécanisme annexe de gestion de blocs et de lignes. De cette façon, en utilisant l'algorithme de localisation d'erreurs, il serait possible de chercher une faute dans l'intégralité d'une chaîne de compilation, transformation de modèles vers texte comprise. Ainsi, des travaux déjà existant sur les oracles de programmes pour identifier l'erreur au niveau du code source de l'application générée pourraient être réutilisés.

Automatisation accrue de l'analyse de mutation

Pour fournir une analyse d'un mutant en cours d'étude, l'assistant se sert d'un modèle de l'opérateur de mutation. Ce modèle d'opérateur de mutation précise, pour un mutant sur les méta-modèles d'entrée et de sortie de la transformation testée, quelle est la modification impactée par l'opérateur. Actuellement, les modèles des opérateurs de mutations sont données manuellement en fonction des mutants déjà créés, ce qui donne un travail double au testeur. Pour éviter cette double charge de travail, il pourrait être intéressant d'utiliser la modélisation des opérateurs pour directement créer les mutants, par exemple via une HOT. Ainsi, le testeur pourrait décrire directement sur les méta-modèles d'entrée et de sortie les opérations qu'il aimerait effectuer pour créer un mutant. Une autre perspective pour le travail que nous avons présenté réside dans la production systématique et automatique des nouveaux modèles de test en s'appuyant sur les retours de notre assistant. Les challenges alors levés sont alors équivalents à ceux existant pour la génération de jeu de modèles de test.

Amélioration des traitements des informations d'exécution

Concernant notre approche d'optimisation de modèles, actuellement, le système expert est capable de générer des conseils pour certains critères de performances. Dans notre cas d'étude, nous avons augmenté le taux d'occupation du GPU, mais nous n'avons pas atteint les 100%. Après modification du modèle, le conseil généré était toujours relatif à l'augmentation de l'occupation alors qu'il nous était impossible de passer au delà du taux atteint. Dans ce contexte, il serait pertinent d'améliorer le système expert pour qu'il puisse croiser les informations de performance et indiquer automatiquement le critère le plus intéressant à améliorer.

Les travaux d'observation partielle et d'optimisation de modèles reposent en partie sur l'exploitation d'un *log* obtenu lors de l'exécution d'une application. Actuellement, les entrées de ce *log* sont traitées indépendamment. Une des évolutions possible sur cette partie serait alors de proposer une analyse des entrées du *log* pour en identifier les séquences. Cette analyse pourrait servir à la déduction de certains comportements particuliers, comme les *deadlocks*, et aider le concepteur à revoir sa modélisation.

Compositions d'observations

Nous avons proposé un moyen d'observer l'exécution d'une application sur les modèles de conception qui ont mené à sa génération. Dans cette approche, le concepteur de modèles utilise un profil d'observation, pouvant s'apparenter à une bibliothèque d'observations, pour spécifier les éléments qu'il aimerait observer. Le profil d'observation est donné au concepteur par le développeur de la chaîne qui doit prévoir au maximum les attentes du concepteur. Il serait alors intéressant de fournir au concepteur un moyen de pouvoir composer des observations existantes pour en créer de nouvelles et ainsi lui permettre d'affiner les informations à récupérer à l'exécution.

Cinquième partie

ANNEXES

A

OPÉRATEURS DE MUTATION

Dans cette annexe, nous présentons les méta-modèles des opéteurs de modélisation proposés par Mottu et al. Les méta-modèles sont définis en fonction de l'EMOF.

A.1 MUTATION OPERATORS RELATED TO THE *navigation*

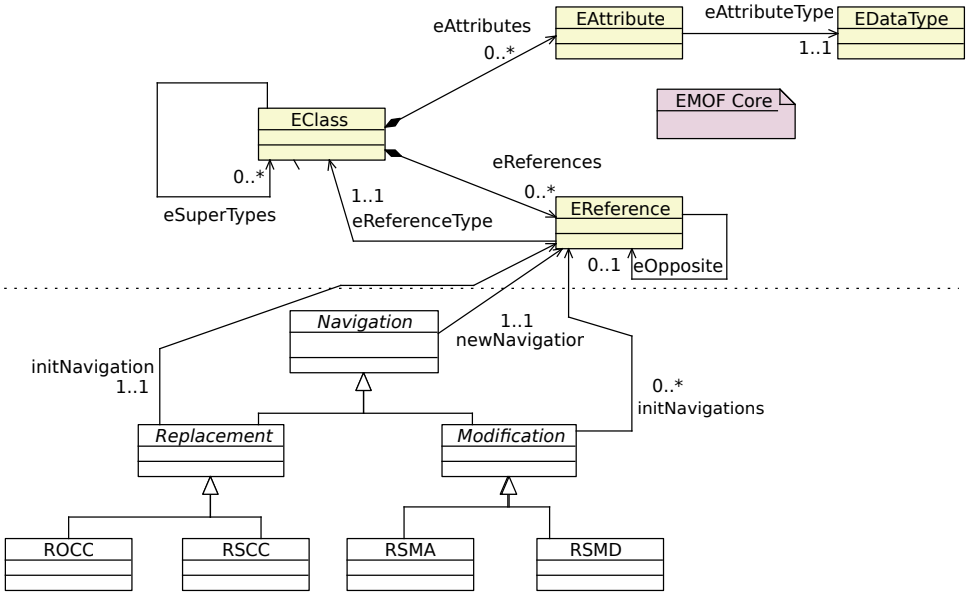


FIGURE 122: Navigation Operators Metamodels

A.1.1 *Relation to the Same Class Change (RSCC) Operator*

DEFINITION "The RSCC operator replaces the navigation of one association towards a class with the navigation of another association to the same class."

