Compilation optimisée des modèles UML

Asma Charfi Smaoui

UNIVERSITÉ PARIS-SUD
ÉCOLE DOCTORALE D'INFORMATIQUE

THÈSE DE DOCTORAT

présentée en vue d'obtenir le grade de Docteur en Informatique

par

Asma CHARFI SMAOUI

COMPILATION OPTIMISÉE DES MODÈLES UML

Soutenue le 12 décembre 2011

devant la commission d'examen composée de :

Guy VIDAL-NAQUET,	Président,	UPS
Pierre BOULET,	Directeur,	USTL-LIFL
Chokri MRAIDHA,	Encadrant,	CEA-LIST
Pierre-Alain MULLER,	Rapporteur,	UHA
Xavier BLANC,	Rapporteur,	LabRI
Albert COHEN,	Examinateur,	INRIA

Remerciements

Je tiens tout d'abord à remercier Chokri Mraidha qui a encadré cette thèse. Chokri m'a donné l'opportunité d'effectuer un travail de recherche et a su créer un contexte idéal pour ce travail. En plus de son soutien, de ses conseils et du savoir qu'il m'a transmis, je tiens vivement à le remercier pour sa disponibilité et son encouragement.

Je suis également extrêmement reconnaissante envers mon directeur de thèse, Pierre Boulet, pour ses conseils avisés, ses remarques pertinentes et l'exemplarité de sa rigueur scientifique.

J'adresse mes remerciements à Pierre-Alain Muller et à Xavier Blanc qui m'ont fait l'honneur d'évaluer mon travail de thèse et d'en être les rapporteurs. Je les remercie pour leurs remarques constructives. Je remercie également Guy VIDAL-NAQUET d'avoir accepté de faire partie des membres du jury.

Je remercie également Albert Cohen pour avoir participé au jury en apportant ses compétences assez unique en France dans la chaine de compilation GCC.

Le travail présenté dans ce mémoire a été réalisé dans le centre CEA Saclay au sein du laboratoire d'ingénierie dirigée par les modèles pour les systèmes embarqués dirigé par François Terrier. J'adresse donc mes remerciements à François pour m'avoir accueilli dans son laboratoire.

Un grand remerciement est adressé à tous mes collègues, membres de l'équipe DILS pour l'ambiance très favorable qu'ils ont su créer autour de moi. En particulier : Takoua, Fadoi, Rania, Lamine et Raouf.

Un remerciement particulier à tous mes ami(e)s, qui se reconnaissent, qui m'ont soutenu dans les moments difficiles en particulier mes deux chères binômes Mariem et Hanen et à la toute chère Raja qui m'a beaucoup aidée dans la mise en page de ce manuscrit.

La totalité de ma reconnaissance et de mes pensées vont à mes parents Abdelwaheb et Emna et à mes 3 sœurs Ichrak et sa petite famille, Mariem et Mouna. Ils n'ont jamais cessé de m'encourager et de m'apporter du support surtout durant les dernières phases, pas toujours faciles, de la rédaction de ce manuscrit. C'est donc tout naturellement que ce document leur soit dédié.

Et bien sûr, je ne peux terminer sans exprimer ma profonde gratitude pour Omar, mon mari, qui a su me supporter, me remonter le moral, avec une patience infinie, pendant ces années pas toujours très faciles. C'est son soutien quotidien, ses sacrifices, sa patience et son encouragement qui m'ont aidée à surmonter toutes les difficultés rencontrées au cours de cette thèse.

Clamart, le 09 décembre 2011

Table des matières

CHAPITRE 1

Introduction

1.1 Contexte

Les systèmes embarqués (enfouis dans d'autres systèmes) sont désormais utilisés dans plusieurs domaines d'applications tels que le transport (l'automobile, l'avionique, l'espace) et les appareils électroniques (téléphones portables, appareils électroménager, etc.). Devant assurer de plus en plus de fonctionnalités, ceux-ci sont devenus de plus en plus complexes. En addition à cette complexité, vient s'ajouter une contrainte liée à la dynamique toute particulière du marché des systèmes embarqués. Cette contrainte oblige les industries à fournir des produits toujours plus novateurs à moindre prix tout en optimisant leurs temps de mise sur le marché (*time to market*).

Pour gérer cette complexité croissante et augmenter la productivité des développeurs, l'ingénierie dirigée par les modèles (IDM) [1] offre un cadre méthodologique outillé aux développeurs des systèmes embarqués, qui se concentrent désormais sur l'élaboration de modèles abstraits, plutôt que sur des concepts liés à l'algorithmique et la programmation. Généralement, une approche de développement de systèmes embarqués basée sur l'IDM comporte 3 étapes : la modélisation, la génération de code et la compilation du code généré pour obtenir le code final de l'application (l'exécutable). Le développeur n'est censé intervenir qu'à la première étape : la modélisation du système (la modélisation de la structure, des fonctionnalités, de l'interaction avec l'environnement, etc.). La génération de code et la compilation du code généré sont des étapes automatiques, ce qui est censé améliorer considérablement la qualité du code produit. En effet, la génération automatique du code à partir des modèles permet d'éviter les erreurs de programmation assez fréquentes dans l'ancienne approche orientée code où les développeurs écrivaient le code à la main. Au-delà du gain de temps de production du code, la génération automatique de celui-ci contribue également à la diminution du temps de correction des programmes écrits à la main tout en permettant la génération de code lisible, maintenable et consistant avec les modèles.

Des efforts ont été consacrés dans l'approche classique d'IDM pour garantir la lisibilité et la maintenabilité du code généré, mais ceci au dépend de la performance du code. En effet, la performance de code joue un rôle très important dans le développement des systèmes embarqués à cause des contraintes imposées sur ces systèmes. A titre d'exemple, la production d'un code le plus compact possible est exigée vu que la capacité de stockage des systèmes embarqués est plutôt limitée (absence de disque dur et l'utilisation de mémoires de taille limitée pour des raisons économiques). De plus, en produisant un code compact pour chaque fonctionnalité du système, ce dernier se trouve en mesure d'intégrer plus de fonctionnalités répondant ainsi aux besoins des utilisateurs (en nombre de fonctionnalités exigées) qui ne cessent d'augmenter.

Pour produire un code optimisé, l'approche classique d'IDM se base essentiellement sur les techniques d'optimisation des compilateurs de code. C'est donc la 3ème étape (l'étape de la compilation) de l'approche IDM qui traite la problématique de l'optimisation du code exécutable. La science de compilation de code est une science en perpétuelle évolution depuis plus de 50 ans. Les techniques d'optimisations de code ont évolué allant des optimisations statiques aux techniques de compilation à la volée (*JIT : Just in Time Compilation*) [2] en passant par les techniques de profilage (*profiling*) [3]. Les compilateurs de code optimisants ont été conçus pour prendre en entrée un code écrit dans un langage de 3ème génération (C/C++/Java/..) et produire en sortie un exécutable optimisé. Dans l'approche

classique d'IDM et pour profiter des optimisations puissantes et variées des compilateurs de code, l'étape de la génération de code produit un code de 3ème génération à partir des modèles spécifiés généralement en utilisant un langage de modélisation de haut niveau tel que le langage UML (*Unified Modeling Langauge*) [4].

Dans cette thèse nous nous intéressons aux approches dirigées par les modèles pour le développement des systèmes embarqués qui utilisent le langage de modélisation UML, génèrent du code C++ à partir de ses modèles et compilent le code C++ en utilisant le compilateur GCC (*Gnu Compiler Collection*) [1]. Nous nous intéressons également à un seul critère d'optimisation de code : la réduction de la taille du code embarqué dans la mémoire du système à développer.

1.2 Problématique

En se basant sur les optimisations puissantes des compilateurs de code, l'approche classique dirigée par les modèles pour le développement des systèmes embarqués semble résoudre le problème de la production de code optimisé à partir des modèles UML. Cependant, prenant toutes les informations liées au système à partir du code généré (et non pas du modèle de départ), les compilateurs de code les plus puissants sont incapables de produire un exécutable optimisé capable de satisfaire les exigences de taille et de performance du système. En effet, les langages de programmation ne sont pas dans le même niveau d'abstraction que les langages de modélisation et n'offrent pas les mêmes concepts. Par conséquent, il est très difficile voir impossible de traduire directement tous les concepts du langage de modélisation vers des concepts du langage de programmation. La génération de code s'accompagne souvent par une perte d'information liée à la sémantique du langage de modélisation. Ces informations perdues suite à la génération de code ne sont plus visibles par le compilateur de code qui se trouve dans l'incapacité de les exploiter à des fins d'optimisation. D'autres informations liées au flot de contrôle de l'application sont perdue lors de la génération de code. En effet, le code généré à partir des modèles possède une forme séquentielle qui n'exprime pas bien le flot de contrôle de l'application à l'opposition des diagrammes comportementaux d'UML qui expriment aussi bien le flot de contrôle que le flot de données. Ces multiples pertes qu'engendre la génération de code rendent inefficaces les optimisations faites pas le compilateur.

Non satisfaits par les performances du code exécutable produit, les développeurs des systèmes embarqués modifient souvent le code C++ généré en changeant, supprimant ou ajoutant des instructions qui, une fois compilées résultent à un code exécutable plus optimisé. Cette modification directe de code crée un *gap* entre le modèle et le code empêchant par la suite la rétro-ingénierie, le débogage de modèles et cassent le lien de traçabilité établi entre le modèle et le code. Souvent, elle invalide les validations et simulations faites au niveau modèle, puisque l'exécutable va être produit à partir du code modifié, qui n'est plus conforme au modèle analysé et validé. Cette modification manuelle de code généré, remet en cause les apports de l'IDM en termes de montée dans le niveau d'abstraction et d'augmentation de productivité. En effet, pour manipuler le code généré, le développeur est amené à bien comprendre le code ce qui limite l'utilité de la génération automatique de

1. http ://gcc.gnu.org

code. Ainsi, cette intervention manuelle sur le code généré est à proscrire dans une approche dirigée par les modèles. En effet, elle ne respecte ni les principes de l'IDM (les développeurs interviennent uniquement dans l'étape de la modélisation) ni les contraintes de temps et de coût imposées sur les systèmes embarqués (le changement dans le code généré peut introduire des erreurs et engendrer des pertes de temps et d'argent).

En conclusion, l'approche classique dirigée par les modèles se base sur la génération de code à partir des modèles UML pour pouvoir profiter des optimisations des compilateurs de code et ainsi produire un code optimisé pour les systèmes embarqués à ressources limités. Cependant, cette génération de code est une arme à double tranchants : elle est la cause de la perte des informations liées à la sémantique des langages de modélisation qui sont utiles pour les optimisations. Maitrisé par les développeurs, ce code généré est souvent modifié pour des fins d'optimisation ce qui est en contradiction avec le premier avantage de l'IDM : libérer les développeurs des problèmes de l'implémentation.

1.3 Contribution

L'objectif de cette thèse est d'essayer de trouver une solution au problème de la production d'un code optimisé à partir des modèles UML que l'approche classique d'IDM pour le développement des systèmes embarqués a tenté de résoudre en se basant sur la génération de code. Dans cette thèse, nous montrerons que la génération de code engendre plusieurs problèmes tels que la perte des informations liées à la sémantique du langage de modélisation utiles pour les optimisations et les conséquences de cette perte (le compilateur de code est incapable de produire un code optimisé, les développeurs modifient à la main le code généré, etc.).

Nous proposons ainsi une nouvelle mise en œuvre de l'approche dirigée par les modèles pour le développement des systèmes embarqués qui élimine l'étape de la génération de code (et donc les problèmes qui en découlent). La solution que nous proposons est un compilateur de modèles UML qui génère directement de l'assembleur sans passer par un langage de programmation de 3ème génération. Le compilateur de modèle proposé dans cette thèse prend en entrée le modèle UML en éliminant ainsi le premier problème de pertes d'informations sémantiques utiles pour les optimisations. La compilation directe de modèle est une transformation dont le résultat est un binaire conforme au modèle (les propriétés du modèle sont conservées dans le binaire généré). Elle permet ainsi, de préserver les résultats des validations et analyses faites au niveau modèle.

Notre compilateur de modèles profite de toutes les optimisations du compilateur de code produisant ainsi un code exécutable aussi performant que le code exécutable produit par l'approche existante de génération de code. En prenant en compte les informations liées à la sémantique du langage UML pour des fins d'optimisation, notre compilateur de modèle produit dans certains cas un code exécutable plus compact.

1.4 Motivation

Notre approche propose un compilateur de modèles UML qui ne génère pas de code de 3ème génération, une approche qui a été longtemps défendue par la communauté de l'UML

[5], [6] et [7]. En effet, Ivar Jacobson [5] l'un des fondateurs d'UML affirme que l'utilisation de deux langages de haut niveau (le langage UML et le langage de programmation) est inutile et que la compilation directe des modèles UML permet d'éviter ce problème. Steve Mellor [6], l'auteur du premier livre sur le langage UML exécutable, met en cause, dans son livre publié il y a presque 10 ans, le besoin d'un langage de programmation si une génération automatique de 90% du code est possible. Bran Selic [7], affirme à son tour que pour suivre l'évolution de la complexité des systèmes et produire un code répondant à leurs contraintes, se baser sur les langages de programmation traditionnels et leurs techniques de compilations usuelles n'est pas la bonne démarche à suivre. L'utilisation des langages de plus haut niveau et de nouvelles techniques de synthèse et de compilation représente la solution à ce problème de complexité.

La communauté de compilation de code notamment celle de développement de la collection de compilateurs GCC[2] affirme que compiler directement un langage source de plus haut niveau (par exemple le C++) plutôt que générer du code de bas niveau (par exemple le C) et utiliser les compilateurs existants du langage bas niveau (compilateur C) est une approche prometteuse qui améliore la qualité du code exécutable en améliorant certains types d'optimisations. Ainsi, produire un code exécutable plus performant à partir d'un nouveau langage source passe selon la communauté GCC par la construction d'un compilateur propre à ce nouveau langage source.

Suivant la chronologie de l'évolution de la construction des compilateurs des langages de programmation (Figure 1.1), la construction d'un compilateur de modèles est la prochaine étape à effectuer.

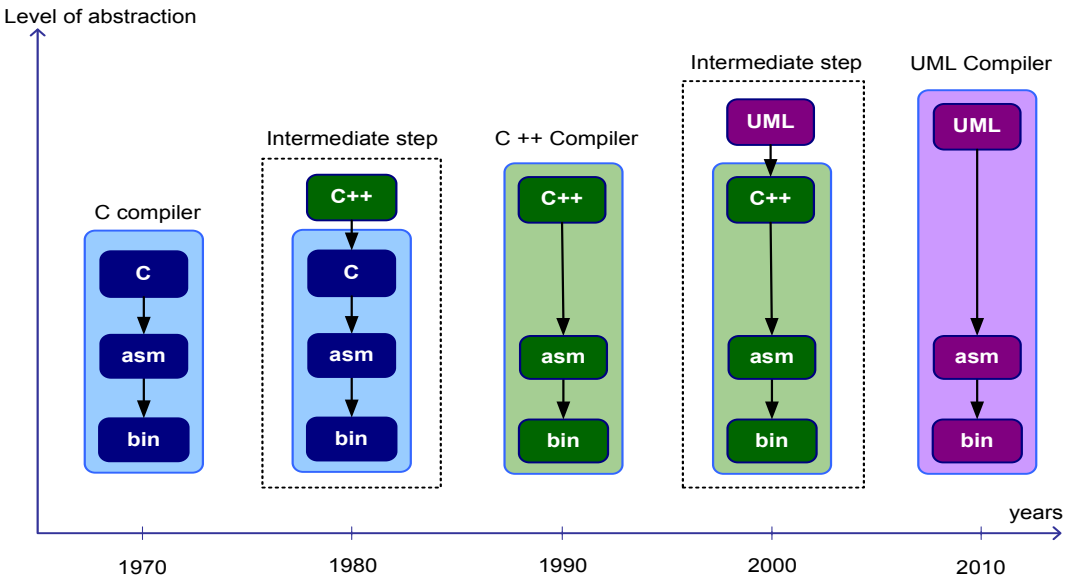


FIGURE 1.1 – Evolution chronologique de la construction des compilateurs C/C++ et UML. Pour le langage UML on est encore dans l'étape intermédiaire de génération de code.

Cette figure montre qu'à chaque fois qu'un langage de programmation de plus haut niveau apparait, une étape intermédiaire apparait : l'étape de la génération de code vers le langage existant en attendant la construction d'un nouveau compilateur pour le nouveau langage.

2. http ://gcc.gnu.org/onlinedocs/gccint/Languages.html# Languages

Ainsi, quand le langage C++ est apparu, et avant la construction d'un compilateur pour ce langage, compiler les programmes C++ revenait à générer tout d'abord du C et à utiliser par la suite le compilateur C déjà existant. Une fois le compilateur C++ prêt, l'étape de la génération du C est devenue inutile et moins efficace. Pour le langage UML, nous nous trouvons encore dans l'étape intermédiaire, à savoir la génération de code C++. Produire un compilateur de modèles UML n'a pas été réalisée jusqu'à maintenant à cause des barrières techniques qui existaient à l'époque telles que l'absence d'une définition claire de la sémantique d'exécution des modèles UML, la dépendance d'UML vis-à-vis des langages de programmation pour produire un code exécutable, etc.

Actuellement, le langage UML a beaucoup évolué et les barrières techniques évoquées ci-dessus ne sont plus d'actualité. Le langage UML peut être considéré comme un langage de programmation vu que l'étape de génération de code C++ à partir des modèles UML est complètement automatique. Nous profitons de cette évolution pour proposer le premier prototype de compilateur de modèles UML qui permet de quitter l'étape intermédiaire en compilant directement les modèles UML (Figure 1.1).

1.5 Organisation

Cette thèse est organisée selon le plan suivant :

Le deuxième chapitre présente l'approche classique dirigée par les modèles pour le développement des systèmes embarqués et les techniques d'optimisation adoptées par cette approche pour produire un code optimisé à partir des modèles UML exécutables.

Le troisième chapitre présente la première contribution de cette thèse qui consiste à proposer le premier compilateur de modèles UML exécutables. Nous présentons ainsi le langage source de ce compilateur, ses formes intermédiaires et un exemple de son utilisation.

La deuxième contribution de cette thèse consiste à améliorer le pouvoir d'optimisation du compilateur de modèles proposé et ceci en prenant en compte les informations liées à la sémantique du langage de modélisation UML en particulier pour les machines à états UML. Ainsi, le quatrième chapitre présente le premier niveau d'optimisation du compilateur de modèles qui améliore l'optimisation de l'élimination de code mort du compilateur GCC.

Le cinquième chapitre présente le deuxième niveau d'optimisation du compilateur de modèles proposé en améliorant une autre optimisation du compilateur GCC : l'optimisation de la fusion des blocs de base (blocs d'instructions) pour éliminer les expressions redondantes.

Le sixième chapitre présente l'évaluation de la taille du code assembleur produit par le compilateur de modèles proposé par rapport à la taille du code assembleur produit en utilisant l'approche existante de génération de code. Les trois générateurs de code les plus utilisés du commerce pour le développement des systèmes embarqués ont étés évaluées.

Enfin, nous concluons cette thèse par une présentation des apports, des limitations et des perspectives de ce travail de recherche.

CHAPITRE 2

État de l'art

L'objectif de ce chapitre est de positionner les contributions de cette thèse face aux travaux existants. La première contribution consiste à proposer le *premier* compilateur de modèles UML produisant directement de l'assembleur à partir des modèles. Ce compilateur représente la mise en œuvre d'une nouvelle approche : *la compilation directe de modèles UML* pour la production de binaire exécutable à partir de modèles de conception des systèmes embarqués. Ainsi, la première section de ce chapitre étudie l'approche existante qui consiste à générer du code dans des langages de 3ème génération (C/C++/Java, etc.) à partir des modèles UML et à compiler ce code généré pour la production du binaire.

La deuxième contribution de cette thèse consiste à améliorer le pouvoir d'optimisation du compilateur de modèles proposé afin de produire un code assembleur plus compact que le code assembleur produit par les compilateurs de code. Ainsi, la deuxième section de ce chapitre présente les différents niveaux d'optimisation dans l'approche classique dirigée par les modèles pour le développement des systèmes embarqués.

2.1 Approche classique dirigée par les modèles

Généralement, une approche dirigée par les modèles pour le développement des systèmes embarqués comporte 3 étapes : la modélisation, la génération de code à partir du modèle et la compilation du code généré (Figure 2.1).

FIGURE 2.1 – Les 3 étapes d'une approche basée sur l'IDM pour le développement des systèmes embarquées : la modélisation, la génération de code et la compilation du code généré

Chacune de ces 3 étapes peut comporter à son tour d'autres étapes selon le besoin et le type du système embarqué considéré. En effet, pour les systèmes embarqués ayant des contraintes de sureté de fonctionnement, la première étape (la modélisation) s'accompagne généralement d'une étape de validation de comportement du système [8], [9] et [10]. Si le concepteur souhaite modéliser à la fois la partie matérielle et la partie logicielle de son système, la modélisation se décompose dans ce cas en sous étapes : la modélisation de l'application, la modélisation de la plateforme et l'association entre les fonctionnalités de l'application et les éléments de la plateforme [11], [12]. De la même manière, l'étape de la génération de code peut comporter plusieurs transformations de modèles jusqu'à atteindre le code cible choisi (généralement un langage de programmation tel que C/C++/Java,..). Si le concepteur souhaite générer du code relatif à sa partie matérielle, les langages de description de matériel peuvent être ciblés comme VHDL et SystemC. Finalement, l'étape de la compilation de code se décompose à son tour de plusieurs phases de génération de code et d'optimisation [13], afin de produire un code binaire optimisé pour les systèmes embarqués à ressources limités.

Dans le cadre de ce travail de recherche, nous limitons notre étude des 3 étapes de développement des systèmes embarqués comme suit : pour la modélisation, nous nous intéressons uniquement à la modélisation des fonctionnalités du système embarqué (la partie logicielle). Pour la génération de code, nous nous intéressons à la production du code de 3èmegénération à partir du modèle de l'application (les langages cibles sont de types C/C++/Java, etc.). Finalement, pour la compilation, nous étudions les différentes formes

intermédiaires ainsi que les différentes optimisations des compilateurs de code. Dans ce qui suit, nous détaillerons chacune de ces 3 étapes.

2.1.1 Première étape : la modélisation

L'histoire du développement des systèmes est une histoire de montée dans le niveau d'abstraction. Les langages de modélisation sont plus abstraits que les langages de programmation (C/C++/Java) qui eux même sont plus abstraits que les langages machines (assembleur, binaire).

Le langage UML (*Unified Modeling Language*) est le langage de modélisation orienté objet standardisé par l'OMG (*Object Managment Group*) depuis plus que 15 ans. Au début, en définissant une syntaxe unifiée pour la modélisation des systèmes, UML intervenait uniquement dans la phase de conception. Il facilitait ainsi la conception des systèmes et la communication entre les concepteurs. L'implémentation du système, bien qu'elle doive être compatible avec les modèles UML conçus, était une étape indépendante. C'est en développant des outils de génération automatique de code à partir des modèles UML que ce langage intervient désormais dans l'étape de l'implémentation des systèmes.

La génération automatique de code a ainsi contribué à la réduction du temps d'implémentation des systèmes et à la production d'un code plus consistant avec le modèle et plus lisible [14], [15]. Les modèles à partir desquels une génération automatique de code exécutable est possible sont appelés modèles *exécutables*. La section suivante présente l'évolution qu'a subit le langage UML dans la définition de sa syntaxe ainsi que sa sémantique pour pouvoir définir des modèles exécutables.

2.1.1.1 UML et la modélisation exécutable

Selon [16], un modèle exécutable doit répondre aux deux critères suivants :

- C1 : Ce modèle doit être décrit dans un langage possédant une sémantique d'exécution rigoureusement définie.
- C2 : Ce modèle doit représenter de manière exhaustive et à un niveau de détail suffisant le comportement du système.

Plusieurs approches existent pour la modélisation du comportement d'un système. La classification de Schattkowsky [17] nous parait intéressante : il définit selon les travaux de Bock [18] 3 classes de modèles pour exprimer le comportement d'un système :

- Les modèles de flots de contrôle
- Les modèles de flots de données
- Les modèles de machines à états

Le langage UML offre plusieurs paquetages pour la spécification du comportement d'un système. Parmi ces paquetages, nous notons les machines à états et les activités UML [4]. Ces deux paquetages couvrent les 3 classes définies par Bock [18]. Les machines à états permettent de spécifier les modèles des machines à états (notions d'état et de transition entre états). Les activités permettent de spécifier les modèles de flot de données (les nœuds objets, les pins d'entrée et de sortie, les arcs de flot d'objet) et de flot de contrôle (les arcs de flot de contrôle, les nœuds de fork/join, etc.). Les machines à états UML sont utilisées pour décrire le cycle de vie du système en définissant les états que peut occuper ce système et

les protocoles de passage d'un état à un autre. Les activités sont souvent utilisées pour la spécification de la partie algorithmique des opérations UML.

Machines à états UML

Les machines à états UML sont inspirées des machines à états de Harel [19] qui ont-elles mêmes été inspirées des FSMs (*Finite State Machine*) [20] . Les FSMs représentent le premier modèle de comportement orienté état-transition. Il permet la modélisation du comportement d'un système à l'aide d'états simples et de transitions (déclenchées par des événements) entre ces états. Les notions de concurrence (états orthogonaux) et de hiérarchie (état composite) ainsi que la possibilité d'exécuter un comportement en passant d'un état à un autre (effet d'une transition) ne sont pas supportées dans les FSMs. Harel [19] a étendu ce modèle afin de pouvoir spécifier le comportement des systèmes complexes et concurrents. Son modèle est connu sous le nom de *StateCharts*. Les *StateCharts* de Harel définies dans les années 80 ont été améliorées et prises en compte dans l'outil STATEMATE [21] qui supporte d'autres notions utiles pour la modélisation des systèmes telles que la possibilité d'exécuter des actions en entrant et en sortant de chaque état (*entry*, *exit*), la définition de transition composée et la priorité entre les transitons.

Les machines à états UML ont été introduites dans la norme UML depuis sa création et ont subit plusieurs évolutions. La version actuelle des machines à états UML mettent les *StateCharts* de Harel dans un contexte orientée objet. Elles ajoutent aussi la possibilité d'exécuter un comportement tant que le système est dans un état donné (*doActivities*). Cependant, les machines à états UML gèrent les priorités entre les transitions d'une manière différente que les *StateCharts* de STATEMATE. Pour STATEMATE, les transitions des états parents sont plus prioritaires que les transitions des états fils. UML définit des priorités inverses (les transitions des états fils sont plus prioritaires). La figure 2.2 présente un exemple d'une machine à états UML illustrant les concepts présentés ci-dessus.

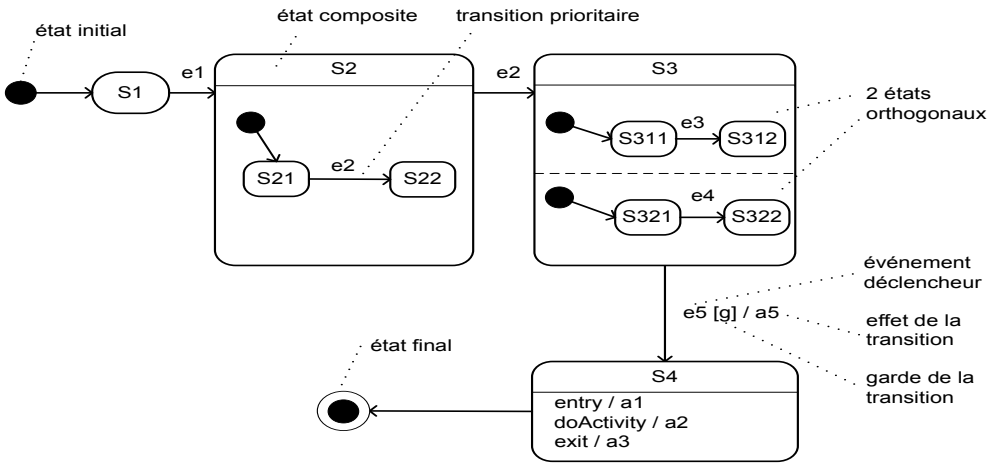


FIGURE 2.2 – Exemple d'une machine à états UML

La version actuelle des machines à états UML intègre plusieurs concepts utiles pour la spécification du comportement (aspect contrôle) des systèmes complexes. Cependant, pour la modélisation des algorithmes complexes et les comportements à flots de données ce modèle n'est pas suffisant [22]. C'est le paquetage *Activités* d'UML qui comble les lacunes

des machines à états en termes de modélisation des algorithmes.

Activités et Actions UML

Dans les modèles de flot de données, l'expression du comportement d'un système est basée sur les calculs modifiant les données [23]. Généralement, ces modèles sont représentés par des graphes dont les arcs expriment le passage de données (le flot de données) et les nœuds présentent les calculs sur ces données. Parmi les modèles de flots de données, nous pouvons citer les réseaux de Petri (*Petri Nets*) [24] souvent utilisés pour exprimer le comportement des systèmes concurrents, les DFG (*Data Flow Graph*) [25] et les SDF (*Synchronous Data Flow*) [26]. Pour le langage UML, c'est le paquetage Activités qui permet l'expression des flots de données et plus généralement les parties algorithmiques des opérations UML. En effet, un diagramme d'activité UML est formé d'un ensemble de nœuds (*Activity Nodes*) reliés par des arcs (*Activity Edges*) de deux types : les arcs de flot de contrôle (*Control Flow Edges*) pour exprimer l'enchaînent des actions (unité de base de spécification du comportement dans UML) et les arcs de flot d'objet (*Object Flow Edges*) pour laisser naviguer les données d'un nœud à un autre. Les données peuvent être explicitement représentées dans une activité à l'aide des nœuds d'objet (*Object Node*) et des *Pin* (*Input* et *Output Pin*). L'action d'un nœud de l'activité n'est exécutée que si elle reçoit toutes les données présentées par ses *Input Pin*. L'exemple de la figure suivante présente un diagramme d'activité UML illustrant les concepts présentés ci-dessus.

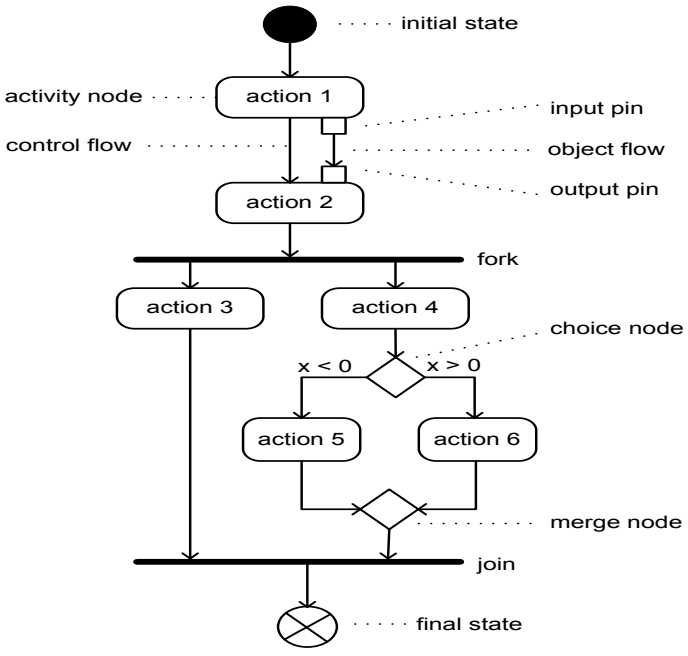


FIGURE 2.3 – Exemple d'un diagramme d'activité UML

Mise à part le flot de données, les activités UML permettent d'exprimer le flot de contrôle et ceci grâce à plusieurs nœuds tels que les *fork* et les *join* pour exprimer le parallélisme entre les actions (Figure 2.3) et les différents nœuds de contrôle (*Choice Node*, *Merge Node*, *Loop Node*, etc).

Les activités et les machines à états sont les paquetages les plus utilisés du langage UML pour la modélisation du comportement d'un système. Cependant, tel qu'ils ont été définis dans la dernière version du langage UML, ces paquetages ne permettent pas de spécifier un modèle exécutable qui satisfait les deux critères C1 et C2 définis au début de cette section. Pour satisfaire le premier critère (en rapport avec la sémantique d'exécution du modèle), l'OMG a standardisé la norme fUML (*semantics of a Foundational subset for executable UML models*) [27]. Le deuxième critère (en rapport avec la capacité de la modélisation suffisamment détaillé du comportement du système) est respecté suite à la définition du standard ALF (*Action Languge for Foundational UML*) [28].

Standard fUML et respect du premier critère

La sémantique d'exécution des éléments syntaxiques du langage UML est exprimée en langage naturel. Cela est source d'ambigüité et peut mener à plusieurs interprétations. Avec l'adoption du standard fUML, l'OMG a défini une sémantique claire et formelle pour un sous ensemble d'UML. Ce sous ensemble considère uniquement deux paquetages UML : un paquetage structurel (*les Classes*) et un paquetage comportemental (*les Activités*). La sémantique d'exécution des Activités a été inspirée de celle des réseaux de Petri (sémantique des flux de jetons (*token*) qui traversent les nœuds du début à la fin). Le sous-ensemble de fUML a été défini de façon à pouvoir faire une projection directe des éléments syntaxiques de fUML vers les éléments syntaxiques des langages de programmation dont la sémantique d'exécution est bien définie. Ceci permet la définition d'une sémantique opérationnelle du langage fUML. Selon la spécification OMG de cette norme [27], fUML représente le sous-ensemble minimal d'UML avec lequel nous pouvons exprimer la sémantique d'exécution de n'importe quel autre élément syntaxique UML. Ainsi, la sémantique d'exécution des machines à état UML peut être exprimée en fUML.

La norme fUML a contribué à la définition d'une sémantique d'exécution claire et formelle des modèles UML et en particulier aux Classes et aux Activités. Ainsi, les modèles fUML satisfont le premier critère (C1) d'un modèle exécutable. Cependant, en tant que standard et pour répondre aux besoins de divers domaines, fUML a conservé des *points de variation sémantique* [29] du langage UML. A titre d'exemple, un des points de variation sémantique concerne la sémantique d'envoi de signal (*SendSignalAction*) à une classe qui déclare la réception (*Reception*) de ce même signal. En effet, la norme UML ne définit ni le protocole d'envoi du signal, ni le temps que celui-ci prend pour arriver à l'objet receveur. Ceci est laissé volontairement non précis pour que le langage UML puisse être utilisé pour la conception de plusieurs types de systèmes indépendamment du protocole utilisé pour l'envoi des signaux. Plusieurs points de variation sémantique ont été trouvés dans la norme UML [30]. Avant d'exécuter son modèle, et pour garantir une exécution déterministe, le concepteur doit fixer les points de variation sémantique qui concernent l'exécution de son modèle.

Standard ALF et respect du deuxième critère

Le deuxième critère (C2) en rapport avec la capacité d'UML à exprimer de manière exhaustive et à un niveau de détail suffisant le comportement du système n'est pas encore satisfait par les modèles UML même après la définition de la norme fUML. En effet, une expression complète des corps des opérations n'était pas possible. L'aspect

algorithmique du comportement était alors décrit dans des langages de programmation de haut niveau (C/C++, Java, etc.). Cela mélange deux langages de deux niveaux d'abstraction différents : UML et le langage de programmation choisi mettant ainsi en doute l'utilité de l'utilisation d'un langage de modélisation pour la production de code exécutable si ce langage est incapable d'exprimer la partie algorithmique des actions. Pour représenter de manière exhaustive et à un niveau de détail suffisant le comportement du système, plusieurs approches basées sur le langage UML pour la modélisation comportementale des systèmes ont défini leurs propres langages d'action doté chacun d'une syntaxe concrète. Deux langages de modélisation exécutable basés sur UML sont apparus : le langage xUML (*Executable UML*) [31] et xtUML (*Executable and Translatable UML*) [6]. xUML est le langage de modélisation exécutable se basant sur un sous ensemble d'UML contenant essentiellement les classes et les machines à états et dont la spécification de la partie algorithmique est assurée par le langage d'action ASL (*Action Specification Language*) [32]. xtUML supporte le même sous ensemble d'UML que xUML mais il est basé sur un autre langage d'action appelé OAL (*Object Action Language*) [33]. D'autres approches et outils ont défini d'autres langages d'action tels que [34], [17] et [35].

L'utilisation des langages d'action dans la modélisation exécutable rend les modèles UML dépendants des outils avec lesquels ils ont été conçus. Ainsi, les modèles exécutables ne sont plus standardisés ce qui met en question la capacité du langage UML à spécifier des modèles exécutables. Cela a poussé l'OMG à compléter la spécification de la syntaxe des actions UML en créant en 2010 le langage ALF. Ce langage représente le standard OMG sous-jacent à fUML pour la spécification de la syntaxe concrète des actions. Il est à noter que fUML a contribué à la définition d'une syntaxe concrète des actions en proposant une bibliothèque de modèle pour la modélisation des comportements primitifs sur les types primitifs tels que l'addition des entiers, la concaténation des String, etc. Cependant, l'utilisation de cette bibliothèque pour les corps d'opérations complexes (présence de boucles, de structures conditionnelles, etc.), rend le modèle complexe, difficile à éditer et ne facilite pas la compréhension du système ce qui explique l'apport du langage textuel ALF qui permet d'exprimer d'une manière compact les corps des opérations les plus compliquées.

2.1.1.2 Synthèse

La construction d'un modèle UML exécutable passe par la modélisation de la structure et du comportement du système. Pour cela, UML offre 3 paquetages (1) les classes pour modéliser la structure (2) les machines à états pour définir le cycle de vie du système et (3) les activités pour exprimer le flot de données et de contrôle, plus précisément les parties algorithmiques des opérations UML.

Le problème de la sémantique ambigüe des modèles UML, empêchant ainsi une modélisation exécutable, a été résolu par l'OMG en définissant le standard fUML qui spécifie une sémantique claire et formelle d'un sous ensemble minimal d'UML (les classes et les activités). Pour modéliser toute la partie comportementale d'un système en utilisant le langage UML, et sans avoir recours à spécifier les corps des opérations UML dans un langage de 3èmegénération, l'OMG a standardisé le langage textuel ALF. C'est uniquement à ce stade là qu'une modélisation exécutable en utilisant le langage UML est possible (les deux critères C1 et C2 de définition d'un modèle exécutable sont ainsi respectés par le langage UML).

Une fois défini, le modèle exécutable servira comme point d'entrée des générateurs de code (Figure 2.1) qui fournissent en sortie un code écrit en langage de 3èmegénération (C, C++, Java, etc) lui-même transformé en code exécutable. La section suivante présentera l'étape de la génération de code à partir des modèles exécutable UML.

2.1.2 Deuxième étape : la génération de code

Selon la classification établie dans [36], la génération de code (C, C++, Java, etc.) à partir des modèles UML est une transformation de modèle *exogène* (le modèle et le code généré ne sont pas conforme au même méta-modèle) et *verticale* (le modèle et le code généré ne sont pas dans le même niveau d'abstraction). Les langages de transformation de modèle peuvent être classifiés en deux catégories : les langages de transformation de modèle à modèle (*M2M : model to model transformation*) et les langages de transformations de modèle à texte (*M2T : model to text transformation*). Pour les deux types de langages, une norme OMG a été standardisée : la norme QVT (*Query View Transofrmation*) [37] pour les langages de transformation M2M tel que ATL [38] et la norme MOF2TEXT (*Model to Text Transformation Language*) [39] pour les langages de transformation M2T tel que Acceleo [40]. L'approche de la transformation de modèle à texte est plus intuitive et plus facile à mettre en œuvre que celle de la transformation de modèle à modèle puisqu'elle n'exige pas la construction d'un méta-modèle pour le langage cible. Cependant, cette approche n'assure pas la validité de la transformation et ne favorise pas la traçabilité et la rétro-ingénierie.

2.1.2.1 Langages et outils de génération de code

Etant donné que la génération de code à partir des modèles exécutables cible des langages de programmation de forme textuelle, elle est généralement basée sur l'approche M2T. Outre Acceleo, plusieurs langages de génération de code basés sur l'approche M2T existent tels que XPand [41] MOFScript [42] ou encore JET [43] basé sur le langage Java.

Avec l'évolution du standard UML (section 2.1.1), les modeleurs UML ont aussi évolués et ils assurent désormais -outre l'édition des modèles UML- la génération automatique du code à partir des modèles édités. A leurs débuts, les éditeurs UML ne permettaient de générer que le code structurel (produit à partir des classes UML). La partie comportementale étant complétée manuellement par le développeur. En suivant l'évolution du langage UML, notamment la définition de la sémantique des actions et la création des langages d'action dédiés, les générateurs de code ont eux aussi évolué et sont maintenant capables de générer automatiquement tout le code à partir des modèles, sans l'intervention manuelle du développeur. Parmi les générateurs de code assurant une génération 100% automatique à partir des modèles, nous pouvons citer iUML [1] qui se base sur l'approche xUML et le langage d'action ASL, BridgePoint [2] qui se base sur l'approche xtUML et le langage d'action OAL et Rhapsody [3] qui se base sur UML et un langage d'action proche du C++ [44].

Dans ce qui suit nous nous intéressons uniquement à la génération de code à partir des modèles comportementaux d'UML étudiés dans la section précédente à savoir les activités

1. http ://www.kc.com/PRODUCTS/iuml/index.php
2. http ://www.mentor.com/products/sm/model_development/bridgepoint/
3. http ://www.ibm.com/software/awdtools/rhapsody/

et les machines à états. En effet, la génération de code à partir de la partie structurelle ne pose aucun problème surtout si le langage cible est un langage orienté objet. Dans ce cas, une correspondance 1 à 1 entre les éléments du modèle structurel et les éléments du code peut être établie pour les diagrammes de classes les plus élémentaires (une classe UML est transformée en une classe du langage cible, une propriété de la classe UML est transformée en attribut, une opération en méthode et ainsi de suite). Dans certains cas (héritage multiple, associations entre les classes, etc.), la correspondance n'est pas toujours triviale [45]. Cependant, pour les machines à états UML les plus élémentaires, la correspondance 1 à 1 entre les éléments de la machine à états et les concepts des langages de programmation est impossible (pas de notions d'état, d'événement ni de transition dans les langages de programmation). Des patrons (*patterns*) de génération de code ont été crées pour faciliter l'implémentation des machines à états [46]. La section suivante présentera les 3 patterns les plus utilisés dans l'implémentation des machines à états UML.

2.1.2.2 Génération de code à partir des machines à états UML

Il existe plusieurs patterns de génération de code à partir des machines à états UML. Le choix d'un concept du langage cible pour implémenter un élément de la machine à états diffère d'un pattern à un autre. A titre d'exemple, certains patterns de génération de code tel que [47] et [48] représentent les *états* de la machine à états UML à l'aide d'une *énumération* : chaque état correspond à un littéral de cette énumération. D'autres comme [49] choisissent d'implémenter chaque *état* à l'aide d'une *classe* dont les *méthodes* représentent les *événements* déclenchant une transition sortante de l'état. Pour les transitions, certains patterns comme [47] ne les implémentent pas explicitement (une transition est représentée implicitement par le changement de la valeur de l'état courant), d'autres comme [50] les implémentent explicitement en utilisant les classes.

Outre les concepts utilisés pour l'implémentation des éléments d'une machine à états (les états, les événements, les transitions, etc.), les patterns de génération de code se différencient essentiellement par la manière avec laquelle ils implémentent le comportement du système vis-à-vis de l'événement reçu. Les 3 patterns les plus utilisés et qui définissent des implémentations différentes du comportement de la machine à états sont le NSC (*Nested Switch Cases*) [47], le STT (*State Transition Table*) [48] et le SP (*State Pattern*) [49]. La machine à états UML de la figure 2.4 servira de support illustratif dans la description des ces 3 patterns de génération de code [4].

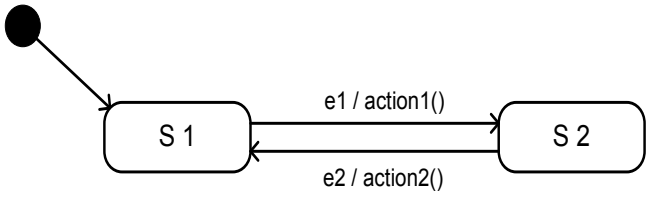


FIGURE 2.4 – La machine à états UML implémentée en utilisant les 3 patterns de génération de code NSC, STT et SP

4. Les codes C++ générés à partir de la machine à états de la figure 2.4 en utilisant les 3 patterns NSC, STT et SP sont disponibles dans l'Annexe A.

NSC pattern

Ce pattern de génération de code est le pattern le plus utilisé pour implémenter les machines à états UML. Généralement les états et les événements sont tous les deux implémentés à l'aide des énumérations. Le comportement du système vis-à-vis des événements reçus est implémenté en utilisant deux *Switch/Cases* imbriqués (d'où le nom du pattern). Le premier *Switch/Cases* détecte la valeur de l'état courant parmi la liste des états de la machine à états, le deuxième *Switch/Cases* détecte l'événement reçu. Une fois la bonne combinaison (état/événement) trouvée, la transition sortante de l'état courant, déclenchée par l'événement reçu est exécutée, l'état courant change ainsi de valeur. La figure 2.5 présente un extrait de l'implémentation de la machine à états de la figure 2.4 en utilisant le pattern NSC.

```
switch (fsm_currentstate) {
case S1:
            switch (evt) {
            case e1:
                  fsm_currentstate= S2;
                  action1();
            break;
            default: break; }
break;
case S2:
            switch (evt) {
            case e2:
                  fsm_currentstate= S1;
                  action1();
            break;
            default: break; }
break;
default: break; }
```

FIGURE 2.5 – Un extrait de l'implémentation de la machine à états UML de la figure 2.4 en utilisant le NSC pattern (deux Switch/Cases imbriqués)

Nous pouvons remarquer que la génération de code en utilisant le NSC pattern est relativement simple. Cependant, pour les machines à états énormes (contenant plusieurs dizaines d'états), cette méthode ne semble pas être la meilleure vu que le code devient peu lisible est très long. Le STT pattern présenté ci après résout le problème de lisibilité de code quand le nombre des états de la machine à états est grand.

STT pattern

Le but des deux *Switch/Cases* imbriquées du pattern NSC est de chercher la transition à franchir selon l'événement reçu et l'état courant du système. Une manière évidente de faire correspondre à un couple de valeurs (état/événement) une troisième valeur (la transition à franchir) consiste à créer un tableau à deux dimensions. C'est le principe adopté par le pattern STT. La figure 2.6 présente un extrait de l'implémentation de la machine à états de la figure 2.4 en utilisant le pattern STT.

Nous pouvons remarquer que le code produit par ce pattern est plus compact que celui produit par le NSC pattern dans le sens où il évite la structure *spaghetti* engendrée par les *Switch/Cases* imbriquées. En effet, pour trouver la transition à franchir, une simple lecture directe dans le tableau liant les états aux événements suffit. Cependant, le tableau peut être une perte de place mémoire dans le cas où la machine possède peu de transitions par état. De

plus, en implémentant des machines à états contenant des transitions avec *gardes* (condition de franchissement d'une transition), plusieurs transitions de *gardes* différentes peuvent être définies pour le même couple (état/événement). La prise en compte des *gardes* complique l'implémentation du pattern STT : pour trouver la transition à franchir, une simple lecture dans le tableau n'étant plus suffisante, il fallait faire des tests sur les *gardes* définies pour les transitions candidates. Le type des éléments du tableau change ainsi d'une unique transition à une collection de transition.

```
STT[0][0]= transitions[0];      // S1 to S2
STT[1][1]= transitions[1];      // S2 to S1

public : void sendevent(fsm_events evt) {

        fsm_currentstate = STT[fsm_currentstate][evt].tostate;
        doAction(STT[fsm_currentstate][evt].func);

}
```

FIGURE 2.6 – Un extrait de l'implémentation de la machine à états UML de la figure 2.4 en utilisant le pattern STT (une lecture directe de la case du tableau STT spécifie la transition à franchir)

Bien que le pattern STT produise un code C++ plus lisible que le pattern NSC, ces deux patterns présentent une implémentation procédurale du comportement des machines à états UML (l'utilisation des *Switch/Cases* pour NSC et d'un *tableau* à deux dimensions pour STT). Une implémentation orientée objet du comportement des machines à états peut apporter une meilleure structuration et lisibilité du code généré. A titre d'exemple, dans une machine à états, chaque état possède ses propres transitions sortantes. Il est sensible à la réception d'un ensemble bien défini d'événements et peut exécuter des comportements spécifiques (les effets de ses transitions sortantes, son action d'entrée (entry), etc.). Ainsi, encapsuler chaque état dans une classe et implémenter ses comportements à l'aide des méthodes de cette même classe peut s'avérer être la solution la plus adéquate pour l'implémentation des états d'une machine à états UML. Le *State Patern* (SP) est l'une des implémentations orientées objet des machines à états.

SP pattern

Le SP est un patron de conception (*design pattern*) de type comportemental. Les patrons de conception [49] fournissent des modèles théoriques qui permettent de résoudre des problèmes récurrents. Ils ont été pensés de façon à ce qu'ils puissent être appliqués avec succès dans les contextes les plus divers. Le SP permet à un objet de modifier son comportement quand son état interne change. Tout se passe comme si l'objet changeait de classe. L'idée principale de ce pattern est d'introduire une classe concrète *Context* et une classe abstraite *State* à partir de laquelle dérivent des classes concrètes implémentant les états de la machine à états UML. La structure du SP appliquée pour la machine à états UML de la figure 2.4 est présentée ci-après (Figure 2.7).

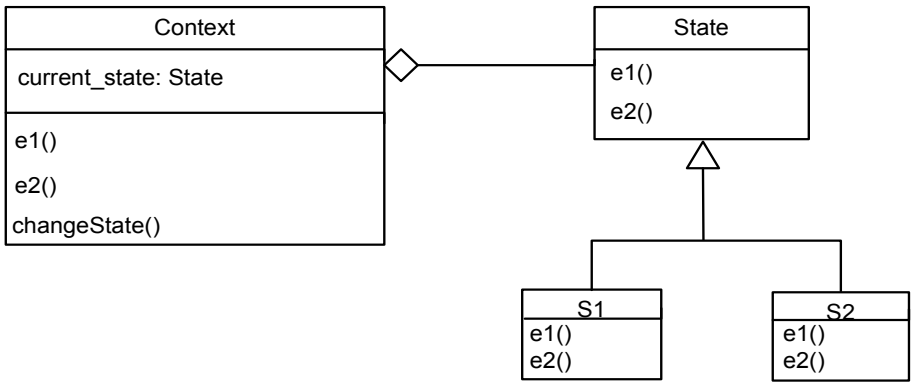


FIGURE 2.7 – La structure du State Pattern appliquée à la machine à états UML de la figure 2.4

La classe *Context* gère la réception des événements (implémentés comme des méthodes de la classe) et le changement de l'état courant (implémenté comme un attribut de la classe). Quand l'instance *Context* reçoit un événement, elle le délègue à l'état concret courant via la classe abstraite *State*. L'état courant exécute le comportement et appelle la méthode *changeState()* de la classe *Context*.

Bien que le code produit par le SP est plus structuré, la taille de l'assembleur généré est beaucoup plus grande que celle obtenue par les autres patterns et ceci à cause du choix d'implémenter les états à l'aide de classes et non pas d'une simple énumération. Le tableau 2.1 résume les avantages et les inconvénients de chaque pattern étudié.

Pattern utilisé	Avantages	Inconvénients
NSC	Facile à implémenter	Code peu lisible pour des grandes machines
STT	Exécution rapide de code	Code binaire qui n'est pas compact pour certains types de machines
SP	Code structuré et lisible	Code binaire qui n'est pas compact

Tableau 2.1 – Tableau comparatif des trois patterns de génération de code à partir des machines à états UML

Le choix d'un pattern dépend du besoin du développeur : la facilité de l'implémentation, la lisibilité du code généré, la performance du code binaire (taille de l'exécutable, temps d'exécution), etc. Dans le cadre de ce travail de recherche, nous nous intéressons à la taille du fichier assembleur généré. Nous allons (dans le Chapitre6) comparer la taille du code assembleur produit par des générateurs de code commerciaux en utilisant ces 3 patterns.

2.1.2.3 Génération de code à partir des activités

Les activités UML sont souvent utilisées pour la spécification des parties algorithmiques des opérations (le corps des opérations). Pour cela, une activité est une suite d'actions (qui représentent les instructions d'un programme) liées entre elles par des arcs de flot de contrôles (qui représentent l'enchainement des instructions) et des nœuds de contrôles tels

que les nœuds de décision (*Décision Node*) transformés au niveau code à des instructions de condition (*if/else*). Pour les activités les plus élémentaires, la génération de code est relativement simple, il suffit de remplacer chaque action UML par l'instruction du langage de programmation correspondante. A titre d'exemple, l'action *AddStructuralFeatureAction*, qui change la valeur vers « `valeur` » d'une propriété « `p` » d'un objet « `o` », est remplacée par l'instruction « `o.p=valeur;` ». De la même manière, l'action *CreateObjectAction* qui crée l'objet « `o` » de type `Object` est remplacée par l'instruction java « `Object o=new Object ();` » et ainsi de suite. Cependant, les diagrammes d'activités UML peuvent contenir des nœuds qui n'ont pas leurs correspondants directs en langages de programmation classiques. A titre d'exemple, les *fork* et les *join* qui expriment le fait que deux actions peuvent s'exécuter en parallèle n'ont pas de correspondants directs dans les concepts de base des langages de programmation classiques. Pour générer du code à partir de ces 2 nœuds de contrôle, les développeurs ont besoin de faire appel à des bibliothèques externes telles que la bibliothèque « *Thread* » pour le langage Java et la bibliothèque « *pthread* » pour le C++. Dans ce cas, des appels de primitives définies pour ces bibliothèques correspondront à l'implémentation des *fork* et *join* des diagrammes d'activités.

2.1.2.4 Synthèse

La génération automatique de code offre de nombreux avantages pour le développement des logiciels, y compris une productivité accrue et une meilleure consistance du code source produit. Cependant, nous avons vu que la génération de code exécutable (provenant des diagrammes comportementaux d'UML tels que les machines à états et les activités) n'est pas évidente et exige des techniques différentes (les patterns de génération de code pour les machines à états, les langages d'action et les appels aux libraires pour les activités, etc.). Ceci est dû au fait que les langages de programmation ne se trouvent pas au même niveau d'abstraction que les modèles UML et ne fournissent pas les mêmes concepts.

Van Gurp et Bosch [50] proposent d'étendre les concepts des langages de programmation en ajoutant explicitement de nouvelles entités dans le langage cible qui correspondent exactement aux concepts UML non existants. Eichberg [45] a rencontré le même problème mais cette fois-ci en implémentant les diagrammes de classes. Il a proposé la même solution que [50] à savoir l'extension des langages de programmation. En effet, même l'implémentation des classes peut révéler des problèmes de correspondance (les associations entre les classes n'ont pas de correspondants directs dans les langages de programmation). Étendre les concepts des langages facilitera la génération d'un code lisible. Cependant, pour compiler ce code généré, il fallait soit exprimer les nouveaux concepts à l'aide des concepts existants (on revient ainsi au même problème de correspondance), soit étendre le compilateur de code [51] pour qu'il prenne en compte les nouveaux éléments syntaxiques ajoutés au langage source (qui n'est pas évident vu la complexité croissante des compilateurs). L'étape de la compilation du code généré sera décrite dans la section suivante.

2.1.3 Troisième étape : la Compilation de code

Généralement, le code généré à partir des modèles UML est le C, C++ ou Java bien que des générateurs de code comme [52] et [53] ciblent parfois d'autres langages tel que VHDL et SystemC. Dans la suite de ce travail de recherche, nous considérons C++ comme langage cible des générateurs de code à partir des modèles UML exécutables. Ainsi, il représente le langage source du compilateur qui produira l'exécutable de l'application (Figure 2.1). Plusieurs compilateurs existent sur le marché pour la production du code exécutable à partir des programmes C++. Certains sont payants tel que ICC (*Intel C++ Compiler*)[5], MS C++ (Microsoft C++)[6], d'autres sont gratuits tels que Clang [54] et GCC (Gnu Compiler Collection). Nous avons choisi d'étudier le compilateur GCC puisqu'il est open source et très utilisé pour compiler le code produit par les générateurs de code des modèles UML tels que iUML, BridgePoint et Rhaspdoy. Avant d'étudier le compilateur GCC, nous présentons d'une façon générale la définition et l'architecture d'un compilateur de code.

2.1.3.1 Généralité sur les compilateurs

Aho et al. dans [13], définissent un compilateur comme étant un logiciel qui lit un programme écrit dans un premier langage - le langage source - et le traduit en un programme équivalent écrit dans un autre langage -le langage cible. Généralement, l'architecture d'un compilateur comporte deux phases : une phase d'analyse et une phase de synthèse. Chaque phase est formée à son tour de plusieurs activités. La figure 2.8 présente l'architecture typique d'un compilateur. La phase d'analyse comporte généralement 3 activités : une analyse lexicale, une analyse syntaxique et une analyse sémantique. La phase d'analyse crée une représentation intermédiaire que la phase de synthèse se charge de transformer vers le langage cible (dans notre cas le langage binaire).

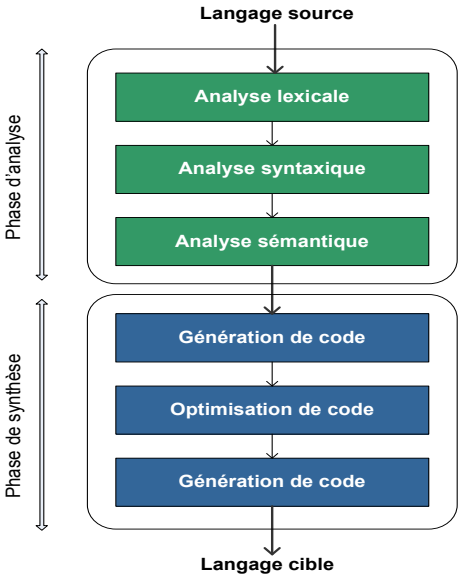


FIGURE 2.8 – Architecture générale d'un compilateur optimisant

5. http ://software.intel.com/en-us/articles/intel-compilers/
6. http ://www.microsoft.com/visualstudio/en-us/products/2010-editions/visual-cpp-express

L'étape d'optimisation du code se fait en général au cours de la phase de synthèse dont le nombre de passes dépend du nombre de formes intermédiaires. L'analyse lexicale regroupe les caractères isolées qui constituent le texte source sous forme d'unités lexicales (appelées encore *tokens* ou *symboles*), qui représentent les mots du langage. Alors que l'analyse lexicale reconnait les mots du langage, l'analyse syntaxique en reconnait les phrases. Le rôle principal de cette passe est de dire si le texte source appartient au langage considérée, c'est-à-dire s'il est correct relativement à la grammaire de ce dernier. La structure du texte source étant correcte, l'analyse sémantique consiste à vérifier certaines propriétés sémantiques (relatives à la signification de la phrase et de ses constituants) comme la vérification des types des variables et la déclaration des variables avant leurs utilisations pour certains langages de programmation. Pour illustrer ces 3 passes, [13] décrit la phase d'analyse de l'instruction suivante *position = initiale + vitesse *60* (Figure 2.9).