METAMODEL DESCRIPTION Table 20 gathers the metaclasses and the references specific to the RSCC operator and explains them.

METACLASS / RELATION	DESCRIPTION
RSCC	The mutation operator
initNavigation	Reference initially navigated by the transformation
newNavigation	Reference navigated after the mutation

TABLE 20: Description of the RSCC Operator Metamodel

A.1.2 *Relation to anOther Class Change (ROCC) Operator*

DEFINITION "The ROCC operator replaces the navigation of an association towards a class with the navigation of another association to another class."

METAMODEL DESCRIPTION The ROCC operator alters the reference navigated in the original transformation with another pointing to another metaclass.

A.1.3 *Relation Sequence Modification with Deletion (RSMD) Operator*

DEFINITION "During the navigation, the transformation can navigate many relations successively. This operator removes the last step off from the composed navigation."

METAMODEL DESCRIPTION The original navigation sequence modified by the RSMD operator is designed by the *ordered initNavigations* reference. This operator deleting the last navigation of an existing sequence, the new last navigation is specified by the *newNavigation* reference. It corresponds to the penultimate reference in the *initNavigations* sequence.

METACLASS / RELATION	DESCRIPTION
RSMD	The mutation operator
initNavigations	Sequence of references initially navigated by the transformation
newNavigation	Last reference of the sequence

TABLE 21: Description of the RSMD Operator Metamodel

A.1.4 *Relation Sequence Modification with Addition (RSMA) operator*

DEFINITION "This operator does the opposite of RSMD". It adds a navigation to an original navigation sequence.

METAMODEL DESCRIPTION The descriptions of the RSMD and RSMA méta-modèles are the same. Only the semantic of the *newNavigation* varies. In the RSMA operator, the *newNavigation* reference corresponds to the navigation added to the existing sequence referred by *initNavigations*.

A.2 MUTATION OPERATORS RELATED TO THE *filtering*

Filtering manipulates collections to select only the elements useful for the transformation. In a general way, a filter may be considered as a guard on a collection, depending on specific criteria. These guards can be altered from different ways leading to the definition of mutation operators : *CFCP*, *CFCD* and *CFCA*.

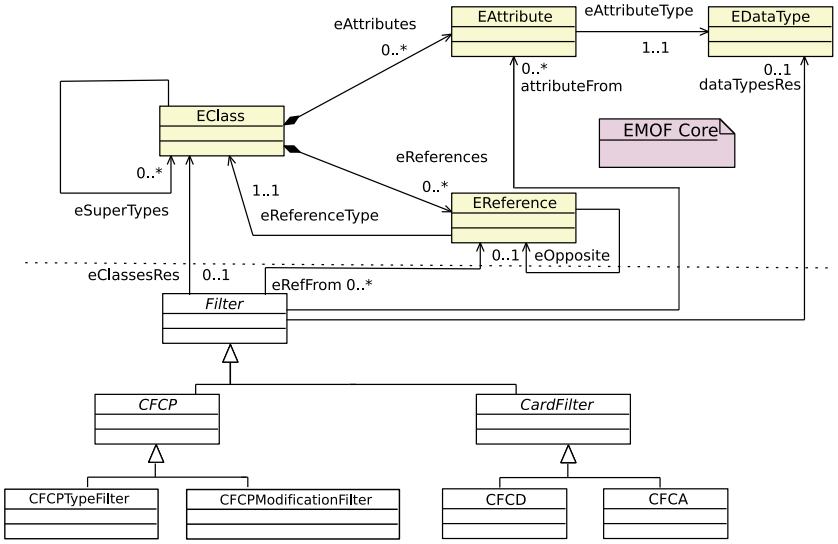


FIGURE 123: Filtering Operators Metamodels

A.2.1 *Collection filtering change with perturbation (CFCP) operator*

DEFINITION "This operators aims at modifying an existing filtering, by influencing its parameters. One criterion could be a property of a class or the type of a class ; this operator will disturb this criterion." As supposed in this definition, an existing filtering can be modified from different ways. Altering the type of a class is only an example. In order to propose a generic modeling of the mutation operators, we decide to divide the CFCP operator introduced by [90] into two operators : *CFCPTypeFilter* and*CFCPModificationFilter*. The first one modifies the filtering by replacing the type of a class by another. The second one modifies the filtering condition for example by replacing a 'AND' by a 'OR' or by negating the condition.

METACLASS / RELATION	DESCRIPTION
CFCPTypeFilter	The mutation operator
CFCPModificationFilter	The mutation operator
eRefFrom	Define the collections of references on which the filtering is applied
attributeFrom	Define the collections of attributes on which the filtering is applied
eClassesRes	Type of the references collections
dataTypesRes	Type of the attribute collections
newEClassesRes	New type of the references collections

TABLE 22: Description of the CFCP Operator Metamodel

METAMODEL DESCRIPTION The filtering is applied either on a reference (*eRefFrom*) or an attribute (*attributeFrom*) collections. In both

cases, the type of the elements contained in the collection is specified (*eClassesRes* or *dataTypesRes*). The CFCPTypeFilter operator replaces the type of the elements on which the filtering is applied by another (*newEClassesRes*) in the type hierarchy. The *CFCPTypeFilter* operator can only be applied on references collection since the dataType hierarchy is flat. The *CFCPModificationFilter* modifies the filtering condition applied either on a reference or an attribute collection. We decide not to design the filtering conditions in the operator méta-modèles, because they often depend on the used transformation language. Thus, the *CFCPModificationFilter* méta-modèle specifies that a filtering condition has been modified without detailing this modification and the filtering condition. As for the other operators, the transformation rule on which the operator is applied is indicated in the mutant itself.

A.2.2 *Collection filtering change with deletion (CFCD) operator*

DEFINITION "This operator deletes a filter on a collection ; the mutant returns the collection it was supposed to filter."

METAMODEL DESCRIPTION In the *CFCD* méta-modèle, the collection on which the operator is applied, is represented by the *eRefFrom* relation or the *attributeFrom* relation. By deleting a filter on the collection, the whole collection is returned and not only one of its subsets. The type of the collection elements is specified by the *eClassesRes* relation or the *dataTypesRes* relation.

A.2.3 *Collection filtering change with addition (CFCA)*

DEFINITION "This operator does the opposite of CFCD. It uses a collection and processes a useless filtering on it. This operator could return an infinite number of mutants, we have to restrict it. We choose to take a collection and to return a single element arbitrarily chosen."

METAMODEL DESCRIPTION The CFCA operator relies on the same classes and references than the CFCD operator. The filtering condition of the original transformation is modified by selecting the first element of the collection. If this collection is not ordered, the first element can be different at each execution.

A.3 MUTATION OPERATORS RELATED TO THE *creation*

Creation mutation operators are relative to the last phase of the transformation process *i.e.* creation or modification of output elements. Three operators of this category have been identified : CCCR, CACD and CACA. They can be expressed on the output méta-modèle since the creation phase only deals with output elements. The méta-modèles of these operators are represented in Figure 124. The upper part of the figure representing the output méta-modèle from a generic point of view is common to the two previous mutation operators categories. The bottom part presents the operator hierarchy. Both parts are linked through references. The following subsections describe each mutation operator related to the creation.

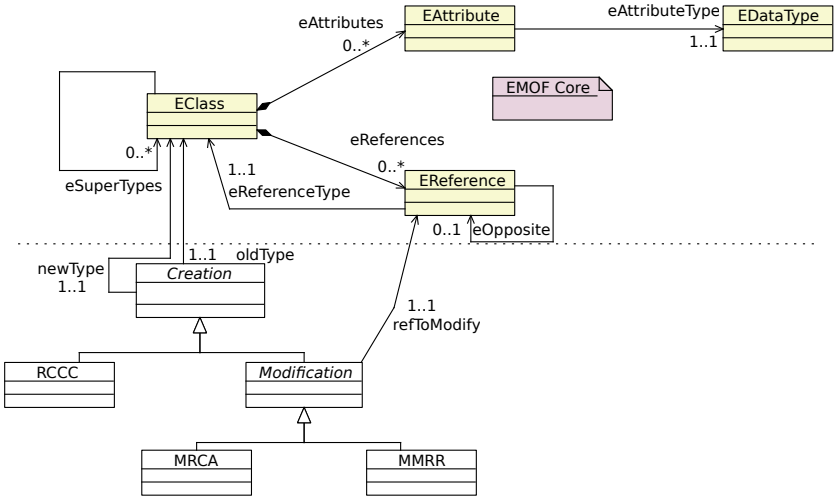


FIGURE 124: Creation Operators Metamodels

A.3.1 *Class Compatible Creation Replacement (CCCR) Operator*

DEFINITION "This operator replaces the creation of an object by the creation of an object of a compatible type. It could be an instance of a child class, of a parent class or of a class with a common parent."

METAMODEL DESCRIPTION Table 23 presents the metaclass and the references specific to the CCCR operator. The operator is applied on an *oldType* class, that represents the original class created by the transformation, and modify it in order to lead to the creation of the *newType* class.

METACLASS / RELATION	DESCRIPTION
CCCR	The mutation operator
oldType	Class created by the original transformation
newType	Class created by the mutated transformation

TABLE 23: Description of the CCCR Operator Metamodel

A.3.2 *Classes' Association Creation Deletion (CACD) Operator*

DEFINITION "This operator deletes the creation of an association between two instances." Consequently, if the original transformation allows the creation of an association between two instances, the mutated transformation does not.

METAMODEL DESCRIPTION Table 24 sums up the description of the *CACD* méta-modèle by describing its specific metaclass and references. For this mutation operator, the *refToModify* reference corresponds to the association which is removed in the output model.

METACLASS/RELATION	DESCRIPTION
CACD	The mutation operator
refToModify	Reference removed by the mutant

TABLE 24: Description of the CACD Operator Metamodel

A.3.3 *Classes' Association Creation Addition (CACA) Operator*

DEFINITION "This operator adds a useless creation of a relation between two class instances of the output model, when the méta-modèle allows it."

METAMODEL DESCRIPTION The méta-modèle designed for this operator is similar to the *CACD* méta-modèle. Only the semantic of the *refToModify* reference is different. It corresponds here to the added association in the output model.

BIBLIOGRAPHIE

[1] Hiralal Agrawal, Joseph R. Horgan, Saul London et W. Eric Wong : Fault Localization using Execution Slices and Dataflow Tests, 1995.