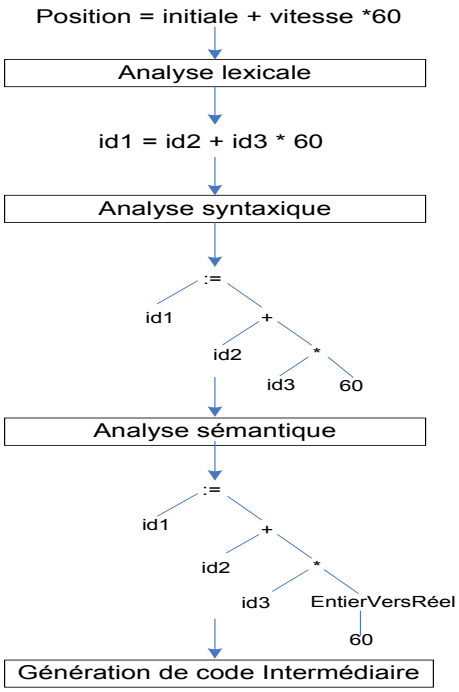


FIGURE 2.9 – Les 3 passes d'analyse pour l'instruction : position= initial + vitesse * 60

D'après cette figure, l'analyseur syntaxique fait appel à l'analyseur lexical pour connaitre le prochain symbole dans la chaine à analyser et produit l'arbre de l'analyse qui sera complété par l'analyseur sémantique. La sortie de l'analyseur sémantique est *l'arbre de syntaxe abstraite* AST qui va être transformé par la suite à la phase de synthèse du compilateur. La phase de synthèse est une suite d'étapes de génération de code allant de l'AST vers le binaire. Le nombre de formes intermédiaires et de passes d'optimisation du code généré dépendent du compilateur choisi. La figure 2.10 complète la chaine de compilation de l'instruction *position = initiale + vitesse *60* en partant de l'AST produite par la phase de l'analyse (Figure 2.9). Une seule optimisation (la propagation de constante) et deux formes intermédiaires sont utilisés (la représentation à 3 adresses et l'assembleur).

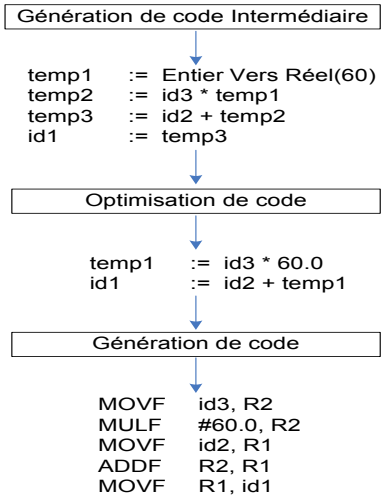


FIGURE 2.10 – La phase de synthèse pour l'instruction position=initial + vitesse *60

Nous allons voir par la suite que pour le même compilateur, le nombre de représentations intermédiaires peut changer d'une version à une autre et que le nombre d'optimisations effectuées peut changer selon le critère d'optimisation choisi. Le compilateur que nous étudions tout au long de ce rapport de recherche est le compilateur GCC.

2.1.3.2 Compilateur GCC

GCC permet de compiler plusieurs langages sources (C, C++, Java, Ada, etc.) vers plusieurs architectures de plateformes (Intel, AMD, PowerPC, etc.). Le code C++ compilé en utilisant GCC peut être exécuté sur plus de 40 architectures différentes[7]. Cela présente un avantage de GCC par rapport à d'autres compilateurs dont le code binaire généré ne peut être exécuté que sur une architecture bien définie. Bien que le compilateur GCC soit une collection de compilateurs de plusieurs langages sources, cela ne veut pas dire que pour chaque couple (langage source, architecture cible) GCC implémente un nouveau compilateur. La figure 2.11 montre que la structure interne du compilateur GCC permet d'implémenter uniquement 6 transformations (au lieu de 9) pour compiler 3 langages différents vers 3 plateformes différentes.

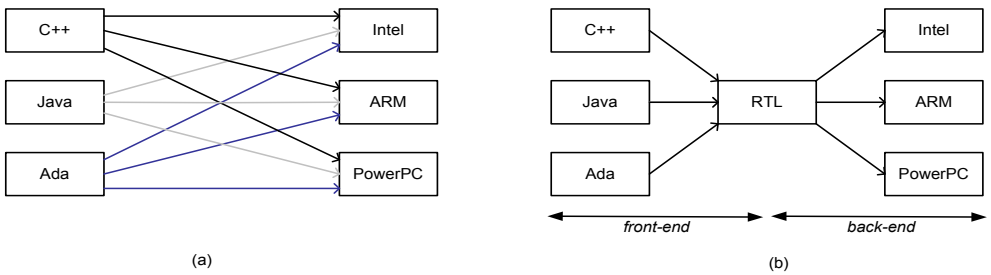


FIGURE 2.11 – La compilation de 3 langages différents vers 3 plateformes différentes : (a) 9 compilateurs sont exigés (b) GCC implémente uniquement 6 transformations

7. http ://gcc.gnu.org/onlinedocs/gcc-4.6.1/gcc/Submodel-Options.html# Submodel-Options

Cela est assuré (dans les premières versions de GCC) grâce à la forme intermédiaire RTL (*Register Transfer Level*) [55] qui divise l'architecture du compilateur GCC en deux parties : le *front-end* et le *back-end* (Figure 2.11). La partie frontale (*front-end*) contient les transformations des langages sources vers la forme RTL. Ces transformations sont dépendantes du langage source mais indépendantes de l'architecture cible. La dernière partie (*back-end*) contient les transformations de la forme RTL vers le binaire. Les transformations du *back-end* sont (contrairement à celle du *front-end*) indépendantes du langage source mais dépendantes de l'architecture cible. La figure 2.12 présente la première architecture du compilateur GCC sur laquelle sont basées toutes les versions qui précèdent la version GCC 4.0.

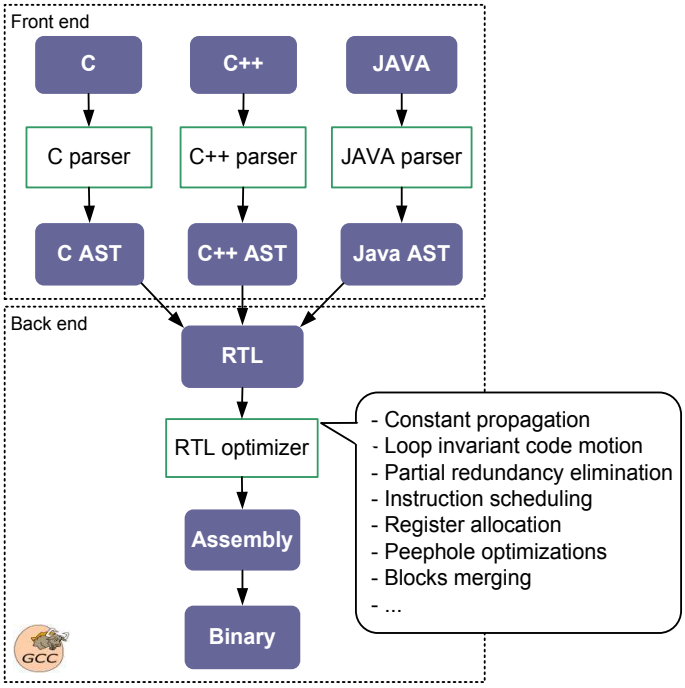


FIGURE 2.12 – Architecture du compilateur GCC avant la version 4.0

Après l'analyse du fichier C++ source, GCC produit l'AST à partir duquel la forme RTL est générée. RTL est une forme intermédiaire de bas niveau très proche de l'assembleur. Elle a une syntaxe proche du langage Lisp. La figure 2.13 présente une instruction RTL pour la somme de deux entiers et son équivalent en langage assembleur.

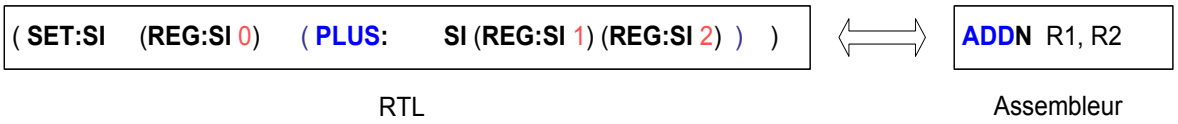


FIGURE 2.13 – Une instruction RTL de somme de deux entiers simples (SI) et son équivalent en code assembleur

Cette instruction indique que le résultat de la somme (PLUS) des deux entiers simples (SI) se trouvant dans les deux registres (R1 et R2) sera stocké dans le registre (R0). Nous remarquons ainsi que la forme RTL tout comme l'assembleur utilise la notion de registres. Le nombre de registres dans la forme RTL (contrairement à l'assembleur) est illimité. La

particularité de la forme RTL est qu'elle se trouve à un niveau plus haut que l'assembleur et que plusieurs optimisations liées essentiellement à la plateforme cible sont implémentés au niveau RTL (Figure 2.12). En revanche, n'ayant pas la notion de types des langages de programmation (les différents types de collections : liste, tableaux, classes, etc.) et n'exprimant pas bien le graphe de flot de contrôle des programmes, la forme RTL est considérée comme une forme de bas niveau utile pour les optimisations de bas niveau (liées à la plateforme) mais ne convient plus aux optimisations de haut niveau.

L'évolution du pouvoir d'optimisation du compilateur GCC pour résoudre les problèmes de la forme intermédiaire RTL sera étudiée plus en détail dans la section suivante qui sera consacrée à l'étude des travaux d'optimisation existants dans les différentes étapes de l'approche classique dirigée par les modèles pour le développement des systèmes embarqués.

2.1.4 Bilan

Dans cette section, nous avons présenté l'approche classique dirigée par les modèles pour le développement des systèmes embarqués. Cette approche comporte 3 étapes : la modélisation, la génération de code et la compilation du code généré. En se basant sur l'IDM, cette approche facilite la gestion de la complexité croissante des systèmes embarqués. De plus, la génération automatique du code à partir des modèles exécutables augmente la productivité et améliore la qualité du code produit en termes de lisibilité et de consistance avec le modèle. Cette approche profite aussi de l'évolution des compilateurs de code qui assurent (comme le compilateur GCC) la production (à partir du même code C++) des exécutables optimisés tournants sur plusieurs plateformes différentes.

Cependant, cette approche présente des problèmes liés essentiellement à la performance du code exécutable produit. En effet, les générateurs de code (à partir des modèles) actuels se concentrent sur la production d'un code lisible en essayant d'utiliser au maximum les concepts des langages de programmation facilitant ainsi la compréhension du code généré (les tableaux à deux dimensions, les classes, etc.). Ceci est fait au détriment de la performance du code qui ne cesse de se dégrader. En effet, les systèmes embarqués à ressources limités exigent que le code produit soit le plus optimisé possible. Malheureusement, les développeurs de ces systèmes ne sont pas satisfaits par les performances du code produit et modifient ce dernier à la main pour l'optimiser. De telle modification ont des conséquences néfastes sur les validations et les analyses que les concepteurs ont tendances à faire au niveau modèle. Pour éviter ce genre de problème et pour produire un code le plus optimisé possible à partir des modèles UML, plusieurs optimisations ont été améliorées. La section suivante présente l'évolution du pouvoir d'optimisation dans l'approche classique basée sur l'IDM pour le développement des systèmes embarqués.

2.2 Optimisations dans l'approche classique dirigée par les modèles

Les optimisations représentent les modifications du programme source ou du programme cible qui visent à augmenter les performances de ces derniers. Il existe plusieurs critères d'optimisation : l'optimisation de la taille du code exécutable, du temps d'exécution, du temps de compilation, de la consommation d'énergie, etc. Dans ce travail de recherche, nous considérons un seul critère d'optimisation : l'optimisation de la taille du code exécutable généré. En effet, les systèmes embarqués assurent souvent plusieurs fonctionnalités à la fois (exemple des téléphones portables). La taille limitée des mémoires de ces systèmes obligent le développeur à optimiser au maximum la taille du code relatif à chaque fonctionnalité et ceci pour pouvoir intégrer plus de fonctionnalités possibles dans le même système. L'approche classique dirigée par les modèles pour le développement des systèmes embarqués se base généralement sur les optimisations du compilateur. La section suivante développe les optimisations du compilateur GCC.

2.2.1 Optimisations du compilateur GCC

Le compilateur GCC offre plusieurs options pour optimiser le code cible (le code binaire produit à partir du code C++). Chacune de ses options optimise le code binaire selon un critère d'optimisation particulier. A titre d'exemple, pour produire le code binaire le plus compact, l'option -Os doit être spécifiée. L'option -O3 est utilisée pour produire le code dont le temps d'exécution est réduit au maximum. Les optimisations qui risquent d'augmenter le temps de compilation du code C++ ne sont pas prises en compte dans le niveau -O1 (le niveau par défaut). Ce dernier critère d'optimisation (la réduction du temps de compilation) n'est pas intéressant dans notre cas, puisque le code des fonctionnalités d'un système embarqué est souvent compilé sur une autre machine et embarqué par la suite dans la mémoire du système embarqué. Le tableau 2.2 résume les différentes options offertes par le compilateur GCC pour produire un code exécutable optimisé.

Option	Objectif des optimisations supportées
- O1	Minimiser la taille du code et le temps d'exécution. Ne pas inclure les optimisations qui augmentent le temps de compilation
- O2	Toutes les optimisations de -O1 plus celles qui diminuent encore plus le temps d'exécution
- O3	Toutes les optimisations de -O2 même celles qui augmentent la taille (e,g, inlining)
- Os	Toutes les optimisations de -O2 sauf celles qui augmentent la taille du code. Inclure d'autres optimisations pour réduire au maximum la taille du code (e.g, prefetching)

Tableau 2.2 – Les options (*flags*) d'optimisation du compilateur GCC (-Os utilisé pour produire le code le plus compact)

Ce tableau classifie les optimisations du GCC selon le critère d'optimisation. Une autre classification possible des optimisations du GCC se base sur le niveau d'abstraction. On note (Figure 2.12) des optimisations de bas niveau (qui dépendent des informations de

la plateforme) telles que l'allocation de registres (*Register allocation*), l'ordonnancement des instructions (*Instruction Scheduling*), etc. D'autres optimisations sont de haut niveau (indépendants de la plateforme cible) telles que l'élimination de code mort (*dead code elimination*), la propagation de constante (*constant propagation*), etc.

2.2.1.1 Optimisations de la forme RTL

Avant la version GCC 4.0, toutes les optimisations du compilateur GCC s'effectuaient au niveau RTL (Figure 2.12). Cependant, certaines optimisations ont besoin d'informations de plus haut niveau d'abstraction tel que les types et les structures de contrôle. En effet, nous avons vu dans la section 2.1.2.2 que l'utilisation d'un tableau à deux dimensions pour implémenter les machines à états (le pattern STT) peut générer un code s'exécutant plus vite que le code produit en utilisant les switch/cases (le pattern NSC). L'utilisation de certains types de données (les tables de hachage, les arbres, etc.) disponibles dans les langages de haut niveau peut résulter en un code plus performant que le code produit en utilisant des tableaux ou des listes. De plus, la forme RTL utilise des concepts de bas niveau tels que les registres, les *jumps* et la pile qui sont inutiles pour les optimisations de haut niveau [56]. Ainsi, l'implémentation des optimisations de haut niveau telles que l'élimination du code mort et la propagation de constante dans un niveau plus haut que le niveau RTL résulte en des algorithmes moins compliqués et plus rapides [57].
En passant de la forme AST vers la forme RTL, plusieurs informations de haut niveau utiles pour les optimisations sont perdues (quelques types de données, le graphe de flot de contrôle, etc.). D'autres informations de bas niveau (utiles pour les optimisations de bas niveau) ont été ajoutées (les registres, la pile, etc.). Elles représentent des informations parasites qui peuvent freiner les algorithmes des optimisations de haut niveau. Ainsi, implémenter les optimisations de haut niveau dans un autre niveau d'optimisation (différent de RTL) s'avère nécessaire. D'après l'architecture du compilateur GCC (Figure 2.12), la seule forme candidate est la forme AST. Cependant, cette forme, bien qu'elle conserve le flot de contrôle et les types de données, est dépendante du langage source. Implémenter des optimisations au niveau AST pour tous les *front-ends* du compilateur GCC implique une réécriture de ces optimisations pour chaque *front-end*. Une représentation commune à tous les *front-ends* s'avère nécessaire. C'est ainsi que l'architecture du compilateur GCC a changé pour introduire une nouvelle partie (autre que le *front-end* et le *back-end*) dans laquelle les optimisations de haut niveau seront implémentées. Cette nouvelle partie est appelée *middle-end*(Figure 2.14).

Une nouvelle représentation de la forme AST, générique pour tous les *front-ends* est ainsi créée. Elle est nommée GENERIC [58]. La forme GIMPLE est la simplification de la forme GENERIC. En effet, GIMPLE est inspirée de la forme intermédiaire SIMPLE utilisée dans le développement du compilateur McCat [59]. Cette forme n'autorise que des expressions à 3 adresses (maximum 2 opérandes par opération) introduisant ainsi des variables temporaires. La forme SSA (*Static Single Assignment*) est le résultat de l'application du principe de l'assignation unique des variables sur la forme GIMPLE.

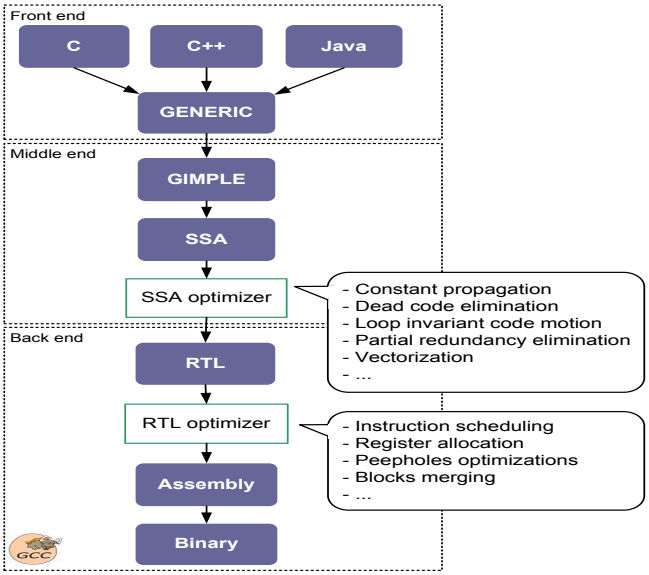


FIGURE 2.14 – L'architecture actuelle du compilateur GCC : 3 parties *front-end*, *middle-end* et *back-end*

2.2.1.2 Optimisations de la forme SSA

Les optimisations de haut niveau du compilateur GCC sont désormais implémentées dans le *middle-end*, au niveau SSA. Cette forme a été développée par Cytron [57] en 1991. Quelques années plus tard, elle est devenue la forme intermédiaire la plus utilisée pour implémenter les optimisations de haut niveau des compilateurs. Plusieurs compilateurs utilisent la forme SSA tels que HotSpot [8], GCC et LLVM [9]. Dans la forme SSA, une variable ne peut être définie qu'une seule fois. Cette forme permet de rendre la relation entre les définitions et les utilisations des variables (*Def/Use Chain*) explicite. Elle simplifie ainsi la mise en œuvre des algorithmes d'analyse de flot de données et des optimisations de haut niveau basées sur le flot de données. La forme SSA est basée sur le graphe de flot de contrôle (GFC) produit à partir de la forme GIMPLE. Le GFC est un graphe direct formé d'un ensemble de nœuds appelé blocs de bases (*bb*) et d'arcs reliant ces blocs. Les blocs de bases de ce graphe représentent les calculs séquentiels (pas de branchements ni de boucles) tandis que les arcs représentent les enchaînements potentiels des calculs. La figure 2.15 présente un simple programme C++ et ses 4 formes intermédiaires : la forme GENERIC, la forme GIMPLE, le GFC et la forme SSA. Pour passer de la forme GENERIC à la forme GIMPLE, des variables temporaires (T1, T2,T3) ont été ajoutées pour simplifier les expressions. Le GFC relatif à ce programme est formé de 4 blocs de base et 4 arêtes. Mettre un bloc de base sous forme SSA est simple puisque la provenance d'une variable est unique. Dans le cas d'un graphe de flot de contrôle, la provenance d'une variable peut être multiple. Pour résoudre ce problème, la forme SSA utilise une fonction fictive $\phi()$. D'après la figure 2.15, cette fonction sélectionne en fonction des branchements la valeur qui sera affectée à a4 (dernier bloc de base de la forme SSA).

8. http ://openjdk.java.net/groups/hotspot
9. http ://llvm.org/

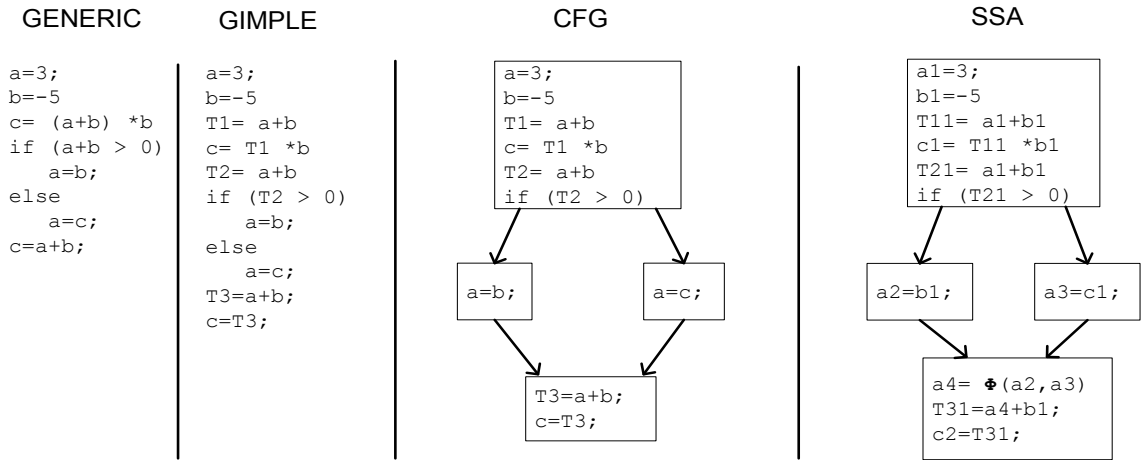


FIGURE 2.15 – Les 4 formes intermédiaires GENERIC, GIMPLE, CFG et SSA du même programme C++

Partant de la forme SSA, et en analysant l'instruction T11= a1+b1, le compilateur GCC pourra remplacer la valeur de T11 par -2 directement, car il est sûr que les valeurs a1 et b1 sont assignées une seule fois. Ceci facilite la tâche d'analyse du flot de données par le compilateur GCC et par conséquent la détection des constantes et des codes morts. La forme SSA basée sur le GFC a amélioré considérablement les algorithmes des optimisations de haut niveau tels que la propagation de constante, l'élimination de code mort et l'élimination des expressions redondantes [57]. La forme SSA a également contribué à l'ajout d'autres optimisations de haut niveau dont l'implémentation au niveau RTL était impossible. Ce sont les optimisations liées à des structures de données et de contrôle absents dans la forme RTL (les boucles, les pointeurs, etc.). Parmi les nouvelles optimisations introduites dans la forme SSA nous trouvons les optimisations qui transforment les corps des boucles, les optimisations liées à l'analyse des pointeurs et des tailles des tableaux et l'auto-vectorisation [60].

L'ajout de la forme SSA et de la partie *middle-end* a amélioré considérablement le pouvoir d'optimisation du compilateur GCC. D'autres types d'optimisation telles que les optimisations au niveau du linker : LTO (*Link Time Optimization*) [61] et les optimisations intra procédurales (IPA) [62] améliorent encore plus le pouvoir d'optimisation du compilateur GCC. Malgré cette évolution et la définition de plusieurs niveaux et types d'optimisations (plus que 200 passes d'optimisation), le compilateur GCC n'arrive pas à produire un code assembleur compact en compilant le code C++ produit à partir des modèles UML exécutables. En effet, certaines informations liées à la sémantique du langage de modélisation UML et utiles pour les optimisations sont perdues lors de la génération de code [63]. Le compilateur prenant en entrée le code généré, est incapable d'exploiter les informations perdues lors de la génération de code pour des fins d'optimisation. Ce point sera clarifié dans le Chapitre 4. La section suivante présente quelques techniques d'optimisation utilisées par les générateurs de code.

2.2.2 Optimisations des générateurs de code

Dans cette section, nous allons développer deux techniques d'implémentation des optimisations au cours de la génération de code. La première consiste à modifier le pattern de génération de code (la relation de correspondance entre les éléments du modèle exécutable et les concepts du langage de programmation cible). Les optimisations du générateur de code BridgePoint qui implémente l'approche xtUML sont basées sur cette alternative. Les patterns de génération de code utilisés sont appelés *Archetypes* [6].

La deuxième alternative consiste à transformer le modèle UML de départ vers une forme intermédiaire adaptée pour les optimisations. Plusieurs générateurs de code académiques tels que RIALTO [64] [65] et SCOPE [66] [67] adoptent cette technique. SCOPE s'intéresse plutôt à la génération de code optimisé à partir des machines à états UML hiérarchiques. Une analyse statique de la taille des variables utilisées pour encoder les éléments syntaxiques de la machine à états (les états, les transitions, etc.) est effectuée sur la forme intermédiaire cible (*Hierarchy TREE*) [67] très spécifique, comme son nom l'indique, aux machines à états hiérarchiques. Ainsi, nous avons choisi d'étudier la forme intermédiaire de l'outil RIALTO (la forme S-Graph), plus générale, définie à l'origine par l'approche Polis [11] de développement des systèmes embarqués.

2.2.2.1 Archetypes de l'approche xtUML

BridgePoint est l'un des générateurs de code capable de produire automatiquement un code optimisé à partir des modèles exécutables conformes au langage xtUML (section 2.1.1). La génération de code assurée par BridgePoint est une transformation de modèle à texte M2T (section 2.2). Cette transformation est réalisée par la définition d'un ensemble de règles et de patterns de génération de code appelés *Archetypes*. Si le code C++ produit ne correspond pas aux performances attendues, le concepteur peut modifier *l'Archetype* et régénérer le code C++. Plusieurs *Archetypes* sont implémentés dans le générateur de code. Le concepteur peut modifier les *Archetypes* existants comme il peut en créer de nouveaux qui correspondent au mieux aux exigences de performances. Le mécanisme de la sélection de l'ensemble des *archteypes* est appelé *Marking*. Ainsi, l'utilisateur contrôle la génération de code en choisissant des *marks*. Ce mécanisme est semblable au mécanisme de contrôle de performance du code utilisé par le compilateur GCC qui offre différentes options d'optimisation (Tableau 2.2). Cependant, le *marking* offert par BridgePoint exige une intervention du concepteur qui choisi les *marks* à utiliser en modifiant les fichiers *.mark* du générateur de code. À titre d'exemple, si l'utilisateur estime que le code généré à partir d'une opération UML n'est pas utile, ce dernier peut choisir le *mark* qui empêche le générateur de code de produire un code pour cette opération et ceci en ajoutant dans un *fichier.mark* l'invocation d'une règle de génération de code (`.invoke TagFunctionTranslationOff( "Operation_name" )`)qui prend comme paramètre le nom de l'opération.

Bien que l'interaction avec l'utilisateur implique une maitrise du générateur de code et un contrôle de la qualité du code produit, le choix à la main des *marks* exige une connaissance des différentes règles de génération de code et leurs impacts sur la performance du code produit. Le compilateur GCC, a échappé à ce problème en regroupant les optimisations selon le critère d'optimisation assuré (Tableau 2.2). Ainsi, l'utilisateur de GCC (contrairement à

celui du BridgePoint) choisit uniquement le critère d'optimisation, sans avoir une idée exacte sur toutes les optimisations qui s'exécutent derrière.

Certaines optimisations prises en compte par le générateur de code en spécifiant des marks particuliers telles que (ne pas générer du code pour l'opération *set_Propriété()* si une propriété d'une classe UML n'est pas utilisée dans le code C++ généré, ne pas générer un code pour une propriété d'une classe UML si la propriété n'est pas utilisé dans le code C++ généré, etc.) sont des optimisations que le compilateur GCC (utilisé par l'outil BridgePoint pour produire le code binaire) peut traiter en exécutant l'optimisation de *l'élimination de code mort*. Ainsi, il est inutile d'optimiser le code C++ si le compilateur GCC est capable d'optimiser le binaire correspondent.

L'utilisation des *archteypes* et des *marks* peut améliorer les performances du code binaire produit à partir des modèles xtUML (implémenter une collection de données en utilisant des tableaux au lieu des listes, etc). Cependant, implémenter les optimisations au cours de la génération de code rend les générateurs de code trop complexes et difficile à certifier (le critère de certification de code est de plus en plus considéré pour les systèmes embarqués temps réels notamment pour les secteurs de l'automobile et l'avionique [68]).

Une autre alternative consiste à implémenter les optimisations dans une forme intermédiaire entre le modèle UML et le code C++. Ainsi, le générateur de code prend en entrée la forme intermédiaire optimisée et génère du code C++ directement à partir de cette forme sans exécuter aucune fonctionnalité autre que la correspondance entre les éléments de la forme intermédiaire et les concepts du code cible. L'approche Polis pour le développement des systèmes embarqués se base sur cette alternative pour produire un code optimisé à partir des machines à états.