[2] Freddy Allilaire, Jean Bézivin, Marcos Didonet Del Fabro, Frédéric Jouault, David Touzet et Patrick Valduriez : AMMA : vers une Plate-forme Générique d'Ingénierie des Modèles. *Génie logiciel*, 2005.

[3] Bastien Amar, Hervé Leblanc, Bernard Coulette et Clémentine Nebut : Using Aspect-Oriented Programming to Trace Imperative Transformations. *In Enterprise Distributed Object Computing Conferece, Vitoria (Brésil)*, octobre 2010.

[4] Vincent Aranega, Anne Etien et Jean-Luc Dekeyser : Using an Alternative Trace for QVT. *In Workshop on Multi-Paradigm Modeling*, Olso, Norway, 2010.

[5] Vincent Aranega, Jean-Marie Mottu, Anne Etien et Jean-Luc Dekeyser : Using Traceability to Enhance Mutation Analysis Dedicated to Model Transformation. *In Workshop on Model driven Engineering Verification and Validation*, octobre 2010.

[6] Vincent Aranega, Jean-Marie Mottu, Anne Etien et Jean-Luc Dekeyser : Traceability for mutation analysis in model transformation. *In Proceedings of the 2010 international conference on Models in software engineering*, 2011.

[7] Argrawal, DeMillo, Hathaway, Hsu, Krauser, Martin, Mathur et Spafford : Design of mutant operators for the c programming language. Rapport technique, 1989.

[8] Simonetta Balsamo, Antinisca Di Marco, Paola Inverardi et Marta Simeoni : Model-Based Performance Prediction in Software Development : A Survey. *IEEE Trans. Soft. Eng.*, May 2004.

[9] Simonetta Balsamo et Moreno Marzolla : A simulation-based approach to software performance modeling. *SIGSOFT Softw. Eng. Notes*, September 2003.

[10] Mikaël Barbero, Marcos Didonet, Del Fabro et Jean Bézivin : Traceability and provenance issues in global model management. *In ECMDA Traceability Workshop*, 2007.

[11] Ellen Francine Barbosa, José Carlos Maldonado et Auri Marcelo Rizzo Vincenzi : Toward the determination of sufficient mutant operators for c. *Softw. Test., Verif. Reliab.*, 2001.

[12] Benoit Baudry, Franck Fleurey, Jean-Marc Jézéquel et Yves Le Traon : From genetic to bacteriological algorithms for mutation-based testing. *STVR Journal*, juin 2005.

[13] M. Belaunde et G. Dupé : SmartQVT, 2006. http ://sourceforge.net/projects/smartqvt/.

[14] Rabie Ben Atitallah : *Modèles et simulation de systèmes sur puce multiprocesseurs – Estimation des performances et de la consommation d'énergie*. ThÃ¨se de doctorat (PhD Thesis), Laboratoire d'informatique fondamentale de Lille, UniversitÃ© des sciences et technologies de Lille, France, décembre 2007.

[15] J Bézivin, E Breton, G Dupé et P Valduriez : The atl transformation-based model management framework. Rapport technique, University of Nantes, 2003. Research Report TR03-08.

[16] Jean Bezivin, Frederic Jouault et Patrick Valduriez : On the Need for Megamodels. *In Proceedings of Workshop on Best Practices for Model-Driven Software Development at the 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications.*, 2004.

[17] Jean Bézivin, Bernhard Rumpe, Andy Schürr et Laurence Tratt : Model transformations in practice workshop. *In Satellite Events at the MoDELS 2005 Conference*, 2005.

[18] Borland : Qvt - o, 2007. http ://www.eclipse.org/m2m/qvto/doc.

[19] Pierre Boulet : Array-OL Revisited, Multidimensional Intensive Signal Processing Specification. Research Report RR-6113, INRIA, février 2007.

[20] Pierre Boulet : Formal Semantics of Array-OL, a Domain Specific Language for Intensive Multidimensional Signal Processing. Research Report RR-6467, INRIA, mars 2008.

[21] Hugo Bruneliere : Global Model Management Traceability Extension. MODELPLEX Project Report D.3.2.d, INRIA, 2008. URL http://docatlanmod.emn.fr/AM3/Documentation/D3-2-d_Global_Model_Management_Traceability_Extension_v1-0.pdf.

[22] Jean Bézivin, Nicolas Farcet, Jean-Marc Jézéquel, Benoît Langlois et Damien Pollet : Reflective model driven engineering. *In Proceedings of UML 2003*, octobre 2003.

[23] Lukai Cai et Daniel Gajski : Transaction level modeling in system level design. Rapport technique, Center for Embedded Computer Systems, Information and Computer Science, University of California, Irvine, CA, 2003.

[24] S. Callanan, R. Grosu, Xiaowan Huang, S.A. Smolka et E. Zadok : Compiler-assisted software verification using plug-ins. *In Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, april 2006.

[25] A. Cicchetti, D. Di Ruscio, R. Eramo et A. Pierantonio. : Model differences for supporting model co-evolution. *In MoDSE, Workshop on Model-Driven Software Evolution*, 2008.

[26] Antonio Cicchetti, Davide Di Ruscio et Alfonso Pierantonio : A Metamodel Independent Approach to Difference Representation. *JOT : Journal of Object Technology*, octobre 2007.

[27] K. Czarnecki et S. Helsen : Feature-based survey of model transformation approaches. *IBM Syst. J.*, July 2006.

[28] Krzysztof Czarnecki et Simon Helsen : Classification of model transformation approaches. *In OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003.

[29] Andrea Darabos, András Pataricza et Dániel Varró : Towards testing the implementation of graph transformations. *Electron. Notes Theor. Comput. Sci.*, April 2008.

[30] DaRT Team : Graphical Array Specification for Parallel and Distributed Computing (GASPARD2). http ://www.gaspard2.org/, 2009.

[31] Markus Debusmann et Kurt Geihs : Efficient and transparent instrumentation of application components using an aspect-oriented approach. *In Self-Managing Distributed Systems*, Lecture Notes in Computer Science. 2003.

[32] Alain Demeure, Anne Lafage, Emmanuel Boutillon, Didier Rozzonelli, Jean-Claude Dufourd et Jean-Louis Marro : Array-OL : Proposition d'un Formalisme Tableau pour le Traitement de Signal Multi-Dimensionnel. *In Gretsi*, Juan-Les-Pins, France, septembre 1995.

[33] R. A. DeMillo, R. J. Lipton et F. G. Sayward : Hints on test data selection : Help for the practicing programmer. *Computer*, 1978.

[34] Future Research Topics Discussion : ECMDA Traceability Workshop, 2005. URL http://www.sintef.no/upload/10558/Future-Research-Topics.pdf.

[35] Dolev Dotan et Andrei Kirshin : Debugging and testing behavioral uml models. *In Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, OOPSLA '07, 2007.

[36] Nikolaos Drivalos, Dimitrios S. Kolovos, Richard F. Paige et Kiran J. Fernandes : Engineering a DSL for Software Traceability. *Software Language Engineering*, 2009.

[37] James Eagan, Mary Jean Harrold, James A. Jones et John Stasko : Technical note : Visually encoding program test information to find faults in software. *In Proceedings of the IEEE Symposium on Information Visualization 2001 (INFOVIS'01)*, 2001.

[38] Alexander Egyed et Paul Grünbacher : Supporting software understanding with automated requirements traceability. *International Journal of Software Engineering and Knowledge Engineering*, 2005.

[39] Karsten Ehrig, Jochen Küster et Gabriele Taentzer : *Software and Systems Modeling*, 2009.

[40] EMF project : Emf compare. http://www.eclipse.org/emft/projects/compare, 2008.

[41] Anne Etien, Alexis Muller, Thomas Legrand et Xavier Blanc : Combining independent model transformations. *In Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, 2010.

[42] Jean-Rémy Falleri, Marianne Huchard et Clémentine Nebut : Towards a Traceability Framework for Model Transformations in Kermeta. *In ECMDA Traceability Workshop*, 2006.

[43] Jean M. Favre : Megamodelling and Etymology. *In Transformation Techniques in Software Engineering*, 2006.

[44] Jean M. Favre et Jacky Estublier : *L'ingénierie dirigée par les modèles au dela du MDA*, chapitre Concepts de base de l'IDM (Modèle, métamodèle, transformation, mégamodèle). 2006.

[45] Fabiano Cutigi Ferrari, José Carlos Maldonado et Awais Rashid : Mutation testing for aspect-oriented programs. *In ICST '08 : Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, 2008.

[46] F. Fleurey, J. Steel et B. Baudry : Validation in model-driven engineering : testing model transformations. *In Model, Design and Validation Workshop*, nov. 2004.

[47] Franck Fleurey, Benoit Baudry, Pierre-Alain Muller et Yves Le Traon : Towards dependable model transformations : Qualifying input test data. *SoSyM*, 2007.

[48] Franck Fleurey, Benoit Baudry, Pierre-Alain Muller et Yves Le Traon : Qualifying input test data for model transformations. *Software and System Modeling*, 2009.

[49] Martin Fowler, Kent Beck, John Brant, William Opdyke et Don Roberts : *Refactoring : Improving the Design of Existing Code*. 1999.

[50] Piero Fraternali et Massimo Tisi : Mutation analysis for model transformations in atl. *In International Workshop on Model Transformation with ATL*, Nantes, France, juin 2009.

[51] Mathias Fritzsche et Wasif Gilani : Model transformation chains and model management for end-to-end performance decision support. *In Proceedings of the 3rd international summer school conference on Generative and transformational techniques in software engineering III*, GTTSE'09, 2011.

[52] Mathias Fritzsche, Jendrik Johannes, Steen Zschaler, Anatoly Zherebtsov et Alexander Terekhov : Application of Tracing Techniques in Model-Driven Performance Engineering. *In In Proccedings of the 4th ECMDA-Traceability Workshop (ECMDA'08*, juin 2008.

[53] Abdoulaye Gamatié, Sébastien Le Beux, Éric Piel, Anne Etien, Rabie Ben Atitallah, Philippe Marquet et Jean-Luc Dekeyser : A model driven design framework for high performance embedded systems. Rapport technique, INRIA, 2008. URL http://hal.inria.fr/inria-00311115/en/.

[54] Markus Geimer, Sameer S. Shende, Allen D. Malony et Felix Wolf : A generic and configurable source-code instrumentation component. *In Proceedings of the 9th International Conference on Computational Science*, ICCS 2009, 2009.

[55] Anna Gerber, Michael Lawley, Kerry Raymond, Jim Steel et Andrew Wood : Transformation : The missing link of mda. 2002.

[56] Flori Glitia, Anne Etien et Cedric Dumoulin : Fine Grained Traceability for an MDE Approach of Embedded System Conception. *In ECMDA Traceability Workshop*, Germany, 2008.