2.2.2.2 Forme S-Graph de l'approche Polis

L'approche Polis [11] est une approche de co-design (développement conjoint des deux parties matérielle et logicielle) des systèmes embarqués. La partie contrôle des systèmes embarqués est modélisée à l'aide d'une extension des FSM appelées CFSM (*Co-design Finite State Machine*). Pour la synthèse de la partie logicielle, un code C est généré à partir des CFSM. Cependant, pour produire un code binaire optimisé, cette synthèse est faite en deux étapes : la première consiste à générer à partir des machines à états un graphe de flot de contrôle (*Control Flow Graph*) nommé S-Graph (*Software Graph*) [69] et la deuxième étape consiste à générer du code C à partir des S-Graph. Un S-Graph est un graphe acyclique direct ayant une seule source (*Begin*) et une seule fin (*End*). Ces nœuds sont de 4 types : les nœuds de test (TEST), les nœuds d'affectation (ASSIGN) et les deux nœuds Begin et End. Chaque nœud de TEST possède deux branches (arêtes) *false_branche* et *true_branche*. Par la suite, un nœud de Test a deux nœuds fils chacun provenant d'une branche. Un nœud d'affection n'a qu'une seule branche et par la suite un seul nœud fils. Les nœuds représentent des calculs sur des variables et les arêtes représentent le flot de contrôle entre les nœuds. La figure 2.16 présente une machine à états UML et son S-Graph équivalent. Les nœuds de forme rectangulaire représentent les nœuds de type ASSIGN. Une ligne en pointillés représente la *false_branche* d'un nœud de test. Une ligne pleine représente la *true_branche*.

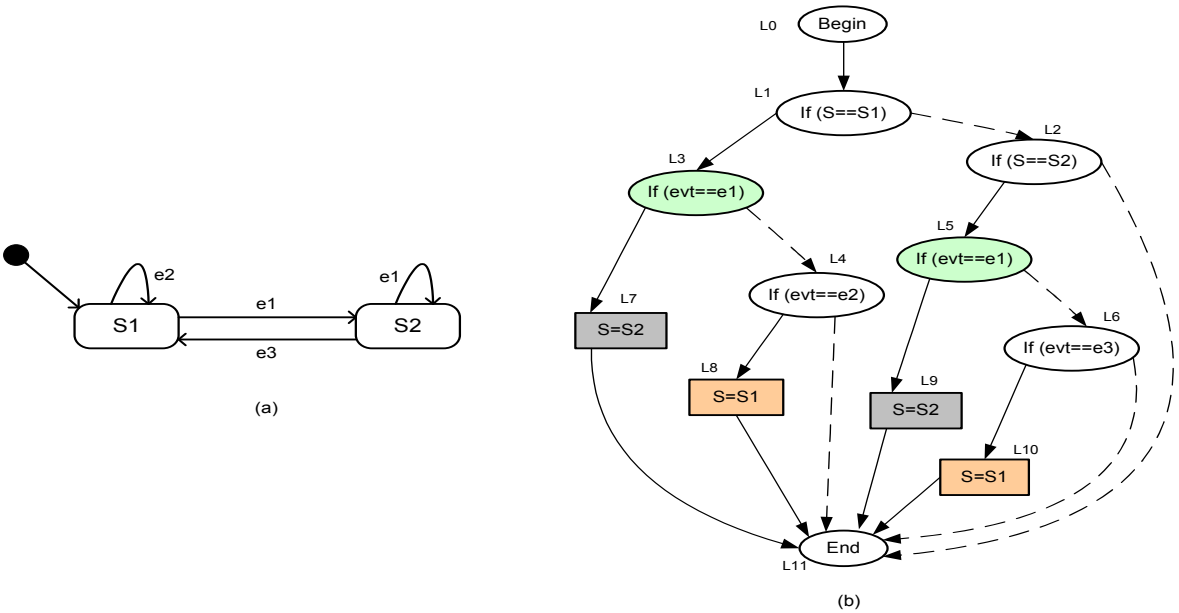


FIGURE 2.16 – (a) Une machine à états UML (b) son S-Graph équivalent contenant 2 couples de nœuds *isomorphes* : (L7, L9) et (L8, L10) et un couple de nœuds *semi-isomorphes* (L3, L5)

Les optimisations définies au niveau S-Graph permettent de minimiser le nombre de nœuds du graphe et par la suite de produire un code binaire plus compact. Parmi ces optimisations nous pouvons citer la réorganisation des variables d'entrée et de sortie pour minimiser la profondeur du graphe en utilisant l'algorithme de Sift [70] et l'élimination des nœuds *isomorphes*. Deux nœuds sont *isomorphes* s'ils ont le même label (même expression) et leurs descendants sont aussi i*somorphes*. A titre d'exemple, les deux nœuds `L7` et `L9` du S-Graph de la figure 2.16 sont *isomorphes*, de même pour les nœuds `L8` et `L10`.

L'algorithme de construction du S-Graph à partir des CFSM défini dans [11] a été amélioré par [64] de façon à produire directement un graphe *canonique* (ne contenant pas de nœuds *isomorphes*). En effet, l'auteur de [64] a repris la même forme S-Graph définie dans [11] et l'a utilisée comme une forme intermédiaire du générateur de code RIALTO développé dans sa thèse. RIALTO prend en entrée les machines à états UML et génère en sortie du code (C ou VHDL). L'auteur de [64] a ajouté d'autres optimisations sur la forme S-Graph telles que l'élimination des nœuds *semi-isomorphes*. Deux nœuds de test sont *semi-isomorphes* s'ils ont la même expression, les mêmes *true_branches* mais deux *false_branches* différentes. Les nœuds formants les *false_branches* doivent avoir exactement un seul parent. En effet, l'idée est de faire descendre les nœuds *semi-isomorphes* jusqu'à atteindre le nœud final (END). Dans ce cas ces nœuds deviennent *isomorphes* (leurs *false_branches* pointent toutes les deux vers le même nœud : le nœud final). La présence de deux parents dans un nœud de la *false_branche* bloque la décente du nœud *semi-isomorphe*. Le S-Graph de la figure 2.16 contient deux nœuds *semi-isomorphes* : `L3` et `L5`.

La figure 2.17 présente deux autres optimisations définies dans le générateur de code RIALTO : (1) la factorisation des nœuds d'affectation des valeurs des états (`S=S1 et S=S2`)

en un seul nœud en utilisant l'opérateur d'incrémentation S++ et ceci uniquement dans le cas ou les états peuvent être présentés dans le langage cible par une énumération { `S1, S2, S3` } (Figure 2.17 (b)) et (2) la factorisation des nœuds de TEST (`if(S==S1)`) et (`if (S==S2)`) en un seul nœud (`if (S=S1) | (S=S2)`) en utilisant l'opérateur logique OU (Figure 2.17 (c)).

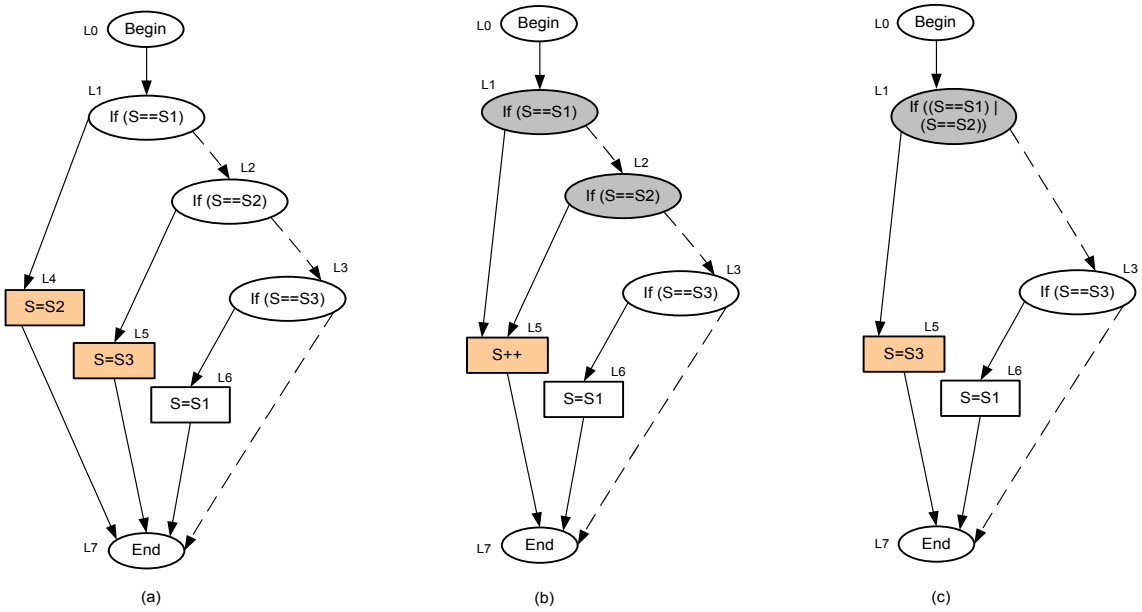


FIGURE 2.17 – (a) un exemple de S-Graph non optimisé (b) le S-Graph résultant de factorisation des nœuds de type *Assign* en utilisant l'opérateur d'incrémentation S++ (c) le S-graph résultant de la factorisation des nœuds de type TEST en utilisant l'opérateur OU

L'avantage de ce type de génération de code (génération à plusieurs étapes en se basant sur des formes intermédiaires) est la séparation des deux tâches différentes : l'optimisation et la génération de code (contrairement à la génération de code de l'outil BridgePoint étudiée dans la section précédente qui assure en une seule étape la génération de code et l'optimisation). Cependant, nous allons montrer dans cette thèse que pour certaines machines à états UML, le S-Graph optimisé produit suite à l'exécution des optimisations définies dans [64] n'est pas optimal et qu'il peut être encore optimisé résultant ainsi en un code binaire plus compact (Chapitre 5).

Dans cette section, nous avons étudié quelques travaux d'optimisations au cours de la génération de code. La section suivante présente les optimisations qui peuvent être faites au cours de l'étape de modélisation.

2.2.3 Optimisations des modèles

L'optimisation de modèles est une transformation de modèle qui prend en entrée un modèle non optimisé et produit en sortie un modèle optimisé en préservant le comportement du modèle de départ. Contrairement à la génération de code (classifiée selon [36] dans la catégorie des transformations de modèles verticales et exogènes, section 2.2), l'optimisation de modèles est une transformation de modèles *endogène* et *horizontale*. La classification de [36] considère l'optimisation et le *refactoring* [71] comme deux transformations de

modèles différentes dans le sens où l'optimisation tend à améliorer les caractéristiques opérationnelles du logiciel (les performances) sans se soucier de la lisibilité du code alors que le *refactoring* tend à améliorer les qualités du logiciel en termes de lisibilité de code, de réutilisabilité et de maintenabilité. Selon [72] maintenabilité et performance ne vont pas ensemble. Il affirme qu'un code maintenable et lisible est rarement performant. Cependant, plusieurs outils de *refactoring* de modèles tel qu'EMF Refactor [73] et l'outil de *refactoring* du logiciel Telelogic TAU [74] peuvent améliorer les performances du code généré à partir des modèles. EMF Refactor est basé sur Eclipse EMF pour le *refactoring* des classes et des machines à états UML. Il transforme les machines à états de façon à éliminer les états isolés, les transitions redondantes, factoriser les mêmes transitions sortantes des états fils en une seule transition sortante de l'état parent, etc. Ces transformations améliorent la qualité du code généré à partir des machines à états de manière à produire un code plus compact.

Dans cette thèse nous considérons les modèles UML exécutables formés des classes, des activités et des machines à états UML. Ces dernières (les machines à états) sont les plus utilisées pour des fins d'optimisation. En effet, les optimisations de minimisation d'automates finis [75] tel que l'élimination des états inatteignables sont applicables pour les machines à états UML. Cependant, l'élimination d'un état inatteignable dans une machine à états UML ne se limite pas aux cas des états isolés, ou ceux qui n'ont pas de transitions entrantes. En effet, un état ayant une seule transition entrante dont la garde est toujours évaluée à faux est un état inatteignable. Sachant que la valeur d'une garde peut dépendre de la valeur d'une variable, des techniques d'exécution symbolique [76] seront utiles pour détecter les états inatteignables. Par exemple, l'outil Diversity [8] (développé au sein de l'équipe LISE) est capable de produire un graphe de couverture des états d'un automate et par la suite la liste des états non couverts.

La plupart des modeleurs UML assurent l'intégrité de leurs modèles en vérifiant qu'ils respectent bien les contraintes exprimées dans la norme UML [4]. Ces outils de validation peuvent être étendus pour vérifier le respect d'autres contraintes ajoutées par l'utilisateur, telles que la contrainte qui vérifie que tous les états d'une machine à états UML sont atteignables. Ainsi, la validation de modèles peut résulter à un modèle optimisé. Cependant, les outils et techniques de validation des modèles UML intégrés actuellement dans les modeleurs UML qui assurent une génération automatique de code exécutable tels que Rhapsody, BridePoint et iUML ne permettent pas de produire un modèle optimisé. La détection des états inatteignables ayant des transitions entrantes n'est pas assurée par ces outils.

2.2.4 Bilan et Positionnement

Dans cette section, nous avons présenté l'évolution du pouvoir d'optimisation dans l'approche classique basée sur l'IDM pour le développement des systèmes embarqués. Cette approche se base généralement sur les optimisations du compilateur. Nous avons étudié les niveaux d'optimisation du compilateur GCC (le niveau SSA et le niveau RTL) et nous avons vu que malgré l'évolution du pouvoir d'optimisation du GCC, les développeurs ne sont pas satisfaits par la performance du code exécutable produit à partir des modèles UML. Ils modifient souvent manuellement le code C++ produit par les générateurs de code pour l'optimiser. Cette modification manuelle du code élargit le *gap* entre le modèle et le

code et affecte les validations et les analyses qui peuvent être faites au niveau modèle. Nous avons également présentées deux techniques d'optimisation au cours de la génération de code : la modification des règles de génération de code (adoptée par le générateur de code BridgePoint) et l'ajout d'une forme intermédiaire (adoptée par le générateur de code RIALTO et l'approche Polis). Les optimisations au cours de la modélisation sont souvent assurées par les outils de *refactoring* et de validation. Cependant, le *refactoring* de modèles concerne généralement la problématique de lisibilité et de maintenabilité (et non pas d'optimisation).

Nous allons montrer dans ce travail de recherche que pour produire un code binaire optimisé en termes de taille de code à partir des modèles des systèmes embarqués, des passes d'optimisations sont exigées dans toutes les étapes de l'approche de développement de ces systèmes. Nous allons également montrer que la génération de code, bien qu'elle paraisse une étape essentielle dans le développement des systèmes, n'est pas utile pour les optimisations [77] au contraire, elle est la cause de la perte d'informations (utiles pour les optimisations) liées à la sémantique du langage UML. Nous proposons ainsi une nouvelle mise en œuvre de l'approche dirigée par les modèles pour le développement des systèmes embarqués qui élimine l'étape de la génération de code. Cependant, nous ne sommes pas les premiers à essayer de compiler directement les modèles UML. En effet, plusieurs travaux tels que [78] et [79] se sont intéressés à la construction de machines virtuelles UML qui ne produisent pas de code de 3ème génération à partir des modèles UML mais génèrent à la place un code intermédiaire (*bytecode*) interprété par la machine virtuelle pour produire le code binaire. Toutefois, la recherche que nous avons menée sur les machines virtuelles UML [80] et [81] a montré que ces outils sont souvent conçus pour simuler les modèles et en particulier les activités UML et par la suite ne s'intéressent pas à l'optimisation du code exécutable. La seule machine virtuelle UML permettant de compiler à la fois les activités et les machines à états est la machine virtuelle UML (UVM) proposée par Schattowsky et Muller [82]. Cette machine virtuelle UML n'est pas disponible et ne s'intéresse pas à la production du code compact à partir des modèles UML. Par conséquent, nous allons nous positionner uniquement par rapport aux travaux de génération de code.

Le tableau 2.3 présente 4 outils de génération de code sur les quelles se base l'approche existante dirigée par les modèles pour le développement des systèmes embarqués. Ils sont classifiés selon les critères suivants : les diagrammes UML comportementaux pris en compte, le langage cible produit et les optimisations supportées.

Critères de classification	Rhapsody	BridgePoint + iUML	RIALTO
Diagrammes comportementaux UML supportés	Activités, Machine à états, Séquence	Machine à états	Machine à états
Code cible	C/C++/Java/Ada	C/C++/Ada/	C++/ Vhdl
Techniques d'optimisation autres que les optimisations du GCC	--	Archetyes	S-Graph

Tableau 2.3 – Classification des outils de génération de code étudiés

Les générateurs de code profitent des optimisations des compilateurs de code mais favorisent aussi la modification manuelle du code généré par les développeurs en produisant un code lisible que le développeur maitrise (3ème ligne du Tableau 2.3). En compilant directement les modèles UML, notre approche préserve les fondements de l'IDM qui considère le modèle comme la seule représentation du système qui peut être modifiée, empêchant ainsi la modification manuelle du code qui affecte les validations et les analyses faites au niveau modèles.

Le compilateur de modèle que nous proposons compile directement les modèles UML tout en profitant des optimisations des compilateurs de code. Il permet aussi d'améliorer quelques unes. Plusieurs générateurs de code commerciaux tel que Rhapsody (deuxième colonne du Tableau 2.3) s'intéressent plutôt à la production d'un code lisible et maintenable et ne favorise pas les optimisations. D'autres générateurs de code se basent sur des techniques d'optimisation telle que la modification des règles de génération de code (Archetype) adopté par BridgePoint et iUML (troisième colonne du Tableau 2.3). Nous avons vu que cette technique, bien qu'elle implique une interaction avec l'utilisateur, exige une maitrise des patterns de génération de code et leurs impacts sur la performance du code binaire généré. Nous allons montrer dans cette thèse que les optimisations implémentés dans la forme intermédiaire S-Graph du générateur de code RIALTO (dernière colonne du Tableau 2.3) peuvent être améliorées.

2.3 Conclusion

Dans la première section de ce chapitre, nous avons présenté l'approche classique basée sur l'IDM pour le développement des systèmes embarqués. Nous avons évoqué le problème des générateurs de code qui n'arrivent pas à faire un *mapping* direct entre les éléments des modèles exécutables (en particuliers les machines à états UML) et les concepts des langages de programmation. Les patterns de génération de code utilisés comme solution à ce problème produisent un code lisible et utilisable mais ceci en détriment des performances du code binaire final. Pour produire un code binaire plus performant et plus compact, cette approche se base généralement sur les optimisations des compilateurs de code. Nous avons présenté dans la deuxième section de ce chapitre les optimisations du compilateur GCC. Malgré l'évolution du pouvoir d'optimisation du GCC, les développeurs ne sont pas satisfaits par la performance du code exécutable produit à partir des modèles UML. Ils modifient souvent manuellement le code C++ produit par les générateurs de code pour l'optimiser. Cette modification manuelle du code affecte les validations et les analyses qui peuvent être faites au niveau modèle. La génération de code représente aussi la cause de la perte d'informations liées à la sémantique du langage UML. Certaines de ses informations sont utiles pour les optimisations mais sont aussi invisibles par le compilateur qui n'est pas capable de les exploiter pour des fins d'optimisations. Tous ces problèmes liées à la génération de code nous ont mené à proposer une nouvelle mise en œuvre de l'approche dirigée par les modèles pour le développement des systèmes embarqués qui élimine l'étape de la génération de code. Concrètement, nous proposons le premier compilateur de modèles UML exécutables qui profite des optimisations des compilateurs de code et améliore quelques unes. Le chapitre suivant présentera en détail le compilateur de modèle proposé.

CHAPITRE 3

GUML un compilateur de modèles UML

Ce chapitre présente GUML (Gnu UML), le compilateur de modèles UML proposé. La première section définit le langage source du compilateur. Comme pour le cas des langages de programmation, le langage cible de notre compilateur de modèles est le langage binaire. La deuxième section en définit l'architecture. Un exemple d'utilisation du compilateur GUML est présenté dans la troisième et dernière section.

3.1 Langage source du GUML

D'après Harel et Rumpe [83], définir un langage revient à bien identifier sa syntaxe (ses éléments syntaxiques et les relations entre eux) et la sémantique de chaque élément syntaxique. Pour le langage UML, la syntaxe est bien définie par le méta modèle UML qui regroupe les différents éléments syntaxiques et les relations entre ces éléments. Ce méta modèle est doté de contraintes OCL [84] exprimant la sémantique statique (de bonne construction) des modèles UML conformes à ce méta modèle. Avec l'apparition des nouveaux standards fUML et ALF définissant d'une part une sémantique et d'autre part une syntaxe textuelle pour un sous-ensemble exécutable du langage UML, ce dernier peut être considéré comme un langage de programmation compilable en langage machine. Nous considérons alors, dans un premier temps, fUML comme étant le langage source de notre compilateur de modèle et ceci pour les raisons suivantes :

- fUML est le sous ensemble exécutable d'UML dont la sémantique d'exécution est bien définie.
- fUML est le sous ensemble minimal d'UML avec lequel nous pouvons exprimer la sémantique d'exécution de n'importe quel autre élément syntaxique UML. Par exemple, la sémantique d'exécution des machines à états UML peut être exprimée en fUML (en utilisant les classes et les actions).
- fUML (2 diagrammes, 29 actions) est plus compact que UML dans son ensemble (13 diagrammes, 45 actions), il s'avère alors judicieux de commencer par compiler ce noyau exécutable de UML.

Par la suite, nous détaillerons la syntaxe et la sémantique du langage source (basé sur fUML) du compilateur GUML.

3.1.1 Syntaxe du langage source

fUML considère uniquement deux diagrammes UML : un diagramme structurel (le diagramme de classes) et un diagramme comportemental (le diagramme d'activité et les actions). Bien que notre compilateur de modèle prenne en compte une grande partie des éléments syntaxiques de fUML, certains éléments syntaxiques présents dans la norme fUML tels que les *Structured Activities*, le *fork* et le *join* ne sont pas pris en compte dans cette première version du compilateur qui ne traite pas les concepts de haut niveau liés au parallélisme. Néanmoins, notre langage source est plus riche que le langage fUML dans le sens où il traite un autre diagramme comportemental d'UML : le diagramme des machines à états. En effet, les machines à états UML sont très utilisées pour la modélisation de la partie contrôle des systèmes embarqués. De plus, comme cela sera démontré dans les deux chapitres 4 et 5), dans le processus de l'exécution des machines à états que ce soit par la génération de code ou par la compilation directe, les éléments syntaxiques et sémantiques des machines à états représentent une source importante d'optimisation. Étant donné que l'objectif de la thèse est d'améliorer le processus d'optimisation dans une approche dirigée par les modèles pour le développement des systèmes embarqués et que ces diagrammes sont très utilisés dans le développement de ces systèmes, le choix d'inclure le diagramme des machines à états dans notre langage source a été fait. La sémantique d'exécution des machines à états reste bien définie puisqu'elle peut être exprimée en fUML. Toutefois, la

version actuelle de GUML ne compile que les machines à états UML simples. Les états composites, orthogonaux et les pseudo-états ne se sont pas pris en compte dans cette première version de GUML.

Nous avons vu dans le chapitre précédant (section 2.1.1.1) que la norme UML ne spécifie pas la syntaxe concrète des actions. Le standard fUML, avec la définition d'une bibliothèque de modèle pour la modélisation des comportements primitifs sur les types primitifs tels que l'addition des entiers, la concaténation des String, etc. a contribué à la définition de cette syntaxe concrète manquante dans UML. La figure 3.1(a) montre une utilisation de la fonction «+» de la bibliothèque de fUML pour la spécification du corps de l'opération `MyOperation() { MyAttribute=MyAttribute+1}`. Pour spécifier le corps des opérations les plus compliqués, l'OMG a standardisé le langage d'action ALF. La représentation textuelle ALF du corps de l'opération `MyOperation()` est présenté par la figure 3.1(b).

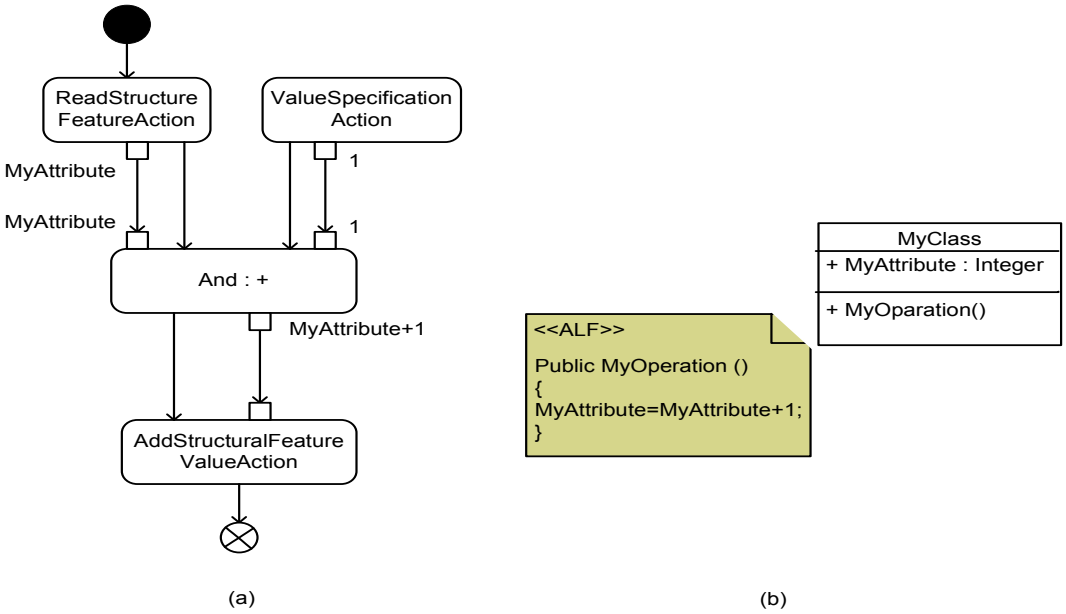


FIGURE 3.1 – La modélisation de l'instruction `MyAttribute=MyAttribute+1` en utilisant UML : (a) notation graphique en utilisant les actions UML (b) notation textuelle de ALF

La modélisation du comportement de `MyOperation()` en ALF revient à ajouter un *commentaire* à l'opération annoté par le mot clé « ALF » et dans lequel le corps de l'opération est spécifié avec la syntaxe de ALF : «`MyAttribute=MyAttribute+1;`». Nous pouvons remarquer que la norme ALF simplifie considérablement la modélisation des corps des opérations. Cependant, ce langage relativement récent n'est pas encore supporté par les modeleurs UML. Nous avons ainsi choisi, dans un premier temps, de modéliser les corps des opérations en utilisant la bibliothèque définie par fUML.

3.1.2 Sémantique du langage source

Bien que fUML définisse formellement la sémantique d'exécution de tous ses éléments syntaxiques, celui-ci ne fixe pas *les points de variation sémantique* introduits explicitement par la norme UML. Par conséquent, afin de bien définir la sémantique de notre langage source,

nous avons fixé les points de variation sémantique qui sont étroitement liés à la sémantique d'exécution de notre modèle. Nous ne fixons que les points de variation sémantique utiles pour notre langage source [1].

A titre d'exemple, UML ne définit pas la sémantique d'exécution de l'action *CallOperationAction* si la pré-condition définie pour l'opération à exécuter est fausse. Ce point de variation sémantique n'est pas fixé dans notre langage source vu que nous ne compilons pas les pré/post-conditions définies dans les modèles UML. Un des points de variation sémantique fixé pour notre langage concerne la sémantique d'envoi de signal (*SendSignalAction*) à une classe qui déclare la réception de ce type de signal (*Reception*). En effet, la norme UML ne définit ni le protocole d'envoi du signal, ni le temps que celui-ci prend pour arriver à l'objet receveur. Pour notre cas, et pour simplifier la compilation de notre modèle, nous supposons que le temps mis par le signal pour arriver à l'objet receveur est nul. Nous modélisons ainsi l'envoi d'un signal et sa réception par un appel à une opération (*signal_operation()*) ajoutée à l'objet receveur `O` et qui prend comme paramètre un signal (*Sig*). Ainsi, l'envoi du signal est représenté par l'appel (dans l'opération *main*) de l'opération `O.signal_operation(Sig)`. Ceci correspond également à la réception du signal par l'objet `O`. Par la suite, le temps mis par le signal pour arriver à l'objet `O` est nul.

La figure 3.2 illustre un autre exemple de point de variation sémantique liée à la gestion des événements (de type *SignalEvent*) dans une machine à états UML. Vu que l'envoi d'un signal est toujours *asynchrone*, une file d'attente doit être associée à l'objet receveur. Cependant, la sémantique d'UML ne précise pas la politique de gestion associée à cette file de signaux. Une politique FIFO consiste à retirer les signaux de la file dans leur ordre d'arrivée. Il est également possible de traiter les signaux selon leur ancienneté, en sélectionnant le plus récent (politique LIFO) ou encore établir un ordre de priorité entre les signaux qui surviennent. Ces différents critères conditionnent le comportement et les réactions du système.

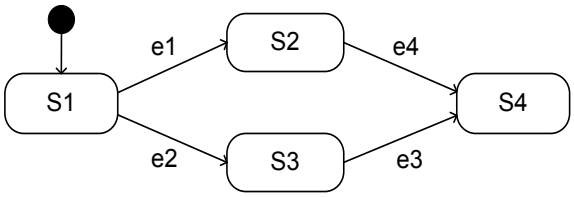


FIGURE 3.2 – Un exemple illustratif d'une machine à états UML dont le comportement dépend de la politique de gestion des signaux (le système peut évoluer vers l'état S2 (politique FIFO) ou bien S3 (politique LIFO) si la file d'attente contient les deux signaux {e1 ; e2} ordonnés selon leurs ordre d'arriver)

Dans le cas de la figure 3.2, nous considérons que les signaux (e1, e2) surviennent à l'étape t. Si ces événements sont retirés de la file dans leur ordre d'arrivée (FIFO), le système évolue vers l'état S2. Par contre, si la file de signaux est utilisée comme une pile (LIFO), le système évolue alors vers l'état S3. Pour simplifier la compilation de notre modèle UML, nous supposons qu'un objet ne peut recevoir qu'un seul signal à la fois (la taille de la file d'attente est fixée à 1). Ainsi le point de variation sémantique de la figure 3.2 ne représente

1. La liste de tous les points de variation sémantique que nous avons fixés pour bien définir la sémantique d'exécution de notre langage se trouve dans l'Annexe B.

plus une source d'ambiguïté dans l'exécution de notre modèle.