[57] E. Gómez-Martínez et J. Merseguer : ArgoSPE : Model-based Software Performance Engineering. volume 4024, pages 401–410. Springer-Verlag, Springer-Verlag, 2006.

[58] Mark Grechanik, Kathryn S. McKinley et Dewayne E. Perry : Recovering and using use-case-diagram-to-source-code traceability links. *In Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 2007.

[59] Wolfgang Haberl, Markus Herrmannsdoerfer, Jan Birke et Uwe Baumgarten : Model-level debugging of embedded real-time systems. *In Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology*, CIT '10, 2010.

[60] Kim Hazelwood : *Dynamic Binary Modification : Tools, Techniques, and Applications*. Morgan & Claypool Publishers, March 2011.

[61] William Heaven et Anthony Finkelstein : Uml profile to support requirements engineering with kaos. *IEE Proceedings - Software*, 2004.

[62] Ábel Hegedüs, Gabor Bergmann, Istvan Ráth et Daniel Varró : Back-annotation of simulation traces with change-driven model transformations. *Software Engineering and Formal Methods, IEEE International Conference on*, 2010.

[63] Ábel Hegedüs, István Ráth et Dániel Varró : Back-annotation framework for Simulation Traces of Discrete Event-based Languages. Rapport technique, BME, 2010. URL http://home.mit.bme.hu/hegedusa/publist.html.

[64] Mei-Chen Hsueh, Timothy K. Tsai et Ravishankar K. Iyer : Fault injection techniques and tools. *Computer*, April 1997.

[65] IEEE : *IEEE standard computer dictionary : a compilation of IEEE standard computer glossaries*. IEEE Computer Society Press, New York, NY, USA, 1991.

[66] Daniel Jackson : Alloy : A new technology for software modelling. *In Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS '02, 2002.

[67] J. Jackson : A keyphrase based traceability scheme. *In Tools and Techniques for Maintaining Traceability During Design, IEE Colloquium on*, 1991.

[68] Kurt Jensen : Coloured petri nets. *In Advances in Petri Nets'86*, pages 248–299, 1986.

[69] Y Jia et M Harman : An analysis and survey of the development of mutation testing. *Software Engineering, IEEE Transactions on*, 2010.

[70] James A. Jones, Mary Jean Harrold et John Stasko : Visualization of test information to assist fault localization. *In Proceedings of the 24th International Conference on Software Engineering*, 2002.

[71] Frédéric Jouault : Loosely Coupled Traceability for ATL. *In ECMDA Traceability Workshop*, 2005.

[72] Frédéric Jouault et Ivan Kurtev : Transforming models with atl. *In Proceedings of the Model Transformations in Practice Workshop at MoDELS 2005*, Montego Bay, Jamaica, 2005.

[73] Hermann Kaindl : The missing link in requirements engineering. *SIGSOFT Softw. Eng. Notes*, 1993.

[74] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean marc Loingtier et John Irwin : Aspect-oriented programming. *In ECOOP*. SpringerVerlag, 1997.

[75] D S Kolovos, R F Paige et F A C Polack : Merging models with the epsilon merging language. *International conference on model driven engineering languages and systems MODELS06*, pages 215–229, 2006.

[76] Dimitrios S. Kolovos, Richard F. Paige et Fiona A. C. Polack : On-Demand Merging of Traceability Links with Models. *In ECMDA Traceability Workshop*, 2006.

[77] Dimitrios S. Kolovos, Richard F. Paige et Fiona A.C. Polack : Model comparison : a foundation for model composition and model transformation testing. *In Proceedings of the 2006 international workshop on Global integrated model management*, 2006.

[78] I. Kurtev, M. Dee, A. Göknil et K.G. van den Berg : Traceability-based change management in operational mappings. *In ECMDA Traceability Workshop*, Israel, 2007.

[79] Ivan Kurtev : State of the art of qvt : A model transformation language standard. 2008.

[80] Jochen M. Küster et Mohamed Abd-El-Razik : Validation of model transformations : first experiences using a white box approach. *In Proceedings of the 2006 international conference on Models in software engineering*, 2006.

[81] Sébastien Le Beux : *Un flot de conception pour applications de traitement du signal systématique implémentées sur FPGA à base d'Ingénierie Dirigée par les Modèles*. ThÃ¨se de doctorat (PhD Thesis), Laboratoire d'informatique fondamentale de Lille, UniversitÃ© des sciences et technologies de Lille, France, décembre 2007.

[82] Nan Li, Upsorn Praphamontripong et Jeff Offutt : An experimental comparison of four unit test criteria : Mutation, edge-pair, all-uses and prime path coverage. *In ICSTW '09 : Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops*, 2009.

[83] Yuehua Lin, Jeff Gray et Frédéric Jouault : DSMDiff : a differentiation tool for domain-specific models. *European Journal of Information Systems*, 2007.

[84] Yuehua Lin, Jing Zhang et Jeff Gray : Model comparison : A key challenge for transformation testing and version control in model driven software development. *In Control in Model Driven Software Development. OOPSLA/GPCE : Best Practices for Model-Driven Software Development*, 2004.

[85] Yuehua Lin, Jing Zhang et Jeff Gray : A testing framework for model transformations. *In Model-Driven Software Development*. 2005.

[86] Yu-Seung Ma, Yong-Rae Kwon et Jeff Offutt : Inter-class mutation operators for java. *In ISSRE '02 : Proceedings of the 13th International Symposium on Software Reliability Engineering*, 2002.