La figure 3.3 résume les différentes composantes de notre langage source qui garantissent un langage bien défini syntaxiquement : sous ensemble du méta modèle UML (CD : le diagramme de classes, AD : le diagramme d'activités et SMD : le diagramme de machines à états). Ce langage est également bien défini sémantiquement grâce à la sémantique d'exécution de fUML.

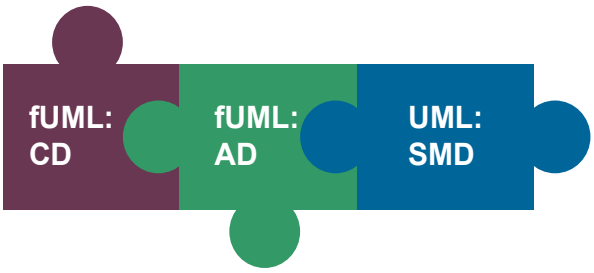


FIGURE 3.3 – Les composantes du langage source de GUML

La section suivante présente la construction du compilateur, à savoir la définition de son architecture qui regroupe les différentes phases et activités du compilateur.

3.2 Architecture de GUML

Étant donné que l'objectif de ce travail de recherche est la construction d'un compilateur de modèle aussi performant que les compilateurs de code, une étude approfondie du compilateur GCC est utile, surtout que nous aurons certainement besoin de réutiliser certaines passes de la phase de synthèse : à titre d'exemple, le passage de l'assembleur vers le binaire est une passe que notre compilateur de modèle devra effectuer. Comme GCC offre plusieurs passes d'optimisation (section 2.2.1), il est important que notre compilateur de modèle puisse bénéficier de ces optimisations ainsi que de l'évolution des optimisations dans les versions à venir de GCC.

3.2.1 Réutilisation de l'architecture du compilateur GCC

En réalité, et comme son nom l'indique, GCC (Gnu Compiler Collection) n'est pas un seul compilateur, mais un ensemble de compilateurs de plusieurs langages de programmation tel que le C, C++, Java, Ada et autres. Cependant, ces différents compilateurs partagent la même architecture. La figure 3.4 présente l'architecture commune aux compilateurs C, C++ et Java du compilateur GCC.

Cette figure montre que ces 3 compilateurs partagent la même partie *back-end* (pour une plateforme d'exécution bien définie) et la même partie middle-end. Cela signifie que les transformations de GENERIC $\rightarrow$ GIMPLE $\rightarrow$ SSA $\rightarrow$ RTL $\rightarrow$ Assembleur $\rightarrow$ Binaire sont implémentées une seule fois et sont réutilisées par la suite pour tous les compilateurs de GCC (pour une plateforme donnée). De la même manière, les environs 200 passes d'optimisation du compilateur GCC sont implémentées une seule fois (que se soit les optimisations du niveau SSA ou celles du niveau RTL) et chaque compilateur du langage

source choisi bénéficiera du résultat de ces optimisations. Cela signifie qu'ajouter un nouveau compilateur à la collection GCC se résume à faire une seule transformation de code : du langage d'entrée choisi → GENERIC. Ainsi, les compilateurs C, C++, Java et autres de la collection GCC ne sont que des *front-ends* GCC. La liste des *front-ends* pour le compilateur GCC ne se limite pas à ces 3 *front-ends*. En effet, la dernière version du GCC est distribuée avec 8 *front-ends* pour les langages impératifs suivants : C, C++, Java, Ada, Fortran, Objective C et Objective C++ et Go[2]. D'autres *front-ends* pour d'autres types de langages sont en cours de développement comme le GHDL (le *front-end* GCC pour le langage de *description de machine* VHDL), le GCCPY (le *front-end* GCC pour le langage *fonctionnel* Python) et le *front-end* GCC pour le langage Mercury, un langage *fonctionnel déclaratif* proche du Prolog.

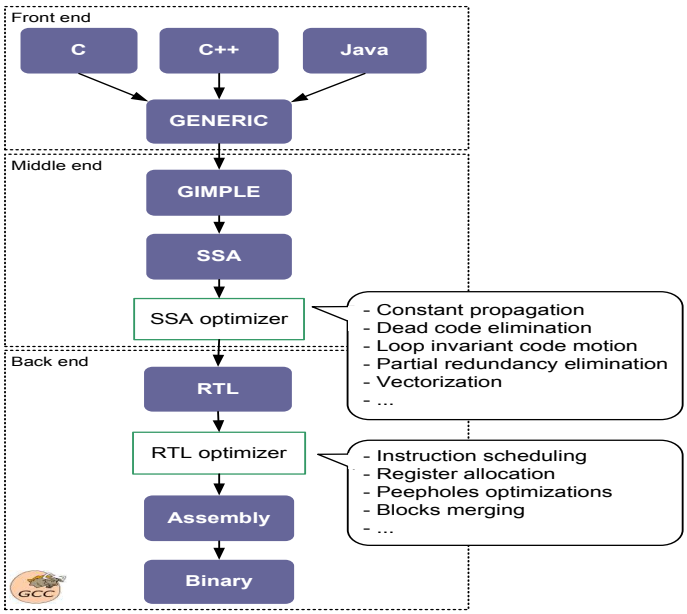


FIGURE 3.4 – Architecture du Compilateur GCC

La structure modulaire du compilateur GCC lui a donc permis de compiler plusieurs types de langages : les langages impératifs, déclaratifs, fonctionnels et même les langages de description de machine. Une question se pose alors : l'architecture de la collection de compilateurs GCC est-elle adaptée pour compiler un nouveau type de langage : le langage de modélisation UML ? Étant donné que GCC compile déjà le langage C++ et que la génération du code C++ à partir des modèles fUML est complètement automatisable, une compilation directe des modèles fUML avec le compilateur GCC serait possible. Cela consiste à étendre le compilateur GCC en ajoutant un nouveau *front-end* : le *front-end* GCC pour le langage UML (GUML). Avant de passer à l'implémentation de GUML, la section suivante présente d'une manière plus précise les deux formes intermédiaires GENERIC et GIMPLE.

2. Langage de programmation crée par Google en 2009, http ://golang.org/

3.2.2 GENERIC vs GIMPLE

GENERIC, comme son nom l'indique, est une représentation générique de l'arbre de syntaxe abstraite (*Abstract Syntax Tree* (AST)) générée par les *front-ends* GCC à l'issue de l'étape de l'analyse du code source. Le fait qu'elle soit commune à tous les *front-ends*, tous les sucres syntaxiques des langages de programmation tels que la conversion implicite de type (*casting*), l'appel implicite au constructeur, etc. sont désormais explicites. La forme GENERIC est une manière de représenter les fonctions sous forme d'arbres. N'importe quel élément du langage source (un type, une variable, une expression, ...) est représenté par un arbre.

Chaque nœud de cet arbre possède un TREE_CODE associé qui définit le type de l'élément représenté par le nœud. Par exemple le type entier est représenté par un arbre dont le TREE_CODE est INTEGER_TYPE, l'expression de l'addition est représentée par le TREE_CODE PLUS_EXPR, etc. L'ensemble de tout les TREE_CODES définis par GCC se trouvent dans le fichier *gcc.x.y.z/gcc /tree.def* (*gcc.x.y.z* est le répertoire racine du compilateur GCC). La construction de la forme GENERIC est implémentée dans le fichier *gcc.x.y.z/gcc/tree.c*. Le type de base *tree* représente un pointeur vers une union *tree_node* comme suit :

```
typedef union tree_node *tree
```

Pour construire un arbre, le fichier *tree.h* définit des fonctions pour chaque type. Par exemple la fonction *build_decl* pour la construction d'un arbre qui représente une déclaration de variable, la fonction *build_call_exp* pour un appel de fonction et ainsi de suite. La construction de la forme GENERIC équivalente à la déclaration d'une variable nommée x de type entier est la suivante :

```
tree mon_type = make_node (INTEGER_TYPE);
tree mon_id= get_identifier("x");
tree decl_de_var = build_decl (BUILTINS_LOCATION, VAR_DECL, mon_id,
mon_type);
```

La figure 3.5 montre la représentation graphique de la déclaration de la variable x en GENERIC. DELC_NAME et TREE_TYPE sont des macros définies pour accéder respectivement à l'identificateur et le type de la variable.

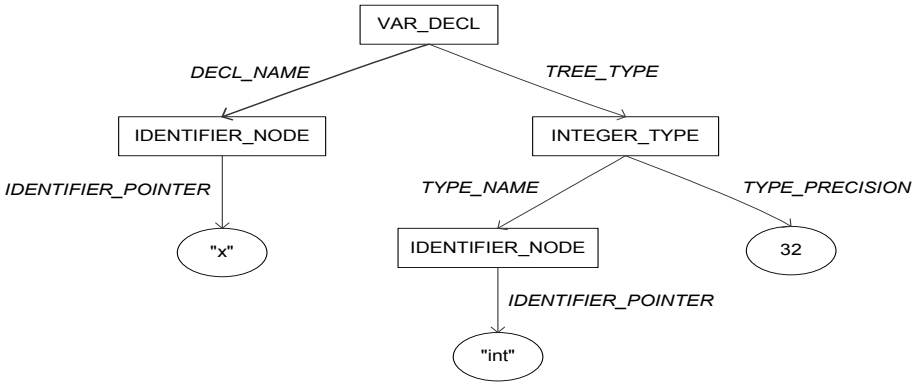


FIGURE 3.5 – La représentation de "int x" en GENERIC

Si GENERIC est une forme simplifiée du langage source, GIMPLE est une forme GENERIC simplifiée et donc une plus simplifiée du langage source (section 2.2.1.1). GIMPLE n'autorise pas les boucles (*for* et *while*) et utilise à la place des *gotos* et des *labels*. La facilité de la construction du graphe de flot de contrôle (GFC) à partir de la forme GIMPLE ainsi que la grammaire compacte de cette forme par rapport à celle de GENERIC, font de GIMPLE la forme intermédiaire de GCC la plus adéquate pour les optimisations de haut niveau (ces optimisations reposent toutes sur le GFC (Figure 2.15). La figure 3.6 présente un programme GENERIC (qui calcule la somme des 10 premiers entiers) et son équivalent en GIMPLE dans une syntaxe proche de celle de C. La représentation du même programme en utilisant les instructions GIMPLE (équivalentes au TREE_CODE pour GENERIC) est aussi présentée par la même figure.

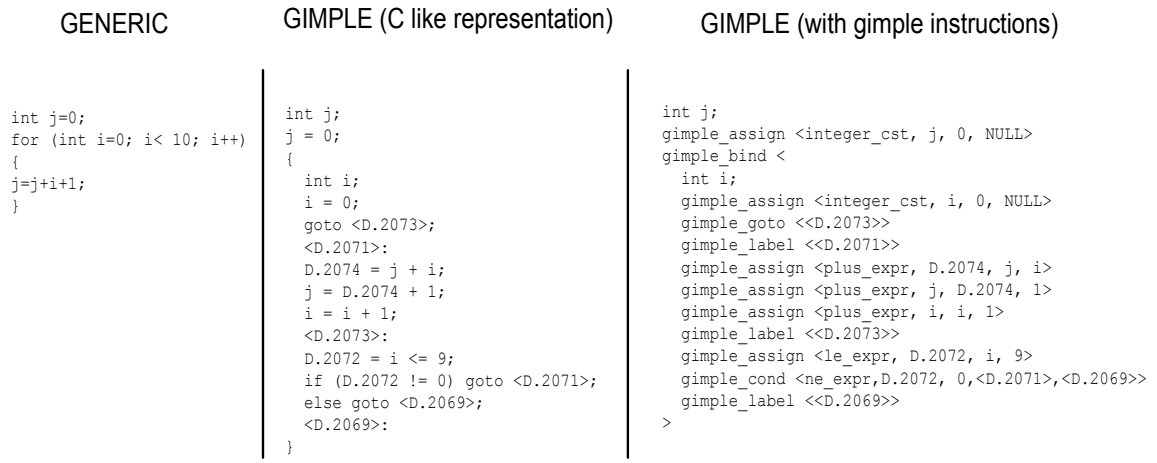


FIGURE 3.6 – GENERIC vs GIMPLE

Vu que GIMPLE n'est qu'une forme simplifiée de GENERIC (bien qu'au niveau de l'implémentation dans GCC, GENERIC se base sur la structure de donnée *arbre* alors que GIMPLE se base sur la notion de *tuple*), certains *front-ends* génèrent directement la forme GIMPLE. C'est le cas des deux *front-end* C et C++. En effet, la structure modulaire de GCC offre un certain degré de liberté au développeur qui peut transformer son langage source vers n'importe quelle autre forme intermédiaire (différente de GENERIC). Cependant, il serait alors nécessaire d'implémenter la transformation de cette forme intermédiaire choisie vers GIMPLE afin de pouvoir bénéficier des optimisations de GCC. Pour éviter ce travail fastidieux (la fonction de *gimplification* de GENERIC -très proche de GIMPLE- est très complexe : 8000 ligne de code), et pour encourager les développeurs à ajouter des nouveaux *front-ends* au compilateur GCC, un mécanisme de définition de nouveaux TREE_CODES GENERIC a été implémenté dans GCC. Dans ce cas, si le langage source contient des éléments qui ne peuvent pas être présentés par les TREE_CODES définis par GCC, le développeur peut définir d'autres TREE_CODES (dans le fichier *tree-def* de son *front-end*) et ceci en utilisant la macro DEFTREECODE comme suit :

```
DEFTREECODE (name, string_id, class_type, n_args)
```

Cependant, la transformation des nouveaux TREE_CODES vers la forme GIMPLE reste à la charge du développeur. A titre d'exemple, le *front-end* C++ définit le TREE_CODE *template_info* pour stocker des informations liées aux *templates*.

```
DEFTREECODE(TEMPLATE_INFO,"template_info", tcc_exceptional, 0)
```

Pour GUML, nous avons fait le choix de générer directement la forme GIMPLE à partir des modèles UML en utilisant les TREE_CODES définis par GCC. Cependant, une extension des TREE_CODES pour les diagrammes de machine à états et notamment pour les concepts *state* et *transition* est envisageable.

La section suivante présente l'implémentation du *front-end* GUML, allant de la phase d'analyse du langage source vers la génération de la forme GIMPLE.

3.3 Implémentation du GUML

La figure 3.7 présente l'architecture de GUML qui réutilise les *middle* et *back-ends* du compilateur GCC. Cette figure détaille uniquement la partie frontale du compilateur de modèle. Le *front-end* C++ est aussi présenté dans la partie frontale de GUML. En effet, la structure de GUML a été largement inspirée de G++ (le *front-end* C++). La majeure différence se situe au niveau de l'implémentation des deux *front-ends* et des outils utilisés surtout dans la phase d'analyse, phase qui précède la génération de la forme GIMPLE.

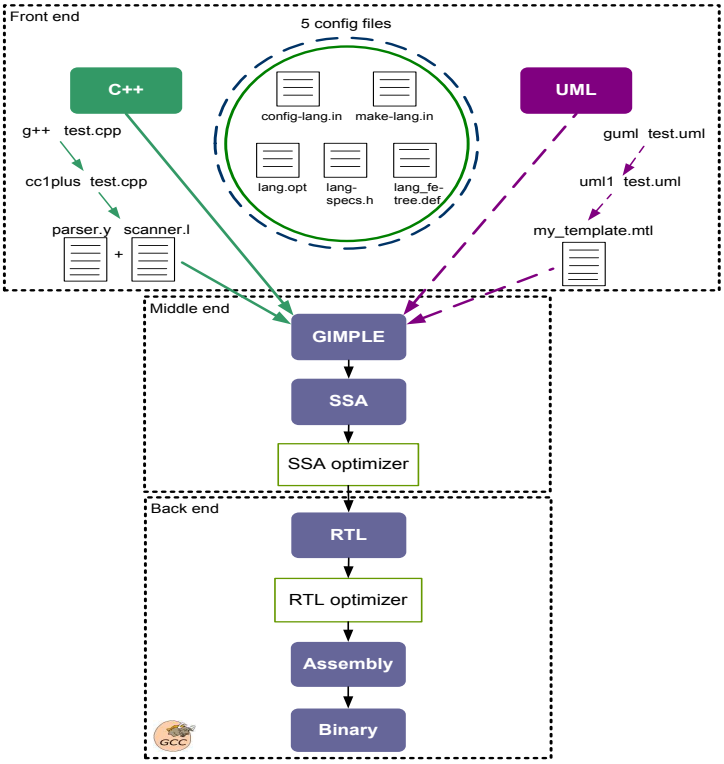


FIGURE 3.7 – Architecture de GUML : un *front-end* UML pour le compilateur GCC

3.3.1 Phase d'analyse

La phase d'analyse du code source est la première phase qu'un compilateur est censé faire. Pour les langages source à forme textuelle, cette phase se décompose généralement en 3 passes : une analyse lexicale, une analyse syntaxique et une analyse sémantique. Selon

[85], si le programme avait été composé avec un outil ne permettant de composer que des programmes du langage source, l'analyse lexicale et syntaxique du programme parait moins cruciale, c'est le cas du langage de modélisation UML. En effet, les modeleurs UML mettent à la disposition du concepteur les éléments syntaxiques du langage UML pour construire son modèle. Cependant, ces outils (y compris Papyrus [3], le modeleur UML développé au sein de l'équipe LISE) n'autorisent que la construction des modèles conformes au méta modèle UML. À titre d'exemple, dans un diagramme de machine à états UML, une transition doit avoir un état source et un état cible, Papyrus ne permet l'ajout d'une transition que si l'état source et cible ont été déjà définis. Par contre, pour les langages de programmation, le développeur est complètement libre dans l'écriture de son programme, il pourra à titre d'exemple écrire *int x* (sans le point virgule) et c'est uniquement au cours de l'analyse sémantique que le compilateur de code va détecter cette erreur syntaxique.

Ainsi, l'implémentation des modeleurs UML gèrent en grande partie l'analyse lexicale et syntaxique d'un modèle UML, bien que certaines règles de bonne construction de modèles (liées à l'utilisation des profils UML, etc) puissent ne pas être implémentées dans le modeleur [86]. Dans ce cas, les modeleurs UML peuvent être étendus par des outils dédiés à la validation des modèles qui analysent d'avantage la syntaxe et même la sémantique des modèles . Papyrus étant un plugin de l'environnement Eclipse [4], utilise le plugin EMF Validation [5] pour l'analyse des modèles à compiler.

Il est à noter que dans l'implémentation de GUML, nous nous sommes plutôt intéressés à la génération de la forme GIMPLE à partir de notre modèle de départ et nous supposons que les analyses faites par Papyrus sont suffisantes. Cependant, le domaine de validation des modèles UML que ce soit vis-à-vis de leur sémantique statique (la sémantique de bonne construction) ou de leur sémantique d'exécution est un domaine de recherche très actif où plusieurs techniques de validation sont proposées telles que le model checking [87]. La section suivante détaille l'implémentation concrète de GUML (passage d'UML vers GIMPLE) en se basant sur celle de G++ (passage de C++ vers GIMPLE).

3.3.2 Génération de GIMPLE

Concrètement, ajouter un *front-end* au compilateur GCC consiste à ajouter un répertoire dans le dossier *gcc* de la racine *gcc-x.y.z*. Ce nouveau répertoire porte généralement le nom du langage source à compiler. La figure 3.8 présente une partie du contenu du répertoire *gcc.x.y.z/gcc* étendu : les répertoires cp, java et uml correspondent respectivement aux 3 *front-ends* G++ (pour le langage C++), GCJ (pour le langage Java) et GUML (pour le langage UML).

Ce répertoire contient l'implémentation réelle de chaque *front-end* et les fichiers de configuration qui assurent la prise en compte du nouveau langage et ses spécificités par le compilateur GCC (Figure 3.7). Quelques informations utiles à la compréhension générale du fonctionnement du processus de l'extension du compilateur GCC (en particulier pour la compilation du langage UML) sont présentées par la suite.

3. http ://www.papyrusuml.org/
4. http ://www.eclipse.org/
5. http ://www.eclipse.org/modeling/emf/downloads/ ?project=validation

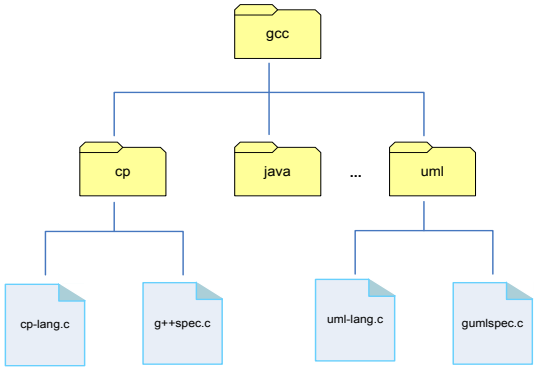


FIGURE 3.8 – Arborescence du répertoire gcc

3.3.2.1 Fichiers de configuration

Les cinq fichiers suivants doivent être configurés par le développeur d'un *front-end* GCC.

Config-lang.in : Ce fichier est une extension du fichier *configure* du compilateur GCC. C'est un script dans le quel est spécifié le nom du langage source à compiler, le fichier qui implémente réellement le *front-end*, les fichiers que génère le *front-end*, etc. Par exemple pour GUML le nom du langage est *uml* et le fichier qui implémente le compilateur est *uml1.c*.

Make-lang.in : Ce fichier représente une extension au *makefile* de GCC contenant le script de génération des fichiers exécutables. Pour GUML, ce fichier contient les commandes qui génèrent les deux exécutables *guml* et *uml1* (voir section 3.3.2.2).

Language-tree.def : C'est dans ce fichier de configuration que les nouvelles TREE_CODES qui forment la représentation GENERIC sont spécifiés. En effet, si un *front-end* assume que les TREE_CODES de GENERIC sont insuffisants pour exprimer la sémantique des éléments syntaxiques de son langage, il pourra ajouter de nouveaux TREE_CODES dans ce fichier. Pour GUML, ce fichier est momentanément vide.

Lang-specs.h : Ce fichier est une extension du fichier gcc.c qui représente l'implémentation du pilote GCC. Il contient des spécifications qui assurent la prise en compte du nouveau *front-end* : nouveau type de fichier à prendre en compte (*.uml pour GUML), etc.

Lang.opt : Ce fichier contient la liste des options (*flags*) que l'utilisateur pourra spécifier dans sa commande de compilation. Par exemple, l'appel de GUML pour compiler un modèle (*mymodel.uml*) est le suivant : *guml mymodel.uml*. Cependant, l'utilisateur pourra spécifier des options qui guideront la compilation de ce fichier : -S pour arrêter la compilation juste après la production du fichier assembleur (ne pas exécuter la transformation de l'assembleur vers le binaire et du binaire vers l'exécutable), -Os : pour optimiser la taille de l'exécutable, etc. Mise à part les 5 fichiers de configuration présentés ci dessus, le répertoire *uml* contient deux fichiers *guml_spec.c* et *uml1.c* qui représentent respectivement l'implémentation du pilote et du compilateur, sujets de la section suivante.

3.3.2.2 Compilateur vs Pilote

Pour compiler le fichier *test.cpp* en utilisant G++, un développeur tape la commande suivante *g++ test.cpp* ayant en tête que l'exécutable g++ représente le compilateur GCC pour les fichiers C++ et que cet exécutable se charge de l'analyse et la synthèse de son fichier en produisant à la fin le code binaire (l'exécutable). Cette constatation est fausse. En effet l'exécutable *g++* n'est que le Pilote du *front-end* G++. C'est un exécutable produit par GCC après la compilation du fichier *g++-spec.c*. Cet exécutable se charge uniquement de la gestion du processus de compilation. Par exemple, pour arrêter la compilation à l'étape de la génération de l'assembleur, on pourra passer l'option *-S* au pilote *g++* qui dans ce cas n'appellera pas le module *as* (qui produit *test.o*) ni le module *ld* (qui produit le *test.exe*). Le fonctionnement du pilote *g++* est décrit par la figure 3.9.

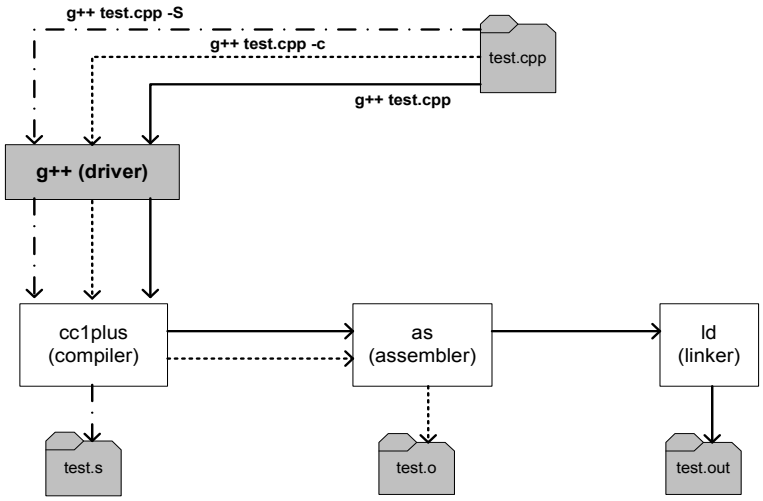


FIGURE 3.9 – Fonctionnement du Pilote "g++" : appelle le compilateur "cc1plus" puis l'assembleur "as" et finalement le linker "ld"

Comme illustré sur cette figure, l'exécutable qui compile réellement le programme source est *cc1plus*. Ce fichier est produit suite à la compilation du fichier source *cp-lang.c* (Figure 3.8). C'est dans ce fichier source que la phase d'analyse ainsi que celle de synthèse sont implémentées. Par analogie au compilateur G++, nous avons ainsi implémenté le pilote (*guml-spec.c* compilé en *guml* exécutable) et le compilateur (*uml-lang.c* compilé en *uml1* exécutable).

Depuis sa création, le *front-end* G++ fait appel aux outils Flex (pour générer l'analyseur lexicale) et Bison (pour générer l'analyseur syntaxique et sémantique)[88]. A partir de la version 3.4 du compilateur GCC et pour des rasions de lenteur et manque de fiabilité dans certains cas, G++ n'utilise plus l'outil Bison et assure désormais la phase d'analyse en utilisant un analyseur écrit à la main (*gcc.x.y.z/gcc/cp/cp-parser.c*). Suite à l'étape d'analyse sémantique, la forme GIMPLE est générée.

Disposant d'un modèle UML plus structuré que les fichiers textes (les programmes C++ possèdent une forme séquentielle) et sachant que l'analyse lexicale (comme cela a déjà été expliqué dans la section 3.3.1) n'est pas utile pour les modèles UML, nous avons choisi (contrairement à G++) d'assurer la phase de l'analyse de nos modèles et la production de la forme GIMPLE en utilisant une technologie de transformation modèle à

texte M2T (section 2.1.2). Nous avons choisi l'outil Acceleo. Cet outil est un plugin Eclipse implémentant la norme MOF2Text (*Model to Text Transformation Language*) de l'OMG. Cette norme décrit un langage de transformation de modèle basé sur les patrons (*templates*). Le principe d'un template est de fournir un code dont une partie sera complétée à partir du modèle source analysé. Un *template* Acceleo est un fichier texte contenant, du texte statique (indépendant du modèle source) et du code Acceleo qui est interprété selon le modèle source (dépendant du modèle source). A titre d'exemple, le *template* Acceleo de la figure 3.10 produit un code Java dont le nom de la classe varie selon le nom de la classe UML du fichier source.

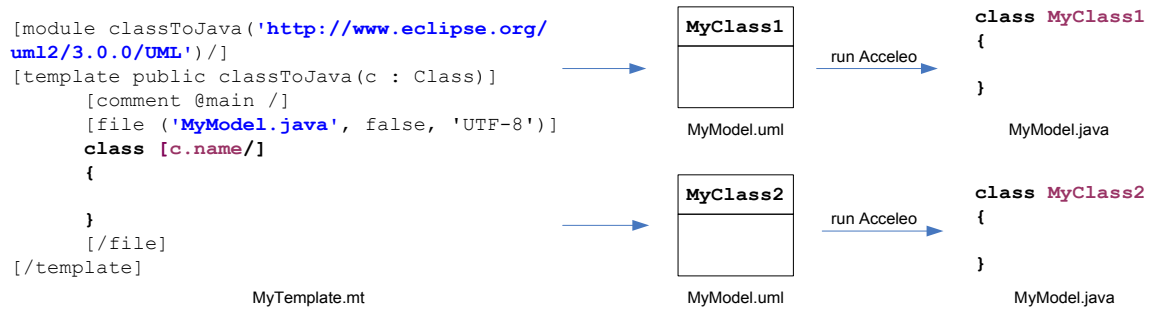


FIGURE 3.10 – Exemple d'un *template* Acceleo générant du Java à partir d'UML

L'outil Acceleo prend en entrée le template spécifié par l'utilisateur, le fichier source (dans notre cas le modèle UML), l'analyse et produit le fichier cible correspondant au *template* spécifié. Acceleo est souvent utilisé pour générer du code (Java, C/C++, etc.) à partir des modèles UML. Nous l'avons ainsi utilisé pour générer la forme intermédiaire GIMPLE. En réalité, le *template* Acceleo produit le fichier de construction des TREE_CODES (section 3.2.2) correspondants au modèle source (appelle aux fonctions prédéfinies par GCC comme *build_decl()*, etc). Ces TREE_CODES (éléments de base de la forme GENERIC) ne sont pas générés explicitement et sont passés comme paramètres de la méthode *gimplify()* du fichier source du compilateur GUML (uml1.c) et ceci pour produire directement la forme GIMPLE.

La partie frontale de la figure 3.7 détaille comment GUML (par analogie à G++) procède pour produire la forme GIMPLE à partir d'un modèle fUML représenté par le fichier *mymodel.uml* :

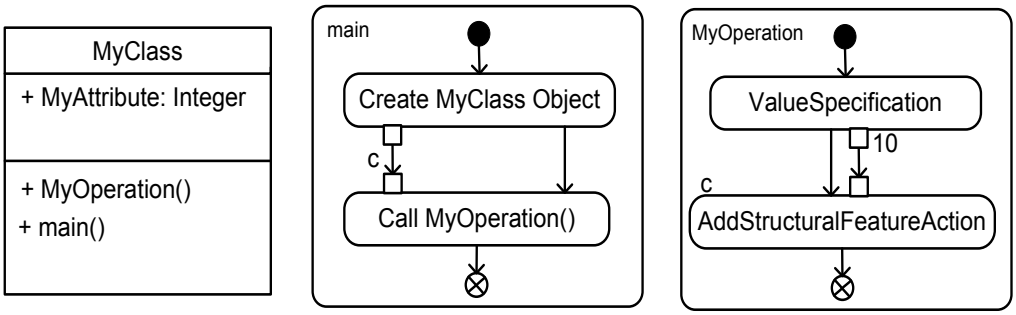
- L'utilisateur de GUML fait appel au pilote *guml* en passant le modèle UML en paramètre (*guml mymodel.uml*)
- Le pilote guml fait appel au compilateur *uml1*
- Le compilateur *uml1* fait appel à Acceleo pour analyser le modèle et construire les TREE_CODES correspondants
- Le compilateur *uml1* fait appel à la méthode *gimplify()* pour produire la forme GIMPLE

Une fois la forme GIMPLE générée, *uml1* fait appel au middle et *back-ends* du GCC pour produire le binaire optimisé. La commande *guml mymodel.uml -fdump-tree-gimple -S* génère le fichier *mymodel.uml.gimple* qui correspond à la forme GIMPLE ainsi que le fichier *mymodel.s* qui représente l'assembleur produit directement à partir du modèle. En éliminant l'option -S de la commande, *guml* produit la forme GIMPLE et l'exécutable du modèle *mymodel.exe*. Dans la section suivante, nous allons compiler (en utilisant GUML) un simple modèle fUML

contenant un diagramme de classes et deux diagrammes d'activité.

3.4 Exemple d'utilisation de GUML sur un modèle fUML

Pour illustrer la compilation directe des modèles fUML à l'aide de GUML, la figure 3.11 présente un exemple simple d'une classe (*MyClass*) possédant une propriété de type entier (*MyAttribute*) et deux opérations (*main()* et *MyOperation()*).




```
class MyClass{
public: void
MyOperation(){
MyAttribute=10;}
int myAttribute; };

int main(){
MyClass c ;
c.MyOperation ();
Return 0;}
```

(a) **g++** test.**cpp** -fdump-tree-gimple

```
int main() ()
gimple_bind <
int D.2074;
gimple_bind <
struct MyClass c;
gimple_call <MyOperation, NULL, &c>
gimple_assign <integer_cst, D.2074, 0, NULL>
gimple_return <D.2074>>
gimple_assign <integer_cst, D.2074, 0, NULL>
gimple_return <D.2074>>
Void   MyClass::MyOperation()(struct   MyClass
*const this)
gimple_bind <
gimple_assign<integer_cst,this-> MyAttribute,
10, NULL>>
```

```
main ()
gimple_bind <
<unnamed-signed:32> D.34;
struct MyClass c;
gimple_call <MyOperation, NULL, &c>
gimple_assign <integer_cst, D.34, 0, NULL>
gimple_return <D.34>>
MyOperation (struct MyClass * this)
gimple_bind <
gimple_assign<integer_cst, this->MyAttribute
, 10, NULL>
gimple_return <NULL>>
```

FIGURE 3.11 – Un modèle fUML (a) son code C++ et sa forme GIMPLE générée par G++ (b) sa forme GIMPLE générée directement par GUML

Le comportement de l'opération *main()* (création d'une instance de la classe *MyClass*

et l'appel de l'opération *MyOperation()*) est spécifié à l'aide d'un diagramme d'activité. Un autre diagramme d'activité est utilisé pour modéliser le comportement de l'opération *MyOperation()* qui consiste à modifier la valeur de *MyAttribute* (*MyAttribute=10*).

La figure 3.11 présente le modèle exécutable et les deux formes GIMPLE produites (a) en utilisant G++ après avoir généré du code C++ équivalent et (b) en utilisant GUML (sans génération de code). Nous pouvons remarquer que les deux formes GIMPLE produites que ce soit en compilant avec G++ ou GUML sont pratiquement équivalentes mises à part quelques différences dans la gestion des types. Les instructions GIMPLE les plus importantes pour ce modèle sont *gimple_assign*, *gimple_call* et *gimple_return*.

Les deux fichiers assembleurs générés par G++ et GUML ont la même taille (résultat attendu vu que les formes GIMPLE sont quasi identiques). Nous pouvons ainsi argumenter que GUML ne dégrade pas les performances du code produit par G++. Par la suite, utiliser GUML au lieu de G++ aura deux avantages : (1) gagner une étape de compilation : la génération de code C++ et (2) interdire complètement la modification à la main du code généré qui engendre plusieurs problèmes discutés dans l'introduction de cette thèse. En effet, aucune forme intermédiaire n'est accessible en écriture. Même si le développeur accède à la forme GIMPLE, il ne pourra pas la modifier car il doit tout d'abord maitriser la forme GIMPLE et ensuite maitriser les mécanismes d'extension de GCC pour qu'il prenne en compte les modifications apportées à GIMPLE.

3.5 Conclusion

Dans ce chapitre, nous avons détaillé l'implémentation du compilateur de modèle GUML. Nous avons fixé notre langage source (les classes, les activités et les machines à états). Nous avons également décrit l'architecture de GUML qui repose sur le *middle* et le *back-end* du compilateur GCC. La réutilisation du *middle-end* de GCC permet à GUML de tirer profit des optimisations de haut niveau du GCC (environ 130 passes SSA). La réutilisation du *back-end* quand à elle présente deux avantages à savoir la réutilisation des optimisations de bas niveau de GCC (environ 70 passes RTL) et la possibilité de compiler le modèle fUML vers 40 plateformes différentes ciblée par GCC tel que ARM, MIPS, x86, etc. Nous avons aussi décrit l'implémentation de GUML et son utilisation sur un exemple simple d'un modèle fUML (un diagramme de classe et deux diagrammes d'activité). Ainsi, nous avons constaté que GUML ne dégrade pas les performances du code exécutable obtenu si un même modèle est compilé avec G++ suite à une étape de génération de code C++ à partir du modèle source.

Toutefois, pour favoriser l'utilisation de GUML, ce résultat n'est pas suffisant, au contraire, il favorise le choix de G++. En effet, même si GUML élimine une étape dans l'approche dirigée par les modèles pour le développement des systèmes, G++ reste un compilateur mature et largement utilisé. De plus, l'intérêt de modifier les processus usuels de génération de code/compilation serait limité au non accès en écriture au code généré si l'exécutable final est de performance égal qu'il soit obtenu par G++ ou par GUML.

Dans le chapitre suivant, nous allons montrer que dans certains cas (compilation de machines à états), GUML produit un code exécutable plus compact que le code produit par G++ en compilant le même modèle source.

CHAPITRE 4

GUML un compilateur de modèle optimisant : premier niveau d'optimisation

Dans ce chapitre, nous montrons que GUML est capable de produire (en compilant certaines machines à états UML) un code assembleur plus compact que le code assembleur généré par G++. Pour cela, la première section de ce chapitre présente les limites du G++ en termes de production de code compact. La deuxième section étudie les différentes alternatives possibles pour combler ces limites. La troisième section présente et évalue la solution adoptée par GUML pour produire un code plus compact que celui produit par G++.

4.1 Études Expérimentales : limites des optimisations du compilateur GCC

Nous rappelons au lecteur que dans ce travail de recherche, nous nous intéressons à l'optimisation de la *taille* du code exécutable. Pour produire un code compact, GCC effectue certaines optimisations telles que *l'élimination du code mort (dce)*. Dans cette section, nous allons montrer que GCC est *incapable* d'éliminer tout le code mort présent dans le fichier C++ généré à partir des modèles UML. Plusieurs expériences ont été réalisées sur des machines à états différentes [63].

4.1.1 Exemple illustratif

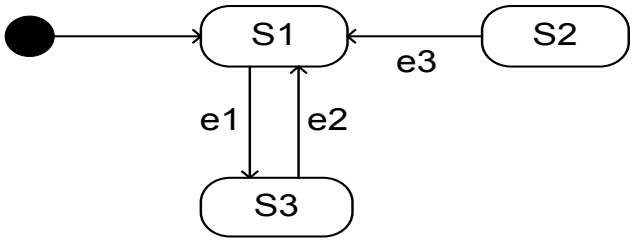
La figure 4.1 présente une machine à états UML composée de 3 états (S1, S2 et S3), 1 pseudo états (l'état initial) et 4 transitions. Nous pouvons remarquer que l'état S2 est un état inatteignable (sans transitions entrantes).

Le code C++ de la figure 4.1 représente un extrait du code généré à partir de la machine à états de la même figure. Ce code a été généré en appliquant le pattern de génération NSC (section 2.1.2.2). Puisque l'état S2 est inatteignable, le code C++ correspondant à cet état (`case S2 : { ... }`) ne sera jamais exécuté. Il constitue ainsi du *code mort*.

Pour éliminer tout le code mort présent dans un fichier C++, GCC fait appel à l'optimisation *dce* à plusieurs reprises (l'exécution d'autres optimisations telles que l*a propagation de constantes* ou *le déplacement d'instructions* hors des boucles peuvent entraîner l'apparition de nouveaux codes morts). Pour voir si le compilateur GCC a pu éliminer le code mort lié à l'état inatteignable, nous avons compilé le code C++ généré à partir de la machine à état de la figure 4.1 en choisissant l'option -Os. Cette option permet d'exécuter toutes les optimisations de GCC qui garantissent la production d'un code exécutable de taille minimale, y compris l'optimisation *dce*. GCC offre également la possibilité de visualiser le résultat de toutes les passes d'optimisations. Pour ce faire, il suffit de passer l'option *-fdump-tree-all* au pilote *g++* (Figure 3.9). Un extrait du fichier généré *exemple1.tree.dce2* est présenté sur la figure 4.1(c). Nous constatons de nouveau la présence du code liée à l'état inatteignable S2.

La génération du fichier assembleur correspondant (Figure 4.1 (d)) nous montre que le compilateur G++ n'a pas éliminé ce code mort. Ce fichier assembleur est produit après avoir effectué toutes les optimisations de GCC. La présence du code lié à l'état inatteignable dans le fichier assembleur confirme le fait que le compilateur GCC n'a pas été capable d'éliminer le code mort lié à l'état inatteignable. En effet, l'optimisation *dce* est implémentée essentiellement pour détecter et supprimer le code mort dans les fichiers C++ tels que la déclaration d'une variable qui n'est jamais utilisée, les instructions englobées dans une condition qui n'est jamais évaluée à *true*, etc. La détection de ce type de code mort est assurée en analysant le fichier C++ d'entrée par rapport à certaines règles liées à la sémantique du langage C++ (La déclaration d'une variable qui n'est pas utilisée est un code mort, etc.). En revanche, considérer le code lié à l'état S2 (`case S2 : { ...}`) comme code mort n'est pas une propriété liée à la sémantique du langage C++. C'est une propriété liée à la sémantique du langage de modélisation UML. En effet, l'information suivante : ≪*un état qui ne possède pas*

de transitions entrantes est un état inatteignable» est une information liée à la sémantique du langage UML. Cette information est perdue lors de la génération de code. Le compilateur GCC prend toutes les informations du système à partir du fichier source C++, il est donc légitime qu'il soit incapable de détecter que ce code est du code mort dans la sémantique UML.



(a)

```
...
switch (fsm_currentstate) {
case S1:
        switch (evt) {
        case e1:
        fsm_currentstate= S3;
        break; default: break; }

break;

case S2:
        switch (evt) {
        case e3:
        fsm_currentstate= S1;
        break;default: break; }

break;

case S3:
        switch (evt) {
        case e2:
        fsm_currentstate= S1;
        break;default: break; }

break ; default: break;
}
...
```

(b)

```
<bb 2>:
  D.2096_2 = this_1(D)->fsm_currentstate;
  D.2097_3 = (int) D.2096_2;
switch (D.2097_3) {
        <default: <L15>,
        case 0: <L18>,
        case 1: <L4>,
        case 2: <L8>,

<L18>:
  evt.0_12 = (int) evt_4(D);
  switch(evt.0_12) <default: <L15>, case 0: <L19>>

<L19>:
  this_1(D)->fsm_crrentstate = 2;
  goto<bb 9> (<L15>);

<L4>:
  evt.0_6 = (int) evt_4(D);
  switch (evt.0_6) <default : <L15>, case 1: <L5>>
<L5>:
  this_1(D)->fsm_crrentstate = 0;
  goto<bb 9> (<L15>);

<L8>:
  evt.0_7 = (int) evt_4(D);
  switch (evt.0_7) <default : <L15>, case 2: <L9>>
<L9>:
  this_1(D)->fsm_crrentstate = 0;

<L15>:
Return;
```

(c)

```
 Movl    (%rdi) %eax
        cmpl    $1, %eax
        je      .L4
        cmpl    $2, %eax
        je      .L5
        testl   %eax, %eax
        jne     .L1
        testl   %esi, %esi
        jne     .L1
        movl    $2, (%rdi)
        ret
.L4
        decl    %esi
        jne     .L1
        movl    $0, (%rdi)
        ret
 .L5
        cmpl    $2, %esi
        jne     .L1
        movl    $0, (%rdi)
 .L1
        ret
```

(d)

FIGURE 4.1 – (a) Une machine à états UML avec un état inatteignable (S2) (b) Un extrait de son code C++ généré à partir du pattern NSC (c) Un extrait du de la forme intermédiaire dce2 du GCC (d) Un extrait du fichier assembleur généré à partir du code C++

Ainsi, dans le code C++ implémentant une machine à état UML, le code correspondant à un état inatteignable est un code mort. La section suivante présente quelques règles de détection des états inatteignables.

4.1.2 Règles de détection des états inatteignables

La première règle de détection d'état inatteignable consiste à vérifier l'existence de transitions entrantes vers cet état. Par exemple, l'état `S2` de la figure 4.1 ne possède pas de transitions entrantes, il est donc inatteignable. D'où la règle suivante :

Règle 1 : *Un état ne possédant pas de transitions entrantes est un état inatteignable.*

Cependant, la détection des états inatteignables dans une machine à états UML ne se limite pas à la vérification de cette règle. En effet, l'état `S2` de la machine à états de la figure 4.2 possède une transition entrante (la transition qui provient de l'état `S5`). Cependant, cet état est inatteignable (il n'existe aucun chemin qui mène à l'état `S2` à partir de l'état initial de la machine à états). D'où la règle sémantique suivante :

Règle 2 : *Un état est inatteignable s'il n'existe aucun chemin (à partir de l'état initial) qui mène vers cet état*

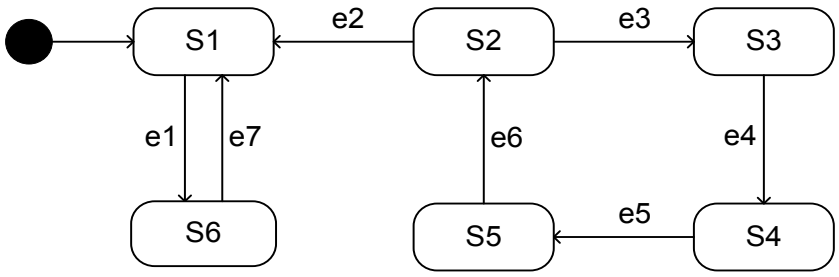


FIGURE 4.2 – Une machine à état UML avec un état inatteignable (`S2`) ayant une transition entrante

Cependant, plusieurs règles liées à la sémantique d'exécution des machines à états sont décrites dans la norme UML et certaines règles, une fois appliquées entraînent l'apparition d'états inatteignables même s'il existe un chemin à partir de l'état initial menant vers ces états. En effet, les transitions entrantes peuvent être non franchissables. Une transition est franchissable si les 3 conditions suivantes sont vérifiées :

- L'état source de la transition est l'un des états actifs du système
- Le système reçoit l'événement déclencheur de la transition
- La garde de la transition (si elle existe) est valide

Ainsi, pour la machine à états de la figure 4.3, si la valeur de x est toujours positive, l'état `S2` est inatteignable vu que la garde de son unique transition entrante est toujours fausse. D'où la règle suivante.

Règle 3 : *Un état possédant des transitions entrantes est inatteignable si et seulement si aucune de ses transitons entrantes n'est franchissable.*

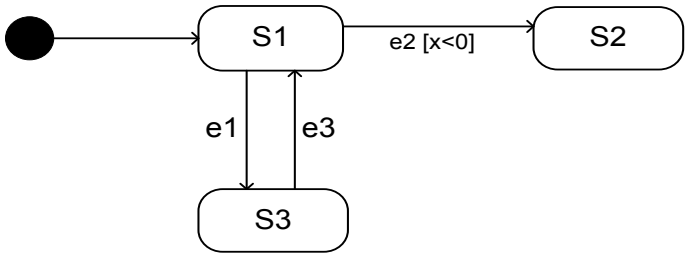


FIGURE 4.3 – Une machine à états UML ayant un état inatteignable à cause d'une garde non valide

La sémantique d'exécution des machines à états UML non concurrentes exige qu'une seule transition soit franchie à la fois. Cela implique que si les 3 conditions exprimées ci-dessus sont valides pour deux transitions franchissables (ou tirables), une seule de ces deux transitions sera franchie. Ces deux transitions sont dites en *conflit*. Pour la gestion de certains types de conflit, UML définit des priorités entre les transitions. À titre d'exemple, les transitions *automatiques* (sans événement déclencheur) sont toujours prioritaires et les transitions *de bas niveau* (originaires d'un état fils) ayant les mêmes événements déclencheurs que les transitions *de haut niveau* (originaires de l'état père) sont prioritaires.

La figure 4.4 présente une machine à état UML où à première vue, tous les états sont atteignables (tous les états possèdent des transitions entrantes). En revanche, l'état `S2` possède une transition automatique qui est donc prioritaire. Ainsi, si le système est dans l'état `S2`, il évoluera automatiquement vers l'état cible de la transition automatique (ici l'état `S4`). Ce qui implique que la transition de `S2` vers `S3` n'est jamais tirée. Puisque c'est la seule transition entrante de l'état `S3`, cet état devient inatteignable.

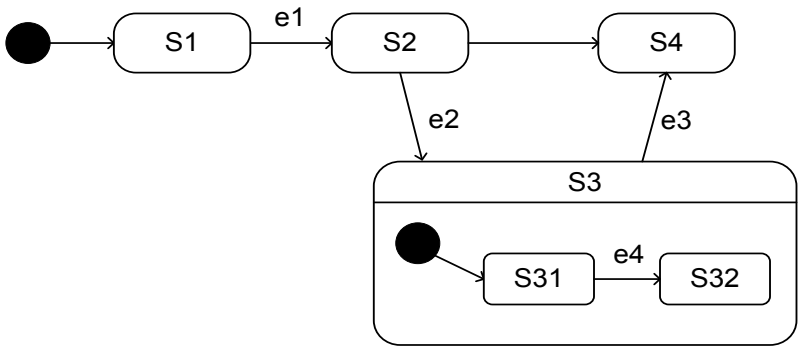


FIGURE 4.4 – Une machine à états UML avec un état composite inatteignable à cause de la priorité de la transition automatique

La figure 4.5 présente une machine à états hiérarchique. Supposons que le système est dans l'état `S21` et il reçoit l'événement `e2`. Dans une machine à états hiérarchique, si l'état fils est actif, son état parent est aussi actif. Ainsi, la réception de l'événement `e2` déclenchera à la fois les deux transitons de l'état fils et de l'état parent. Vu que la transition originaire de l'état fils est prioritaire, la transition de l'état parent `S2` n'est jamais tirée, ainsi l'état `S3` est inatteignable et par conséquent, l'état `S4` l'est aussi.

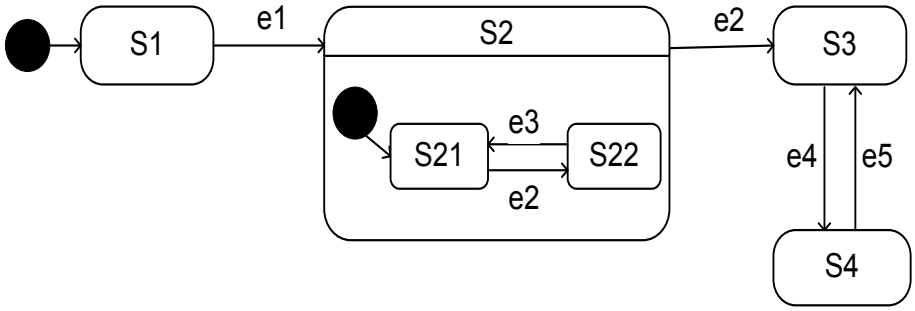


FIGURE 4.5 – Une machine à état UML hiérarchique avec deux états inatteignables (S3) et (S4) à cause de la priorité des transitions de bas niveau

Ainsi, un état qui respecte les 3 règles définies ci-dessus (il possède au moins une transition entrante, il existe un chemin de l'état initial vers cet état, il possède au moins une transition franchissable) peut être inatteignable, et ceci dans le cas où sa transition entrante franchissable (en conflit avec d'autres transitions) n'est pas prioritaire. D'où la règle suivante :

Règle 4 : *un état possédant des transitions entrantes franchissables est inatteignable si et seulement si aucune de ces transitons n'est prioritaires*

4.1.3 Bilan

Les 5 machines à états présentées ci-dessus possèdent toutes au moins un état inatteignable, mais la détection de cet état n'est possible qu'en vérifiant les différentes règles définies ci-dessus, autrement dit, en prenant en compte les informations sémantiques liées au langage de modélisation de ces machines à états. Puisque ces informations sont perdues au cours de la génération de code, GCC est incapable d'éliminer le code mort lié à tous ces états inatteignables. La génération de code C++ à partir de ses 5 machines à états a été faite en appliquant le même pattern de génération de code (le pattern NSC). Nous avons également généré du code à partir de ces machines à états en utilisant les deux autres patterns de génération de code présentés dans la section 2.1.2.2 (SP : *State Pattern* et STT : *State Transition Table*). Par la suite nous avons compilé tout le code généré avec G++. Le même résultat est obtenu : G++ est incapable d'éliminer le code mort lié à un état inatteignable dans une machine à états UML et ceci indépendamment du pattern de génération de code choisi [63].

Pour éliminer ce type de code mort (lié aux états inatteignables), l'exploitation des informations liées à la sémantique d'exécution des machines à états UML s'avère nécessaire. Puisque ces informations sont perdues lors de la génération de code, leur exploitation peut se faire au niveau UML (avant la génération de code). Ainsi la prise en compte de ces informations consiste à implémenter des passes d'optimisation qui permettent de produire un modèle optimisé qui produira à son tour un code exécutable optimisé. L'implémentation de ces optimisations peut également être faite au niveau du générateur de code qui produira un code optimisé à partir d'un modèle non optimisé. Une autre alternative consiste à implémenter ces optimisations dans le compilateur de code. Ainsi le modèle et le code

correspondant ne prennent pas en compte les informations liées à la sémantique d'UML mais c'est uniquement au cours de la compilation que ces informations sont prises en compte. La section suivante présente les avantages et les inconvénients de chaque alternative.

4.2 3 alternatives pour combler les limites des optimisations de GCC

Chacune des 3 alternatives évoquées ci-dessus correspond à l'implémentation des optimisations liées à la sémantique du langage de modélisation UML dans l'une des 3 étapes de l'approche classique dirigée par les modèles pour le développement des systèmes embarqués (Figure 1.1). La première alternative implémente ces optimisations au cours de l'étape de modélisation (avant la génération de code). La deuxième alternative les implémente au cours de la génération de code et la troisième au cours de la compilation (après la génération de code). Après avoir présentée séparément chaque alternative, nous les classifions selon des critères que nous fixons afin de choisir la meilleure alternative.

4.2.1 Optimiser avant la génération de code

Comme nous l'avons déjà expliqué dans le chapitre état de l'art (section 2.2.1.2), la forme intermédiaire SSA a été ajoutée à la phase de synthèse du compilateur GCC pour effectuer les optimisations de haut niveau qui ont besoin de certaines informations du système que la forme RTL a perdu lors de la génération de code (notion de type, le graphe de flot de contrôle, etc). Cependant, la forme SSA, bien qu'elle ait amélioré certaines optimisations de haut niveau, souffre du même problème que la forme RTL : la perte d'informations de haut niveau liées au système qui sont utiles pour l'optimisation. Ce problème de perte d'informations de haut niveau est rencontré dans le cas où le code à compiler provient d'un modèle UML. L'optimisation de l'*élimination de code mort* a été améliorée en l'implémentant dans le niveau SSA au lieu du niveau RTL. L'ajout d'une couche d'optimisation implémentant l'élimination de code mort au niveau UML tout en continuant de bénéficier de cette optimisation au niveau SSA ne peut qu'améliorer cette optimisation. Nous tirons ainsi parti de la montée en abstraction par les modèles pour améliorer le processus de l'optimisation du code exécutable.

L'optimisation de modèles (section 2.2.3) est une transformation de modèle qui prend en entrée un modèle non optimisé et produit en sortie un modèle optimisé en préservant le comportement du modèle de départ. Cette transformation est assurée en analysant le modèle source. La structure compacte d'un fichier *uml* et la facilité de son analyse par rapport à celle du code (forme séquentielle), rend l'implémentation des optimisations au niveau modèle ainsi que celle au niveau du générateur de code plus facile que l'implémentation au niveau du compilateur. En effet, pour implémenter les optimisations au cours de la compilation, une analyse du code (le point d'entrée du compilateur) est nécessaire, alors que lorsque ces optimisations sont implémentées au cours de la génération de code, c'est le modèle qui doit être analysé, ce qui est moins complexe à mettre en œuvre. Les avantages et inconvénients de l'implémentation des optimisations liées à la sémantique du langage UML au cours de la génération de code sont présentés dans la section suivante.

4.2.2 Optimiser pendant la génération de code

Plusieurs générateurs de code tels que Papyrus et Acceleo sont basés sur le mécanisme de *templates* (Figure 3.10). Ainsi implémenter les optimisations liées à la sémantique d'UML consiste à ajouter d'autres *templates* au générateur de code. A titre d'exemple, pour implémenter l'optimisation de l'élimination de l'état inatteignable de la machine à états présentée par la figure 4.1 dans le générateur de code Acceleo, un *template* qui détecte les états inatteignables est ajoutée. En prenant en compte le résultat de ce *template*, Accéleo ne génère du code que pour les états atteignables.

Bien que cette solution soit facile à implémenter, elle engendre plusieurs problèmes liés essentiellement au *débogage* de modèles et à la *certification* des générateurs de code. En effet, une des nouvelles préoccupations liées à la génération de code, mise à part la génération complètement automatique du code exécutable à partir des modèles UML, est le débogage de modèles. Certains générateurs de codes tels que [52] et [89] assurent cette fonctionnalité qui permet, en cas de détection d'une erreur lors de la compilation de code, de débugger directement le modèle, le corriger (s'il le faut) et régénérer le code équivalent au modèle corrigé. L'ancienne approche de débogage consiste à debugger le code et le corriger perdant ainsi le lien de conformité entre le modèle et le code modifié. Certaines approches telles que [90] et [91] ajoutent des liens de traçabilités entre le modèle et le code pour détecter la partie du modèle correspondante au code défaillant. Le débogage de modèle, quand à lui, permet d'ajouter les *breakpoints* directement dans le modèle, qui constitue le seul moyen de communication avec le concepteur qui n'a pas à maitriser les langages de programmation ni leurs techniques de débogage. Cependant, implémenter les optimisations liées à la sémantique des modèles UML dans le générateur de code rend difficile la tâche de débogage de modèles (il est possible d'ajouter un *breakpoint* sur un élément du modèle alors que cet élément, suite à l'exécution des optimisations, n'a plus d'instructions correspondantes dans le code).

Ainsi, implémenter les optimisations dans le générateur de code élargit l'écart entre le modèle et le code généré. Cette réalisation rend aussi ces générateurs de code complexes, difficiles à maintenir et à certifier (certains secteurs comme l'avionique et l'automobile exigent la certification des générateurs de code [68] afin de garantir le bon fonctionnement du code généré). En revanche, il est plus facile de manipuler les générateurs de code UML que de manipuler le compilateur GCC qui fait partie des logiciels les plus complexes (5 millions de lignes de code). La section suivante présente l'alternative de l'implémentation des optimisations liées à la sémantique du langage UML dans le compilateur GCC.

4.2.3 Optimiser après la génération de code

Les informations liées à la sémantique d'exécution des machines à états sont inaccessibles au compilateur GCC qui prend toutes les informations au système à partir du code. Ces informations étant utiles pour améliorer l'optimisation *dce*, une solution pour les prendre en compte consiste à implémenter des nouvelles passes d'optimisation liées à la sémantique du langage UML dans le compilateur GCC.

Cette solution est devenue possible à partir de la version 4.5 du compilateur GCC et ceci grâce à la définition de l'API de constructions de greffons (des *plugins*) [92] qui peuvent

étendre les fonctionnalités du compilateur GCC sans se soucier de la compréhension de tout son code développé depuis 30 ans. Au moment de l'étude de cette alternative, cette API n'est pas encore stable, nous avons ainsi préféré éviter cette solution.

Outre la difficulté de l'extension du compilateur GCC, ajouter ces nouvelles optimisations requiert l'analyse du code généré (le code C++ produit par exemple en appliquant le NSC pattern) afin de détecter les états inatteignables. Vu que les transitions ne sont pas explicitement générées en utilisant le NSC pattern, il est très difficile de déterminer les états inatteignables. De pus, l'implémentation de ces optimisations après la génération de code est dépendante du code généré. Ainsi si le développeur change son pattern de génération de code, l'implémentation de ses optimisations doit aussi changer. Elle changera aussi si le développeur décide de changer le langage cible de la génération de code (de C++ à JAVA par exemple). La deuxième alternative souffre aussi de ce même problème de dépendance de langages et de pattern de génération de code.