[87] Yu-Seung Ma, Jeff Offutt et Yong Rae Kwon : Mujava : an automated class mutation system. *Softw. Test. Verif. Reliab.*, 2005.

[88] István Madari, László Angyal et László Lengyel : Traceability-based incremental model synchronization. *W. Trans. on Comp.*, 8 (10), 2009.

[89] Tom Mens et Pieter Van Gorp : A taxonomy of model transformation. *In Proc. International Workshop on Graph and Model Transformation (GraMoT 2005)*, 2006.

[90] Jean-Marie Mottu, Benoit Baudry et Yves Le Traon : Mutation analysis testing for model transformations. *In ECMDA 06*, juillet 2006.

[91] Pierre-Alain Muller, Franck Fleurey, Didier Vojtisek, Zoé Drey, Damien Pollet, Frédéric Fondement, Philippe Studer et Jean-Marc Jézéquel : On Executable Meta-Languages applied to Model Transformations. *In Model Transformations In Practice Workshop*, Montego Bay, Jamaïque, octobre 2005.

[92] T Murmane, K Reed, T Assoc et V Carlton : On the effectiveness of mutation analysis as a black box testing technique. *In Software Engineering Conference*, pages 12–20, 2001.

[93] NVIDIA : Compute Visual Profiler. http://developer.download.nvidia.com/compute/cuda/4_0_rc2/toolkitMdocs/VisualProfiler/Compute_Visual_Profiler_User_Guide.pdf, 2010.

[94] A. Wendell O. Rodrigues, Frédéric Guyomarc'H et Jean-Luc Dekeyser : An MDE Approach for Automatic Code Generation from MARTE to OpenCL. Rapport technique, 2011. http ://hal.inria.fr/inria-00563411/PDF/RR-7525.pdf/.

[95] A. Wendell O. Rodrigues, Frédéric Guyomarc'H et Jean-Luc Dekeyser : Programming Massively Parallel Architectures using MARTE : a Case Study. *In 2nd Workshop on Model Based Engineering for Embedded Systems Design (M-BED 2011) on Date Conference 2011*, mars 2011.

[96] Obeo : Qvtr, 2007. http ://wiki.eclipse.org/M2M/QVTR.

[97] Obeo : Traceabilitypro, 2007. http ://obeo.fr/pages/obeo-traceability/.

[98] Object Management Group, Inc. : MOF Query / Views / Transformations. http ://www.omg.org/spec/QVT/1.0/, avril 2008. OMG paper.

[99] Object Management Group, Inc. : UML Profile for MARTE, Beta 2. http ://www.omgmarte.org/Documents/Specifications/08-06-09.pdf, juin 2008.

[100] Object Management Group, Inc. : Object Constraint Language. http ://www.omg.org/spec/OCL/2.2/, février 2010. OMG paper.

[101] Object Management Group, Inc. : UML 2.3 Superstructure and Infrastructure. http ://www.omg.org/spec/UML/2.3/, mai 2010. OMG paper.

[102] Object Management Group, Inc. : Modeling and Analysis of Real-Time and Embedded Systems. http ://www.omg.org/spec/MARTE/1.1/, janvier 2011. OMG paper.

[103] Object Managment Group, Inc. : UML Profile for Schedulability, Performance, and Time, version 1.1. http://www.omg.org/spec/SPTP, 2005. URL http://www.omg.org/spec/SPTP.

[104] A J Offutt et J Pan : Detecting equivalent mutants and the feasible path problem. *Software Testing, Verification and Reliability*, 1997.

[105] A. Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H. Untch et Christian Zapf : An experimental determination of sufficient mutant operators. *ACM Trans. Softw. Eng. Methodol.*, 1996.

[106] Jon Oldevik et Tor Neple : Traceability in model to text transformations. *In ECMDA Traceability Workshop*, 2006.

[107] Gøran K. Olsen et Jon Oldevik : Scenarios of traceability in model to text transformations. *In Proceedings of the 3rd European conference on Model driven architecture-foundations and applications*, ECMDA-FA'07, 2007.

[108] Éric Piel : *Ordonnancement de systèmes parallèles temps-réel, De la modélisation à la mise en œuvre par l'ingénierie dirigée par les modèles*. ThÃ¨se de doctorat (PhD Thesis), Laboratoire d'informatique fondamentale de Lille, UniversitÃ© des sciences et technologies de Lille, France, décembre 2007.

[109] Imran Rafiq Quadri : *MARTE based model driven design methodology for targeting dynamically reconfigurable FPGA based SoCs*. Thèse de doctorat, Laboratoire d'informatique fondamentale de Lille, UniversitÃ© des sciences et technologies de Lille, Lille, France, April 2010.

[110] Sagar Sen, Benoit Baudry et Jean-Marie Mottu : On Combining Multi-formalism Knowledge to Select Models for Model Transformation Testing. *In ICST.*, Norway, avril 2008.

[111] Sagar SEN, Benoit BAUDRY et Doina PRECUP : Partial model completion in model driven engineering using constraint logic programming. *In International Conference on the Applications of Declarative Programming*, 2007.

[112] S. SENDALL et W. KOZACZYNSKI : Model transformation : the heart and soul of model-driven software development. *Software, IEEE*, 2003.

[113] Seyyed M. A. SHAH, Kyriakos ANASTASAKIS et Behzad BORDBAR : From uml to alloy and back again. *In Proceedings of the 6th International Workshop on Model-Driven Engineering, Verification and Validation*, MoDeVVa '09, 2009.