Il est à noter que si le concepteur du système décide de changer la sémantique d'exécution des machines à états en fixant autrement les points de variation sémantique [30], toutes les alternatives seront sensibles à ce changement (une ré-implémentation des optimisations en prenant en compte la nouvelle sémantique d'exécution est nécessaire).

4.2.4 Classification des 3 alternatives

Une classification de ces 3 alternatives est présentée dans le tableau 4.1. Les critères de classification sont les suivants : la facilité de détection des états inatteignables, la facilité de l'implémentation de ces nouvelles optimisations, l'indépendance par rapport à l'implémentation des machines à états, l'indépendance par rapport à la variation de la sémantique de la machine à état et l'impact négatif sur le débogage de modèles.

	Facile à implémenter	Facile à détecter	Facilite le débogage des modèles	Independent de l'implémentation de la machine à états	Indépendant de la sémantique des modèles
Après la génération de code	- -	- -	- -	- -	- -
Au cours de la génération de code	+	++	- -	- -	- -
Avant la génération de code	++	++	-	++	- -

Tableau 4.1 – Une classification des 3 alternatives d'implémentation des optimisations liées à la sémantique d'UML

D'après le tableau 4.1, les 3 alternatives affectent le débogage de modèles vu que peu importe le niveau de sa mise en œuvre, l'optimisation est susceptible de changer l'une des représentations du système (le modèle, le code généré ou bien le code assembleur). De la même manière, les 3 solutions proposées pour la prise en compte des informations liées à la sémantique d'UML dépendent toutes de la modification de cette sémantique qui se fixe au niveau le plus haut (le niveau UML). Cependant, la première alternative (implémenter les

optimisations liées à la sémantique du langage UML au niveau UML) est la seule alternative indépendante de l'implémentation de la machine à états UML (le choix du pattern de génération de code et du langage cible). Cette alternative est aussi la plus facile à réaliser (un simple plugin Eclipse transformant une machine à état UML non optimisée vers une autre optimisée (ne contenant pas d'états inatteignables) est suffisante pour la réalisation de l'optimisation. Ainsi, pour prendre en compte les informations liées à la sémantique du langage UML à des fins d'optimisation, nous proposons de modifier l'architecture du compilateur de modèles GUML (Figure 3.7) en ajoutant un niveau d'optimisation autre que SSA et RTL. : le *niveau* UML.

La section suivante présente la nouvelle architecture du compilateur GUML qui en effectuant les optimisations implémentés dans le niveau UML, produit un code assembleur plus compact à partir des machines à états UML possédant des états inatteignables.

4.3 Premier niveau d'optimisation de GUML : le niveau UML

La figure 4.6 montre qu'en ajoutant ce niveau d'optimisation (le niveau UML), toutes les parties qui forment le compilateur GUML contribueront -chacune à sa propre manière- à l'amélioration du processus d'optimisation. Les optimisations de haut niveau liées à la sémantique du langage UML sont implémentées dans le *front-end*. Le *middle-end* est dédié à l'implémentation des optimisations indépendantes du langage source et de la plateforme cible et le *back-end* implémente les optimisations liées à la plateforme cible.

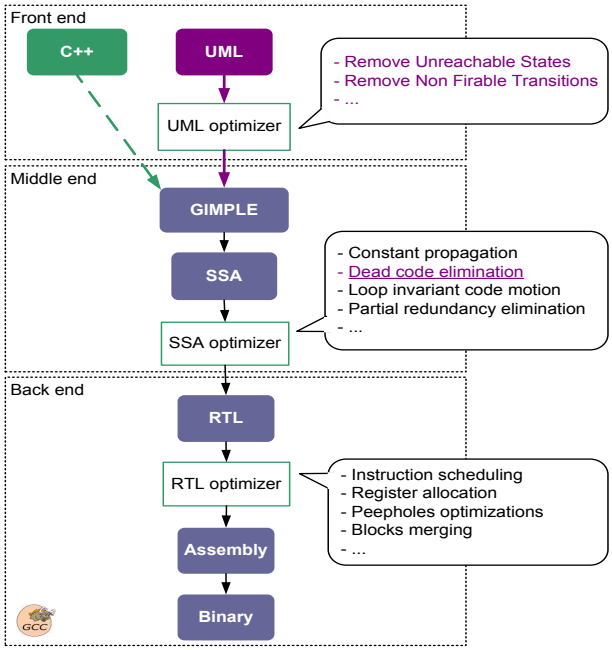


FIGURE 4.6 – L'ajout d'un niveau d'optimisation dans le *front-end* GUML, le niveau UML

Nous détaillerons par la suite l'optimisation de l'*élimination* des états inatteignables (implémentée au niveau UML) afin d'améliorer l'optimisation de l'*élimination du code mort* (implémentée au niveau SSA).

4.3.1 Optimisation de l'élimination des états inatteignables

Nous avons montré au début de ce chapitre (Figure 4.1) que l'optimisation de l'élimination du code mort implémentée au niveau SSA était incapable d'éliminer le code mort lié à la présence d'un état inatteignable dans une machine à états UML. Afin d'améliorer cette optimisation, nous avons implémenté (au niveau UML) l'optimisation qui élimine les états inatteignables et ceci en suivant les règles de détection d'états inatteignables (R1, R2, R3 et R4) présentées dans la section précédente. Notre outil d'optimisation « SM_OPTIMIZER » est un plugin Eclispe qui prend en entrée une machine à états UML non optimisée (contenant des états inatteignables) et renvoie en sortie la machine à états UML optimisée (ne contenant pas d'états inatteignables).

Si la détection de l'état inatteignable de la figure 4.1 exige tout simplement le respect de la règle R1 (tout état sans transitions entrantes est un état inatteignable), la détection des états inatteignables n'est pas toujours évidente. A titre d'exemple, pour optimiser les machines à états des deux figures (Figure 4.2 et Figure 4.3), des techniques plus complexes d'analyse des machines à états UML sont nécessaires. En effet, l'optimisation de la machine à états de la figure 4.2 nécessite de déterminer tous les chemins possibles d'évolution de la machine à états à partir de l'état initial et voir si un des états n'est pas inclus dans aucun de ces chemins. De plus pour optimiser la figure 4.3 il faut déterminer si pour toutes les exécutions possibles de cet automate, la valeur de x est toujours positive.

Pour ne pas ré-implémenter des algorithmes complexes existants et pour tirer profit de l'outil Diversity [8] développé dans notre laboratoire, nous avons implémenté une passerelle à cet outil utilisé essentiellement pour la validation et la génération automatique de tests. L'outil Diversity prend en entrée une spécification XFSP (*eXtensible Formalism for SPecification*) du système à bases d'automates communicants et produit l'arbre d'exécution symbolique couvrant tous les chemins d'exécution possibles du système. Basé sur la technique d'exécution symbolique [76], l'outil Diversity est capable de produire un graphe de couverture des états d'un automate et par la suite la liste des états non couverts. Pour utiliser l'outil Diversity sur les machines à états UML, nous avons développé en utilisant Acceleo un générateur de code « UML2XFSP » qui transforme les machines à états UML simple vers leur spécification XFSP correspondante. Ainsi la passe d'optimisation de notre plugin Eclipse comporte 3 étapes :

1. Faire appel au générateur de code UML2XFSP
2. Faire appel à l'outil Diversity qui renvoi la liste des états inatteignables
3. Supprimer ces états de la machine à états UML

Plusieurs améliorations peuvent être apportées à « SM_OPTIMIZER » : implémenter d'autres optimisations (les optimisations de minimisation des automates tels que l'élimination des états équivalents), établir un ordre automatique d'appel des optimisations, etc. Cependant, dans le cadre de ce travail de recherche, nous avons uniquement besoin d'un outil fonctionnel qui prend en compte les informations liées à la sémantique d'UML pour optimiser les machines à états et par la suite le code exécutable. Améliorer l'implémentation de cet outil, son interfaçage avec Acceleo et Diversity, son utilisation et l'augmentation du nombre de ses optimisations feront partie des perspectives de cette thèse. La suppression des états inatteignables entraine la suppression du code mort liée à ses états que le compilateur GCC était incapable d'éliminer en effectuant l'optimisation *dce* implémentée au niveau SSA.

La section suivante présente les différents gains en termes de taille de code assembleur obtenus en optimisant dans le front end GUML (au niveau UML)les machines à états des figures 4.1, 4.2, 4.3 et 4.4.

4.3.2 Gain engendré par l'ajout du niveau d'optimisation UML

Pour chacune des machines à états présentées dans les figures 4.1, 4.2, 4.3 et 4.4 nous avons généré du code C++ en appliquant le NSC pattern et nous avons compilé le code généré en utilisant G++ et ceci en activant toutes les optimisations qui assurent la production d'un code compact (utilisation de l'option -Os lors de la compilation). Nous avons également compilé les mêmes machines à états avec GUML (pour produire directement la forme GIMPLE) en utilisant la même options -Os, mais en activant également les optimisations de haut niveau implémentées dans le niveau UML (utilisation de l'outil ≪ SM_OPTIMIZER ≫ que nous avons décrit dans la section précédente). Pour chaque machine à états, nous comparons la taille des deux fichiers assembleurs générés respectivement par G++ et GUML.

Nous avons choisi de comparer les fichiers assembleurs et non pas les binaires pour deux raisons : (1) le fichier assembleur représente la dernière forme intermédiaire lisible et compréhensible par les humains. L'analyse de cette forme nous donnera une idée plus claire sur l'effet des optimisations (élimination d'instructions,etc.) (2) GCC n'effectue pas d'optimisations lors du passage de l'assembleur vers le binaire (à part les optimisations de linkage LTO ≪ Link Time Optimization ≫ qui sont utiles lors de la compilation de plusieurs fichiers). Dans notre cas, nous compilons un seul fichier C++ généré à partir d'une ou plusieurs machines à états UML. Pour ces raisons, comparer les fichiers assembleurs ou les fichiers binaires revient au même.

En réalité la différence entre les deux fichiers assembleurs produits représente l'impact de l'ajout des optimisations UML du compilateur GUML, car toutes les autres optimisations (les optimisations des niveaux SSA RTL) sont partagées par les deux compilateurs. Le tableau 4.2 présente ainsi l'impact de l'implémentation des optimisations au niveau UML du *front-end* GUML.

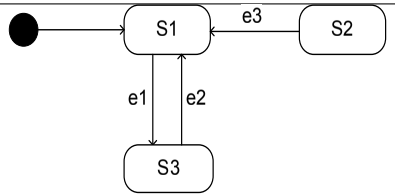
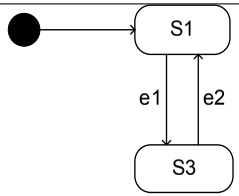
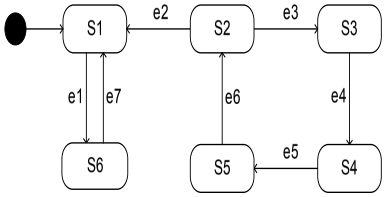
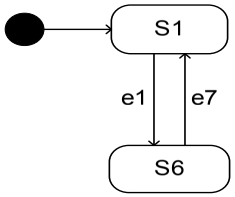
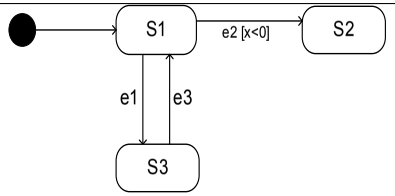
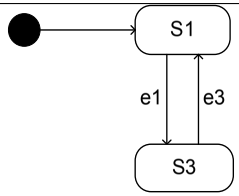
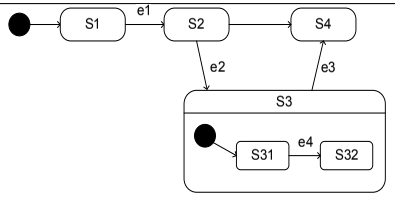
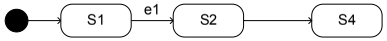
Information utile pour l'optimisation	**Machine à états UML**	**Taille (octets) du fichier assembleur produit par G++**	**Machine à états UML optimisée (optimisation GUML)**	**Taille (octets) du fichier assembleur produit par GUML**	**Taux d'optimisation**
Un état sans transitions entrantes est un état inatteignable		1740		1441	17.18 %
L'ensemble des états non présents dans aucun chemin originaire de l'état initial sont inatteignables		2234		1441	35.49 %
Les transitions de garde toujours fausse sont non franchissables		1684		1441	14.42 %
Les transitions sortantes d'un état ayant une transition de complétion sont non franchissables		7352		1583	78.46 %

Tableau 4.2 – L'impact de l'ajout du premier niveau d'optimisation (niveau UML) du compilateur GUML sur la taille du code assembleur.

Puisque ces optimisations améliorent l'optimisation de l'*élimination du code mort*, tous les fichiers assembleurs produits par GUML sont plus compacts que ceux produits par G++. Cependant, le gain en termes de taille de code varie selon le nombre d'états supprimés et le nombre d'états dans le modèle de départ (environ 17% pour la suppression d'un seul état parmi 3 (deuxième ligne du tableau 4.2) et environ 35% pour la suppression de 4 états parmi 6 (troisième ligne du tableau 4.2)).

Ce gain varie aussi selon le type de la machine à états de départ (plus que 75% de gain pour la suppression d'un seul état composite parmi 3 états). En effet, dans le cas d'une machine à états UML hiérarchique (présence d'au moins un état composite), l'état courant ne peut plus être représenté par un littéral d'une énumération. En effet, plusieurs états peuvent être actifs à la fois (par exemple en franchissant la transition de S2 vers S3 de la figure 4.4 (5ème ligne du tableau 4.2), l'état actif passe de S2 vers l'ensemble { S3, S31}, et en franchissant la transition de S31 vers S32, l'ensemble des états actifs (la configuration des états selon la norme UML2) passe de { S3, S31} vers { S3, S32}.

Nous avons choisi d'implémenter l'ensemble des états actifs en utilisant une *liste* (plus flexible qu'un tableau). Ainsi, le passage d'un état actif à un autre implique deux appels à des opérations différentes sur les listes : *remove(élément)* pour supprimer le ou les anciens états actifs, et *insert(élément)* pour ajouter le ou les nouveaux états actifs ce qui augmente la taille du code assembleur généré à partir de la machine à états hiérarchique. L'ajout de `#include <list>` en tête du fichier C++ implémentant la machine à états augmentera aussi la taille du code assembleur puisque le code relatif à la bibliothèque `<list>` sera aussi inclus dans le fichier assembleur. La suppression de l'état composite inatteignable de la machine à états de la figure 4.4 (5ème ligne du tableau 4.2) a changé le type de la machine à états de hiérarchique à simple. L'implémentation d'une machine à états simple n'incluant pas la bibliothèque `<list>` donne un code assembleur beaucoup plus compact (environ 78%).

La version actuelle de GUML ne permet pas de compiler les machines à états hiérarchiques. C'est pour cette raison que la machine à états de la figure 4.5 n'est pas présentée dans le tableau 4.2. La prise en compte des états composites dans la compilation des machines à états UML est l'une des perspectives à court terme de ce travail.

4.4 Conclusion

Nous avons vu dans ce chapitre que le compilateur GCC était incapable d'éliminer tout le code mort présent dans un fichier C++ généré à partir d'une machine à états UML. Cet inconvénient du compilateur GCC vient du fait que son plus haut niveau d'optimisation (le niveau SSA) ne contient pas toutes les informations liées au langage source (le langage UML), en particulier les informations liées à la sémantique d'exécution des machines à états UML. Comme nous l'avons montré, ces informations, perdues lors de la génération de code, sont utiles pour les optimisations. Nous avons étudié les 3 alternatives de l'exploitation de ces informations au cours des 3 étapes du développement dirigé par les modèles des systèmes embarqués (la modélisation, la génération de code et la compilation). Nous avons conclu que la meilleure alternative était d'exploiter ces informations au niveau UML. Ainsi, nous avons implémenté l'outil d'optimisation « SM_OPTIMIZER » ajouté au modeleur

Papyrus qui représente le premier niveau d'optimisation du compilateur GUML.

L'ajout des optimisations au niveau UML ont produit un code assembleur plus compact que le code assembleur produit par G++ à partir de la même machine à états UML (tableau 4.2). Ceci est dû au fait que le compilateur G++ prend en entrée le code généré à partir de la machine à états UML non optimisée. Ainsi, même si le compilateur GUML produit un code plus compact, cela ne met pas en cause le compilateur G++ qui peut produire le même code assembleur s'il compile le code C++ généré à partir de la machine à états optimisée. En d'autres termes, le compilateur G++ peut aussi bénéficier de l'apport de ce premier niveau d'optimisation si nous supposons que les générateurs de code C++ optimisent leurs modèles avant la génération de code. Cependant, les générateurs de code les plus utilisés comme Rhapsody, BridgePoint et iUML n'effectuent pas les optimisations de haut niveau de GUML. L'ajout de ce premier niveau d'optimisation apporte une réponse à ce problème, mais nous allons montrer dans le chapitre suivant qu'il existe des cas où ce niveau d'optimisation s'avère insuffisant. L'ajout d'un autre niveau d'optimisation intermédiaire entre le niveau UML et la forme intermédiaire SSA est nécessaire.

CHAPITRE 5

GUML un compilateur de modèles optimisant : deuxième niveau d'optimisation

Dans ce chapitre, nous montrons que même en compilant une machine à états UML optimisée (ne possédant pas d'états inatteignables) GUML (grâce à son deuxième niveau d'optimisation) est capable de produire un code assembleur plus compact que le code assembleur généré par G++ à partir de la même machine à états optimisée. Pour cela, la première section de ce chapitre détaille la compilation des machines à états UML en utilisant GUML. La deuxième section présente le deuxième niveau d'optimisation ajouté à GUML. Les optimisations implémentées dans ce niveau sont présentées dans la troisième section.

5.1 Exemple illustratif : la compilation des machines à états UML

La figure 5.1 servira de support illustratif dans la description du deuxième niveau d'optimisation de GUML. Il s'agit d'une machine à états UML à 3 états (S1, S2 et S3) et 9 transitions qui assurent le passage du système d'un état à un autre suite à la réception de l'un des 4 signaux e1, e2, e3 et e4.

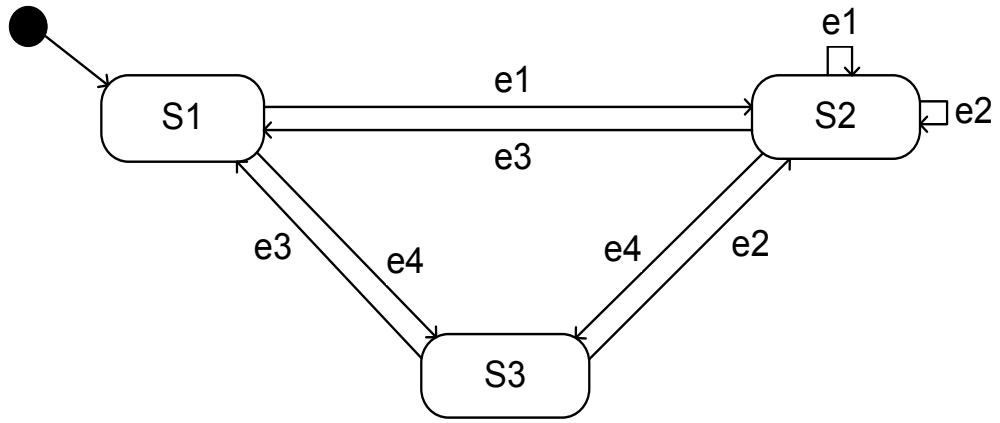


FIGURE 5.1 – Exemple d'une machine à états UML optimisée(tous les états sont atteignables)

Nous pouvons remarquer que tous les états de la machine à états UML de la figure 5.1 sont atteignables. Ainsi, l'exécution de l'optimisation (*élimination des états inatteignables*) implémentée dans le premier niveau d'optimisation de GUML reste sans effet sur cette machine à états. Par conséquent, les deux formes GIMPLE produites par G++ et GUML à partir de cette même machine à états sont pratiquement équivalentes. Dans ce qui suit, nous détaillerons le résultat de la compilation d'un modèle UML (Figure 5.2) dont le comportement du système est modélisé par la machine à états UML de la figure 5.1.

Le modèle UML exécutable de la figure 5.2 est formé de 3 diagrammes UML différents : un diagramme de classe (contenant une opération *main()*), un diagramme d'activité (pour la modélisation du comportement de l'opération *main()*) et une machine à états (pour la modélisation du changement de l'état du système suite à la réception d'un signal). L'opération *main()* de la classe *MyClass* se charge de l'instanciation de la classe et de l'envoi des signaux que cette même classe est prête à recevoir. Nous avons choisi dans cet exemple d'envoyer successivement les 3 signaux (e1, e2 et e3).

Comme on l'a déjà expliqué dans (section 3.1.2) nous supposons que le temps écoulé entre l'envoi et la réception d'un signal est nul. Ainsi, ces deux événements sont implémentés par un même appel à une nouvelle opération *signal_operation()* du receveur (la classe *MyClass*). L'implémentation d'une machine à états consiste à décrire le comportement du système suite à la réception de chaque signal. Ce comportement est décrit dans le corps de l'opération *signal_operation()*.

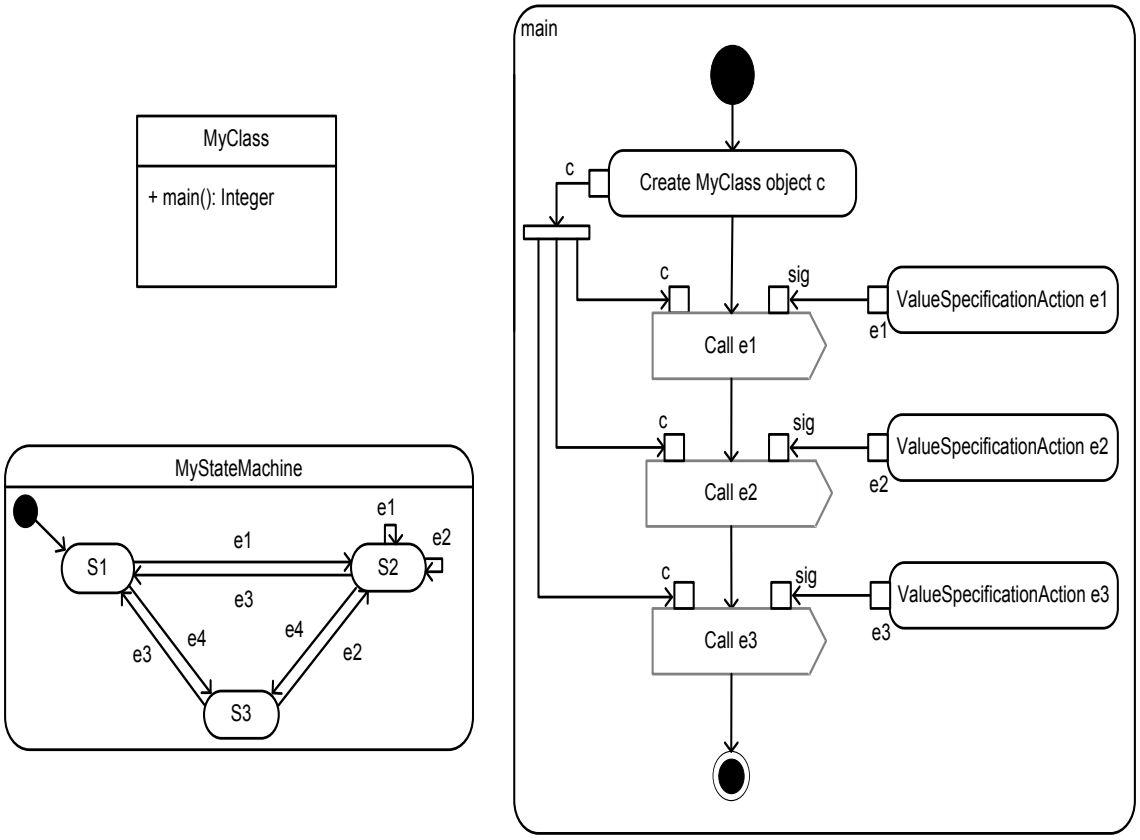


FIGURE 5.2 – Un modèle UML exécutable formé de 3 diagrammes UML : un diagramme de classe, un diagramme d'activité et une machine à états

Le code C++ implémentant le modèle de la figure 5.2 ainsi que la forme GIMPLE que G++ génère en compilant ce code sont présentés par la figure 5.3. Mise à part les instructions GIMPLE utilisées pour l'exécution des activités (Figure 3.11) telles que *gimple_assign*, *gimple_call* et *gimple_return*, de nouvelles instructions GIMPLE comme *gimple_switch*, *gimple_label* et *gimple_goto* sont utilisées pour décrire la forme GIMPLE équivalente à la machine d'états UML de la figure 5.1.

La forme GIMPLE produite par GUML en compilant directement le modèle décrit par la figure 5.2 est présentée sur la figure 5.4. Nous pouvons remarquer que, pour l'implémentation du diagramme d'activité (*main*) de la figure 5.2, les deux formes GIMPLE générées par G++ et GUML sont équivalentes (voir la fonction *main()* des deux figures 5.3 et 5.4). Cependant, une légère différence apparait dans l'implémentation de la machine à états UML. En effet, G++, part du code C++ généré à partir du patron NSC, utilise l'instruction *gimple_switch* pour tester la valeur de l'état courant (*fsm_current_state*) et celle de l'événement reçu (*fsm_event*). Par contre, GUML part directement du modèle UML, n'utilisant ainsi aucune construction d'un langage de programmation (i.e, les *switchs_cases*). Il se base de son coté sur l'instruction de base de GIMPLE pour effectuer les tests : l'instruction *gimple_cond* (Figure 5.4).

```
enum fsm_events {
        e1,
        e2,
        e3,
        e4, };

enum fsm_states {
        S1,
        S2,
        S3, };

class MyClass {
public : fsm_states fsm_currentstate;
public : void signal_operation
(fsm_events evt) {
switch (fsm_currentstate) {

case S1:
        switch (evt) {
          case e1:
             fsm_currentstate= S2;
          break;
          case e4:
              fsm_currentstate= S3;
          break;
          default: break; }
break;
case S2:

        switch (evt) {
          case e1:
             fsm_currentstate= S2;
          break;
          case e4:
             fsm_currentstate= S3;
          break;
          case e3:
             fsm_currentstate= S1;
          break;
          case e2:
             fsm_currentstate= S2;
          break;
          default: break; }
break;
case S3:

        switch (evt) {
          case e3:
             fsm_currentstate= S1;
          break;
          case e2:
             fsm_currentstate= S2;
          break;
          default: break; }
break;

default : break; }
} };

int main() {
MyClass c;
c.fsm_currentstate=S1;
c.signal_operation(e1);
c.signal_operation(e2);
c.signal_operation(e3);
return 0;
}
```

(a)

```
int main() ()
  gimple_bind <
  int D.2099;
  gimple_bind <
    struct MyClass c;
    gimple_assign <integer_cst, c.fsm_currentstate, 0, NULL>
    gimple_call <signal_operation, NULL, &c, 0>
    gimple_call <signal_operation, NULL, &c, 1>
    gimple_call <signal_operation, NULL, &c, 2>
    gimple_assign <integer_cst, D.2099, 0, NULL>
    gimple_return <D.2099>>>

void MyClass::signal_operation(fsm_events) (struct MyClass * const this, fsm_events evt)
gimple_bind <
  fsm_states D.2102; int D.2103; int evt.0;

  gimple_assign <component_ref, D.2102, this->fsm_currentstate, NULL>
  gimple_assign <nop_expr, D.2103, D.2102, NULL>
  gimple_switch <D.2103, default: <D.2094>, case 0: <D.2080>, case 1: <D.2084>,
  case 2: <D.2090>>
  gimple_label <<D.2080>>
  gimple_assign <nop_expr, evt.0, evt, NULL>
  gimple_switch <evt.0, default: <D.2083>, case 0: <D.2081>, case 3: <D.2082>>
  gimple_label <<D.2081>>
  gimple_assign <integer_cst, this->fsm_currentstate, 1, NULL>
  gimple_goto <<D.2104>>
  gimple_label <<D.2082>>
  gimple_assign <integer_cst, this->fsm_currentstate, 2, NULL>
  gimple_goto <<D.2104>>
  gimple_label <<D.2083>>
  gimple_goto <<D.2104>>
  gimple_label <<D.2104>>
  gimple_goto <<D.2101>>
  gimple_label <<D.2084>>
  gimple_assign <nop_expr, evt.0, evt, NULL>
  gimple_switch <evt.0, default: <D.2089>, case 0: <D.2085>, case 1: <D.2088>,
  case 2: <D.2087>, case 3: <D.2086>>
  gimple_label <<D.2085>>
  gimple_assign <integer_cst, this->fsm_currentstate, 1, NULL>
  gimple_goto <<D.2106>>
  gimple_label <<D.2086>>
  gimple_assign <integer_cst, this->fsm_currentstate, 2, NULL>
  gimple_goto <<D.2106>>
  gimple_label <<D.2087>>
  gimple_assign <integer_cst, this->fsm_currentstate, 0, NULL>
  gimple_goto <<D.2106>>
  gimple_label <<D.2088>>
  gimple_assign <integer_cst, this->fsm_currentstate, 1, NULL>
  gimple_goto <<D.2106>>
  gimple_label <<D.2089>>
  gimple_goto <<D.2106>>
  gimple_label <<D.2106>>
  gimple_goto <<D.2101>>
  gimple_label <<D.2090>>
  gimple_assign <nop_expr, evt.0, evt, NULL>
  gimple_switch <evt.0, default: <D.2093>,  case 1: <D.2092>, case 2: <D.2091>>
  gimple_label <<D.2091>>
  gimple_assign <integer_cst, this->fsm_currentstate, 0, NULL>
  gimple_goto <<D.2107>>
  gimple_label <<D.2092>>
  gimple_assign <integer_cst, this->fsm_currentstate, 1, NULL>
  gimple_goto <<D.2107>>
  gimple_label <<D.2093>>
  gimple_goto <<D.2107>>
  gimple_label <<D.2107>>
  gimple_goto <<D.2101>>
  gimple_label <<D.2094>>
  gimple_goto <<D.2101>>
  gimple_label <<D.2101>>>
```