[114] Adenilso SIMAO, José Carlos MALDONADO et Roberto da SILVA BIGONHA : A transformational language for mutant description. *Comput. Lang. Syst. Struct.*, 2009.

[115] Ben H. SMITH et Laurie WILLIAMS : On guiding the augmentation of an automated test suite via mutation analysis. *Empirical Software Enggeering*, June 2009.

[116] Ben H. SMITH et Laurie WILLIAMS : Should software testers use mutation analysis to augment a test set ? *Journal of Systems Software*, November 2009.

[117] Richard SOLEY : Model driven architecture. *Object Management Group*, 2000.

[118] Quan SUN et Hui TIAN : A flexible automatic source-level instrumentation framework for dynamic program analysis. *In Software Engineering and Service Science (ICSESS), 2011 IEEE 2nd International Conference on*, july 2011.

[119] Julien TAILLARD : *Une approche orientée modèle pour la parallélisation d'un code de calcul éléments finis*. Thèse de doctorat, Université des Sciences et Technologies de Lille, feb 2009. in french.

[120] Jie TAO et Josef WEIDENDORFER : Cache simulation based on runtime instrumentation for openmp applications. *In Proceedings of the 37th annual symposium on Simulation*, ANSS '04, 2004.

[121] Massimo TISI, Frédéric JOUAULT, Piero FRATERNALI, Stefano CERI et Jean BÉZIVIN : On the use of higher-order model transformations. *In Model Driven Architecture - Foundations and Applications*. Springer Berlin / Heidelberg, 2009.

[122] I. TRAORE, I. WOUNGANG, A.A. EL SAYED AHMED et M.S. OBAIDAT : UML-based Performance Modeling of Distributed Software Systems. *In Performance Evaluation of Computer and Telecommunication Systems (SPECTS)*, july 2010.

[123] Stefan VAN BAELEN et Bert VANHOOFF : Traceability management toolset. MARTES Project Report D.2.3, ITEA-MARTES consortium, septembre 2007. URL https://lirias.kuleuven.be/handle/123456789/156270.

[124] K.G. Berg van DEN, B. TEKINERDOGAN et H. NGUYEN : Analysis of crosscutting in model transformations. *In ECMDA Traceability Workshop*, SINTEF Report, July 2006.

[125] Bert VANHOOFF, Dhouha AYED, Stefan Van BAELEN, Wouter JOOSEN et Yolande BERBERS : Uniti : A unified transformation infrastructure. *In MoDELS*, 2007.

[126] Bert VANHOOFF, Dhouha AYED et Yolande BERBERS : A framework for transformation chain development processes. *In Proceedings of the ECMDA Composition of Model Transformations Workshop*, 2006.

[127] Bert VANHOOFF, Stefan Van BAELEN, Aram HOVSEPYAN, Wouter JOOSEN et Yolande BERBERS : Towards a transformation chain modeling language. *Embedded Computer Systems : Architectures, Modeling, and Simulation*, 2006.

[128] Bert VANHOOFF, Stefan Van BAELEN, Wouter JOOSEN et Yolande BERBERS : Traceability as Input for Model Transformations. *ECMDA*, 2007.

[129] Bert VANHOOFF et Yolande BERBERS : Breaking up the transformation chain. *20th Annual ACM SIGPLAN Conference on Object*, 2005.

[130] Bert VANHOOFF et Yolande BERBERS : Supporting modular transformation units with precise transformation traceability metadata. *In European Conference on Model Driven Architecture,*, 2005.

[131] Jeffrey M. VOAS et Keith W. MILLER : The revealing power of a test case. *Software Testing, Verification and Reliability*, 1992.

[132] M. VOELTER et B. KOLB : Best Practices for Model-to-Text Transformations. 2006.

[133] S. WANTHAKIS, Régnier P. et K. KARAPOULIOS : *Le test des logiciels*. 2000.

[134] Roel WIERINGA : An introduction to requirements traceability, 1995.

[135] Manuel WIMMER, Gerti KAPPEL, Johannes SCHOENBOECK, Angelika KUSEL, Werner RETSCHITZEGGER et Wieland SCHWINGER : A petri net based debugging environment for qvt relations. *In Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, 2009.

[136] Manuel WIMMER, Angelika KUSEL, Johannes SCHÖNBÖCK, Gerti KAPPEL, Werner RETSCHITZEGGER et Wieland SCHWINGER : Reviving qvt relations : Model-based debugging using colored petri nets. *In MoDELS*, USA, 2009.

[137] Stefan WINKLER et Jens PILGRIM : A survey of Traceability in Requirements Engineering and Model-Driven Development. *Software System Modelling*, 2010.

[138] Zhenchang XING et Eleni STROULIA : Umldiff : an algorithm for object-oriented design differencing. *In Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, 2005.

[139] Andrés YIE, Rubby CASALLAS, Dirk DERIDDER et Dennis WAGELAAR : A Practical Approach to Multi-Modeling Views Composition. *ECEASST*, 2009.

[140] Andrés Yie et Dennis Wagelaar : Advanced Traceability for ATL. *In International Workshop on Model Transformation with ATL (MtATL 2009)*, France, 2009.

[141] Huafeng Yu : *A MARTE-Based Reactive Model for Data-Parallel Intensive Processing : Transformation Toward the Synchronous Model*. Thèse de doctorat, Lille, France, November 2008.