(b)

FIGURE 5.3 – (a) Le code C++ généré à partir du modèle de la figure 5.2 (b) la forme GIMPLE équivalente produite par G++

```
main ()
 gimple_bind <
 <unnamed-signed:32> D.53;
 struct MyClass c;

 gimple_assign <integer_cst, c.fsm_currentstate, 0, NULL>
 gimple_call <signal_operation, NULL, &c, 0>
 gimple_call <signal_operation, NULL, &c, 1>
 gimple_call <signal_operation, NULL, &c, 2>
 gimple_assign <integer_cst, D.53, 0, NULL>
 gimple_return <D.53>>

signal_operation (struct MyClass * this, <unnamed-signed:32> fsm_evt)
gimple_bind <
 <unnamed-signed:32> iftmp.0;
 <unnamed-signed:32> fsm_currentstate1;

 gimple_label <L0>
 gimple_assign <component_ref, fsm_currentstate1, this->fsm_currentstate, NULL>
 gimple_cond <eq_expr, fsm_currentstate1, 0, L3, L1>
 gimple_label <L3>
 gimple_cond <eq_expr, fsm_evt, 0, L4, L5>
 gimple_label <L4>
 gimple_assign <integer_cst, this->fsm_currentstate, 1, NULL>
 gimple_goto <fin>
 gimple_label <L5>
 gimple_cond <eq_expr, fsm_evt, 3, L6, fin>
 gimple_label <L6>
 gimple_assign <integer_cst, this->fsm_currentstate, 2, NULL>
 gimple_goto <fin>
 gimple_label <L1>
 gimple_cond <eq_expr, fsm_currentstate1, 1, L7, L2>
 gimple_label <L7>
 gimple_cond <eq_expr, fsm_evt, 0, L8, L9>
 gimple_label <L8>
 gimple_assign <integer_cst, this->fsm_currentstate, 1, NULL>
 gimple_goto <fin>
 gimple_label <L9>
 gimple_cond <eq_expr, fsm_evt, 3, L10, L11>
 gimple_label <L10>
 gimple_assign <integer_cst, this->fsm_currentstate, 2, NULL>
 gimple_goto <fin>
 gimple_label <L11>
 gimple_cond <eq_expr, fsm_evt, 1, L12, L13>
 gimple_label <L12>
 gimple_assign <integer_cst, this->fsm_currentstate, 1, NULL>
 gimple_goto <fin>
 gimple_label <L13>
 gimple_cond <eq_expr, fsm_evt, 2, L14, fin>
 gimple_label <L14>
 gimple_assign <integer_cst, this->fsm_currentstate, 0, NULL>
 gimple_goto <fin>
 gimple_label <L2>
 gimple_cond <eq_expr, fsm_currentstate1, 2, L15, fin>
 gimple_label <L15>
 gimple_cond <eq_expr, fsm_evt, 1, L16, L17>
 gimple_label <L16>
 gimple_assign <integer_cst, this->fsm_currentstate, 1, NULL>
 gimple_goto <fin>
 gimple_label <L17>
 gimple_cond <eq_expr, fsm_evt, 2, L18, fin>
 gimple_label <L18>
 gimple_assign <integer_cst, this->fsm_currentstate, 0, NULL>
 gimple_goto <fin>
 gimple_label <fin>
```

FIGURE 5.4 – La forme GIMPLE produite directement par GUML à partir du modèle exécutable de la figure 5.2

Les deux fichiers assembleurs générés à partir de ces deux formes GIMPLE légèrement différentes sont à leurs tours différents. Cependant, ils contiennent tous les deux le même nombre d'instructions assembleurs (32 instructions). La figure 5.5 (a) présente l'extrait du fichier assembleur généré par G++ représentant l'implémentation de la machine à états UML du modèle de la figure 5.2. La même implémentation, mais cette fois-ci extraite du fichier assembleur produit par GUML est donnée par la figure 5.5 (b). Nous pouvons remarquer que la différence au niveau des deux formes GIMPLE n'est plus apparente puisque l'instruction *gimple_switch* suivie des *gimple_goto* tout comme l'instruction de *gimple_cond* suivie des *gimple_goto*, sont transformées au niveau de l'assembleur vers des instructions de tests (*cmpl*, *testl*, etc.) suivies de sauts conditionnels (*jne*, *jmp*, etc.)

```
_ZN7MyClass16signal_operationE10fsm_events:
.LFB0:
        movl    (%rdi), %eax
        cmpl    $1, %eax
        je      .L4
        cmpl    $2, %eax
        je      .L5
        testl   %eax, %eax
        jne     .L1
        testl   %esi, %esi
        je      .L9
        jmp     .L17
.L4:
        cmpl    $1, %esi
        je      .L9
        jg      .L12
        testl   %esi, %esi
        je      .L9
        ret
.L12:
        cmpl    $2, %esi
        je      .L15
.L17:
        cmpl    $3, %esi
        jne     .L1
        movl    $2, (%rdi)
        ret
.L9:
        movl    $1, (%rdi)
        ret
.L5:
        cmpl    $1, %esi
        je      .L13
        cmpl    $2, %esi
        jne     .L1
.L15:
        movl    $0, (%rdi)
        ret
.L13:
        movl    $1, (%rdi)
.L1:
        ret
.LFE0:
        .size
        _ZN7MyClass16signal_operationE10fsm_events,
        .-_ZN7MyClass16signal_operationE10fsm_events
```

(a)

```
signal_operation.26:
.LFB0:
.L2:
        movl    (%rdi), %eax
        testl   %eax, %eax
        jne     .L3
.L4:
        testl   %esi, %esi
        je      .L25
.L5:
        cmpl    $3, %esi
        jne     .L24
        jmp     .L26
.L3:
        cmpl    $1, %eax
        jne     .L9
.L10:
        testl   %esi, %esi
        je      .L24
.L11:
        cmpl    $3, %esi
        jne     .L14
.L26:
        movl    $2, (%rdi)
        ret
.L14:
        cmpl    $1, %esi
        je      .L24
.L15:
        cmpl    $2, %esi
        jne     .L24
.L16:
        movl    $0, (%rdi)
        ret
.L9:
        cmpl    $2, %eax
        jne     .L24
.L17:
        cmpl    $1, %esi
        jne     .L18
.L25:
        movl    $1, (%rdi)
        ret
.L18:
        cmpl    $2, %esi
        jne     .L24
.L20:
        movl    $0, (%rdi)
.L24:
        ret
.LFE0:
        .size
        signal_operation.26,
        .-signal_operation.26
```

(b)

: Changement de valeur de l'état courant (ex: fsm_current_sate= S1)
: Test de la valeur de l'état courant (ex: if (fsm_current_state== S1)
: Test de la valeur de l'évennement reçu: (ex: if (evt==e1)

FIGURE 5.5 – Implémentation en assembleur de la machine à états UML de la figure 5.1 : (a) assembleur généré par G++ (b) assembleur généré par GUML

Il est à noter que la taille totale du fichier assembleur généré à partir de GUML est plus compacte que celle du fichier assembleur produit par G++ (deuxième ligne du tableau 5.1). Cette différence de taille est principalement due au changement de noms (*name mangling*) des méthodes que G++ impose afin d'éliminer toute confusion lors de la déclaration de deux méthodes ayant le même nom mais des signatures différentes. La figure 5.6 présente les deux instructions assembleur de l'appel de la méthode *signal_operation()* générées respectivement par G++ et GUML.

Type de fichier	Taille (octets) du fichier produit par G++	Taille (octets) du fichier produit par GUML
Assembleur (.s)	2021	1815
Binaire (.o)	1952	1448
Exécutable (.out)	8092	7927

Tableau 5.1 – Tailles des fichiers assembleur, binaire et exécutable générés par G++ et GUML à partir de la machine à états UML de la figure 5.1

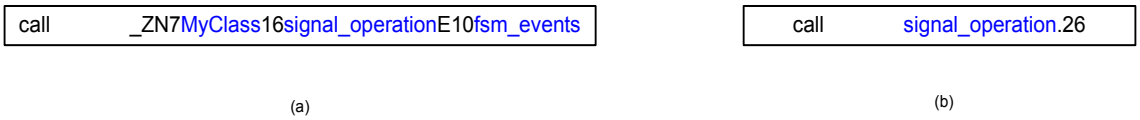


FIGURE 5.6 – L'instruction d'appel de l'opération signal_operation() dans le fichier assembleur : (a) généré par G++ (b) généré par GUML

L'augmentation du nombre de caractères (19 caractères pour GUML contre 42 caractères pour G++) dans le nom de la méthode utilisée par G++ implique une augmentation dans la taille totale du fichier assembleur (le nom de la méthode *signal_operation()* est utilisé 10 fois dans le fichier assembleur produit à partir de la figure 5.1). Ce changement de nom de méthode augmente également la taille du fichier binaire (.o) et du fichier exécutable (.out) produits à partir du modèle de départ. Puisque GUML ne traite pas actuellement ce problème de la confusion éventuelle lors de la l'appel d'une méthode redéfinie, on ne peut pas affirmer que G++ génère un code moins compact. Notre objectif est de produire à partir d'une machine à états UML un code assembleur plus compact dans le sens où il contient moins d'instructions et non pas des noms de variables contenant moins de caractères.

Nous rappelons que selon la figure 5.5, le nombre d'instructions assembleur implémentant la machine à états UML de la figure 5.1 est le même pour les deux compilateurs (32 instructions). Ce résultat est attendu vu que les deux formes GIMPLE à partir desquelles les deux fichiers assembleur sont générés sont équivalentes. Les deux graphes de flot de contrôle (GFC) produits à partir de ces deux formes GIMPLE sont à leurs tours équivalents. Effectuer les mêmes passes d'optimisation (les optimisations du *middle* et *back-ends* du GCC) sur ces deux GFC produira certainement des codes assembleur équivalents. Ainsi, obtenir un code assembleur plus compact que celui généré à partir de la forme GIMPLE produite par G++ passe nécessairement par la production d'une forme GIMPLE différente de celle produite par G++. À partir de cette nouvelle forme, un nouveau

GFC sera généré. Pour assurer la production d'un code assembleur plus compact, le nouveau GFC doit répondre à l'une des deux conditions suivantes :

- Condition 1 : Ce GFC est plus optimisé que le GFC produit par G++ (contient moins de blocs de base ou moins d'expressions). Cette minimisation du nombre d'instructions va être propagée jusqu'à la forme assembleur.
- Condition 2 : Ce GFC améliore le résultat d'optimisation d'au moins une passe d'optimisation du *middle* ou du *back-end* qui garantit la réduction de la taille de l'assembleur. (Par exemple, à partir du nouveau GFC, l'optimisation de l'*élimination de code mort* (*dce*) détectera plus de code mort, ou bien celle de l'*élimination des expressions redondantes partielles* (*pre*) découvrira plus d'expressions redondantes.)

Cette dernière condition demande une étude de l'implémentation des optimisations de haut niveau et de bas niveau du compilateur G++ pour produire un GFC sur lequel ces optimisations sont plus performantes. Cependant, pour assurer la première condition, nous n'avons pas à étudier les implémentations des optimisations de GCC ni le GFC lui-même. Tout ce que nous avons à faire, est d'essayer d'optimiser la forme GIMPLE produite à partir de GUML, ce qui implique la production d'un GFC optimisé et par conséquent un assembleur plus compact.

GUML produit directement la forme GIMPLE à partir des machines à états UML. Dans la section suivante, nous introduisons une nouvelle forme intermédiaire nommée G-Graph (GIMPLE-Graph) produite à partir des machines à états UML et à partir de laquelle une forme GIMPLE optimisée sera générée. Sur cette nouvelle forme, et par analogie à la première forme d'optimisation ajoutée par GUML (la forme UML), certaines optimisations seront implémentées. Si le premier niveau d'optimisation de GUML (le niveau UML) a amélioré l'optimisation d'*élimination de code mort* (*dce*), nous allons voir dans la section suivante que ce deuxième niveau d'optimisation (le niveau G-Graph) améliore l'optimisation de *la fusion de blocs de base* (*block merging*) qui élimine des instructions redondantes dans le GFC en fusionnant les blocs de bases contenant les mêmes instructions.

5.2 Forme intermédiaire G-Graph (GIMPLE_GRAPH)

En transformant une machine à états UML vers la forme GIMPLE équivalente peu importe le *front-end* utilisé (G++ ou GUML), le compilateur GCC perd encore une fois des informations qui sont liées cette fois ci à la nature du modèle à compiler. En effet, le compilateur GCC ne se rend plus compte qu'il est entrain de compiler des *automates*. Ainsi, en se basant sur son GFC que lui-même produit à partir de la forme GIMPLE, GCC se trouve dans l'incapacité d'éliminer certaines expressions redondantes. Ces expressions peuvent être éliminées si le graphe de flot de contrôle produit directement à partir des machines à états UML a été généré et optimisé avant même la construction de la forme GIMPLE. Pour illustrer cette idée, revenons au code C++ produit à partir de la machine à états UML de la figure 5.1. En analysant ce code (Figure 5.7), nous remarquons la présence de code redondant. En effet, les instructions des lignes 6→11 et celles des lignes 16→21 sont les mêmes et sont exécutées si le système est dans l'un des deux états S1 ou S2. De la même manière les instructions des lignes 22→27 et celles des lignes 32→37 sont exécutées si le système est dans l'un des deux états S2 ou S3.

```
1  public : void signal_operation (fsm_events evt)
2  {
3  switch (fsm_currentstate) {
4  case S1:
5        switch (evt) {
6           case e1:
7              fsm_currentstate= S2;
8           break;
9           case e4:
10             fsm_currentstate= S3;
11          break;
12          default: break; }
13 break;
14 case S2:
15       switch (evt) {
16          case e1:
17             fsm_currentstate= S2;
18          break;
19          case e4:
20             fsm_currentstate= S3;
21          break;
22          case e3:
23             fsm_currentstate= S1;
24          break;
25          case e2:
26             fsm_currentstate= S2;
27          break;
28          default: break; }
29 break;
30 case S3:
31       switch (evt) {
32          case e3:
33             fsm_currentstate= S1;
34          break;
35          case e2:
36             fsm_currentstate= S2;
37          break;
38          default: break; }
39 break;
40 default : break; }
41 }
```

(a)

```
1  public : void signal_operation (fsm_events evt)
2  {
3  If (fsm_currentstate==S1) || (fsm_currentstate==S2)
4  switch (evt) {
5        case e1:
6              fsm_currentstate= S2;
7        break;
8        case e4:
9              fsm_currentstate= S3;
10       break;
11       default: break; }
12
13 If (fsm_currentstate==S2) || (fsm_currentstate==S3)
14 switch (evt) {
15       case e3:
16             fsm_currentstate= S1;
17       break;
18       case e2:
19             fsm_currentstate= S2;
20       break;
21       default: break; }
22 }
```

(b)

FIGURE 5.7 – Le code C++ généré à partir de la figure 5.1 avec (a) et sans (b) expressions redondantes

Ainsi, une factorisation de ces expressions communes à deux tests différents (en utilisant l'opérateur logique OU) peut éliminer le code redondant présent dans cet exemple. Le code C++ qui ne contient pas d'expressions redondantes est présenté dans la figure 5.7(b). Il est à noter que le compilateur GCC est capable d'éliminer le code redondant lié au changement de la valeur de l'état courant. En effet, le code assembleur de la figure 5.5(a) montre que ce changement de valeur ne se fait qu'une seule fois. A titre d'exemple, l'instruction assembleur «`movl $0, (%rdi)`» correspondante à l'instruction C++ «`fsm_current_state=S1`» est présente une seule fois dans tout le fichier assembleur.

En revanche, le compilateur GCC est incapable d'éliminer le code redondant qui teste la valeur de l'événement reçu. A titre d'exemple, la même Figure 5.5(a) montre la présence de l'instruction assembleur «`cmpl $1, %esi`» correspondante à l'instruction C++ «`case e1`» deux fois.

Nous rappelons au lecteur que toutes les passes d'optimisation d'élimination de code redondant sont faites soit au niveau SSA (le *middle-end*) soit au niveau RTL (le *bak-end*). A partir du code assembleur qui représente la cible de la dernière transformation du *back-end*, aucune optimisation d'élimination de code redondant n'est possible. Ainsi, le compilateur GCC n'est capable d'éliminer que certaines instructions redondantes. C'est le GFC imposé par la forme GIMPLE qui est derrière cette incapacité d'éliminer tout le code redondant.

La figure 5.8 présente le GFC produit par GCC à partir de la forme GIMPLE de la figure 5.4 (en spécifiant l'option *-fdump-tree-cfg*, le compilateur GCC génère le GFC produit à partir de la forme GIMPLE).

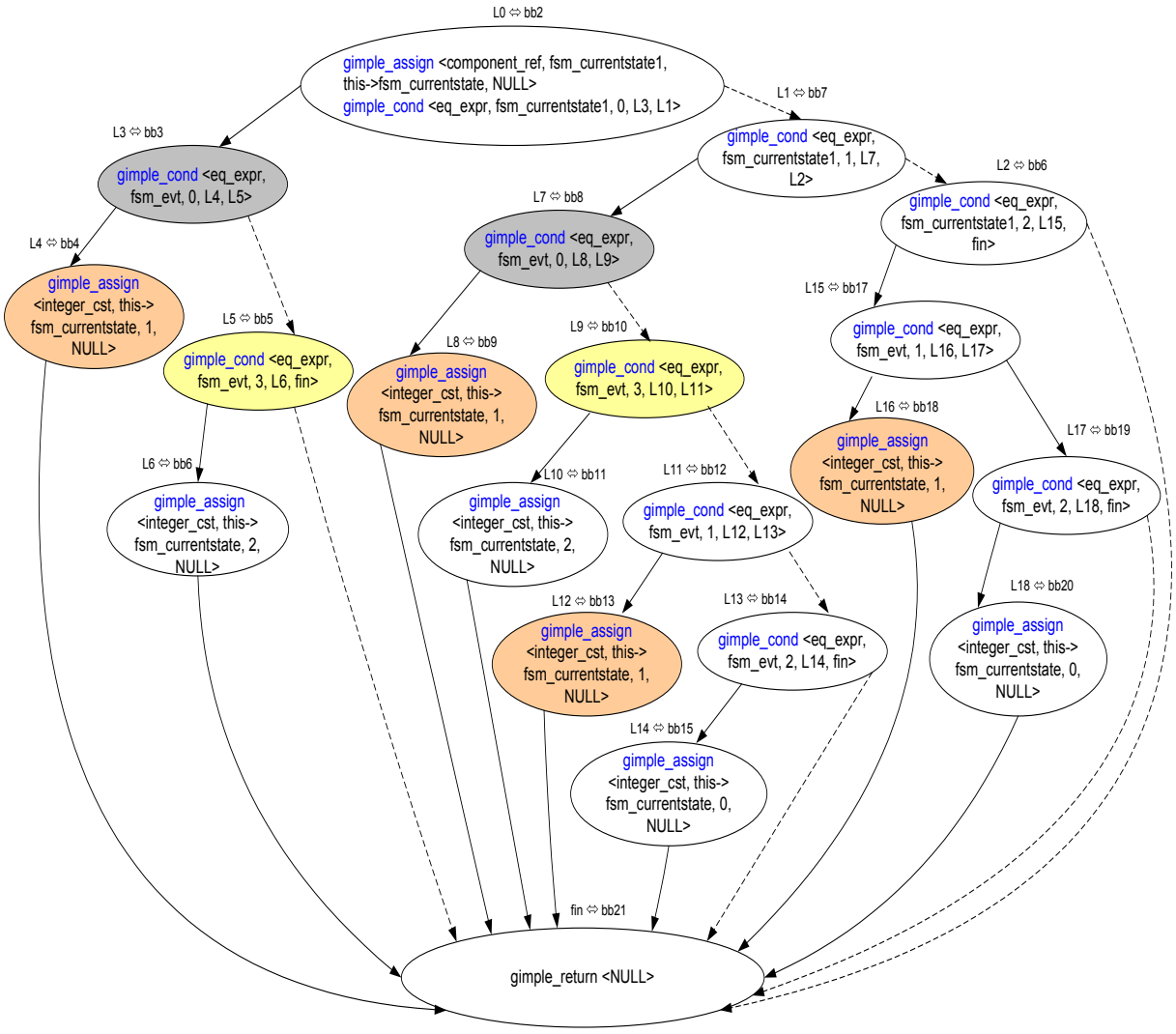


FIGURE 5.8 – Le GFC généré par GUML à partir de la forme GIMPLE de la figure 5.4

Ce graphe généré par GCC nous rappelle la forme S-Graph présenté dans le chapitre état de l'art (section 2.2.2.2). Ainsi l'*élimination des expressions redondantes* liées au changement de la valeur de l'état courant (par exemple les blocs de bases `bb9`, `bb13` et `bb18` qui partagent la même expression que le bloc de base `bb4`) correspond à l'optimisation de l'*élimination des nœuds isomorphes* du S-Graph. L'*élimination des expressions redondantes* qui testent la valeur de l'événement reçu (par exemple le bloc de base `bb8` qui partagent la même expression que le bloc de base `bb3`) n'est pas prise en compte dans GCC. Cette optimisation correspond à celle de l'*élimination des nœuds semi-isomorphes* (les définitions des nœuds *isomorphes* et *semi-isomorphes* se trouvent dans la section 2.2.2.2). A titre d'exemple, les deux blocs de bases `bb3` et `bb8` correspondent à deux nœuds *semi-isomorphes*. La présence à deux reprises de l'expression «`cmpl $0, %esi`» correspondante à l'instruction C++ «`if (evt ==e1)`» dans le fichier assembleur de la figure 5.5 montre que GCC était incapable

d'éliminer le bloc de base `bb8`. Nous constatons ainsi que l'optimisation d'*élimination de code redondant* du compilateur GCC consiste à éliminer uniquement les nœuds *isomorphes*. Ce qui explique la présence des nœuds *semi-isomorphes* dans la version optimisée du graphe de flot de contrôle qu'il est possible de déduire à partir du fichier assembleur (figure 5.5). Cette figure montre que GCC était dans l'incapacité d'éliminer tous les nœuds *semi-isomorphes*. La présence à deux reprises de l'expression «`cmpl $3, %esi`» correspondante à l'instruction C++ «`if (evt ==e4)`» montre que GCC n'a pas éliminé le bloc de base `bb10` isomorphe au bloc `bb5`).

La solution que nous proposons pour éliminer tout le code redondant présent dans un GFC est de produire la forme S-Graph à partir de la machine à états UML, optimiser cette forme en éliminant tous les nœuds *isomorphes* et *semi-isomorphes* et finalement générer la forme GIMPE optimisée. Ainsi, le GFC produit par GCC ne contient plus de code redondant. La figure 5.9 présente le S-Graph original produit à partir de la machine à états UML de la figuree 5.1 ainsi que le S-Graph optimisé après l'élimination des nœuds *isomorphes* et *semi-isomorphes*.

En analysant le S-Graph optimisé (Figure 5.9(b)), nous pouvons facilement déduire que le GFC de la forme GIMPLE correspondante contiendra une seule fois chacune des deux instructions de test suivantes : `if(evt==e2)` et `if(evt==e3)`. Par contre, pour les deux autres instructions de test de la valeur de l'événement reçu : `if(evt==e1)` et `if(evt==e4)`, nous notons la présence de deux nœuds pour chaque test (les deux couples de nœuds `(L3, L7)` et `(L5, L9)`). Par la suite, ces deux instructions de tests seront présentes à deux reprises dans la forme GIMPLE, dans le GFC correspondant et même dans le fichier assembleur final. Ainsi, les optimisations définies par [11] et [64] pour la production d'un S-Graph optimisé ne sont pas suffisantes pour éliminer tous les nœuds qui représentent des calculs redondants. Le nœud `L7`, bien que représentant la même instruction de test que le nœud `L3`, n'a pas été éliminé vu que les nœuds `L3` et `L7` ne sont pas *semi-isomorphes*. En effet, le nœud `L11` de la *false_branche* du nœud `L7` possède deux parents ; une condition nécessaire pour l'identification des nœuds *semi-isomorphes* est que tous les nœuds qui forment le chemin des deux branches (*true* et *false*) jusqu'au nœud final ne possède qu'un seul parent (section 2.2.2.2).

Bien que l'optimisation d'*élimination des nœuds semi-isomorphes* telle qu'elle a été définie dans [64] ait amélioré le pouvoir d'élimination de code redondant du compilateur GCC, cette optimisation est incapable d'éliminer tout le code redondant. Ainsi, afin d'éliminer tout le code redondant et produire un GFC optimisé à partir des machines à états UML, nous proposons d'ajouter une nouvelle forme intermédiaire dans la chaine de transformation de UML→GIMPLE. Sur cette nouvelle forme, des optimisations de haut niveau qui assurent l'élimination de tout code redondant seront implémentées. Cette forme, inspirée de la forme S-graph, représente le graphe de flot de contrôle de la forme GIMPLE. Ainsi, nous l'avons nommée : G-Graph (GIMPLE-Graph). G-Graph est une forme évoluée de S-Graph qui permet d'éliminer tout le code redondant présent dans la forme S-Graph et ceci en définissant deux types de nœuds de test. Si S-Graph définit 2 types de nœuds : les nœuds d'affectation (*Assign Nodes*) et les nœuds de test (*Test Nodes*), la forme G-Graph définit en plus des nœuds d'affection, deux types de nœuds de test (*State_Cond Nodes*) et (*Event_Cond Nodes*). La figure 5.10 présente le méta-modèle de la forme G-Graph.

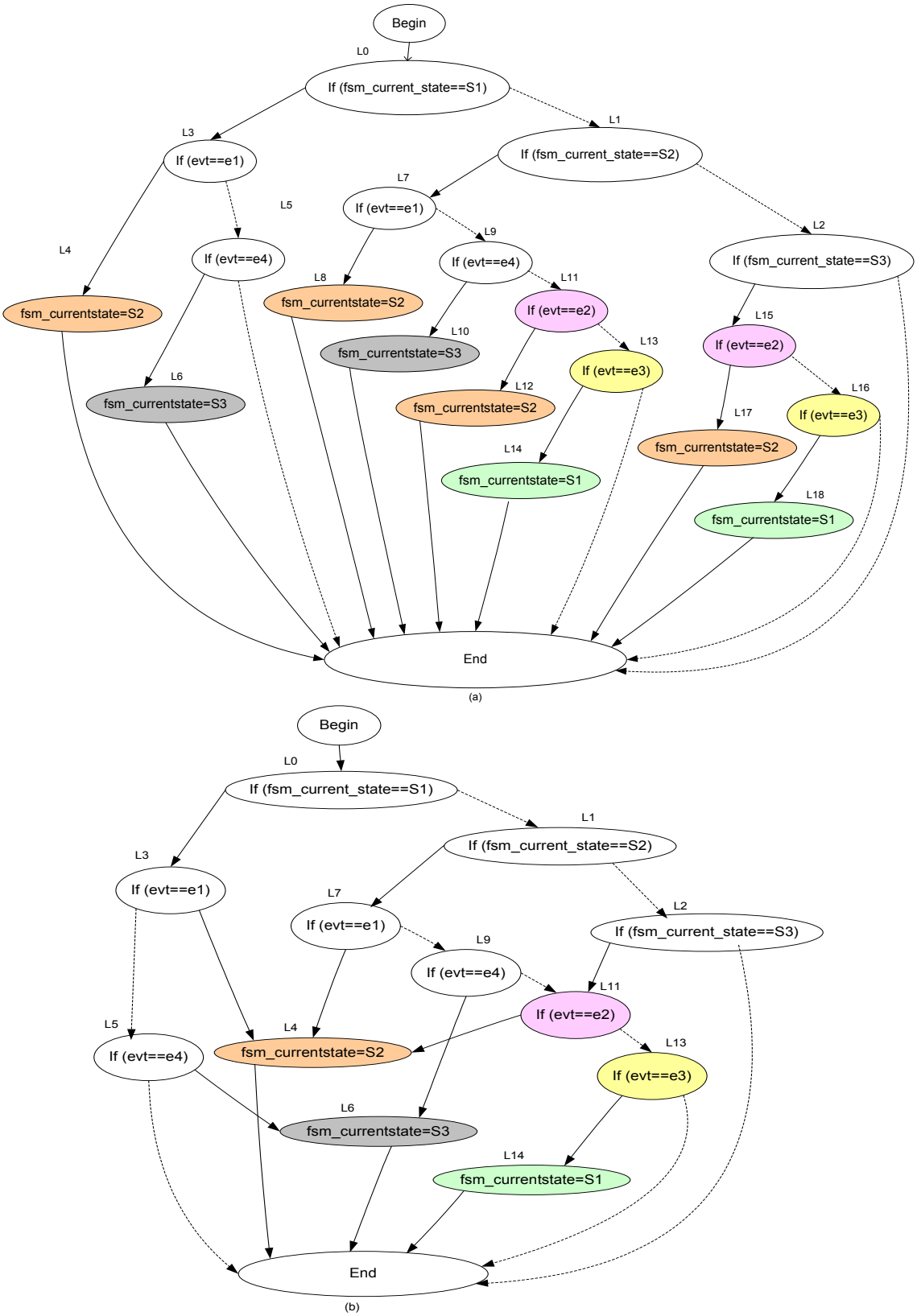


FIGURE 5.9 – (a) La forme S-graph produite directement à partir de la figure 5.1 (b) la forme S-graph optimisée

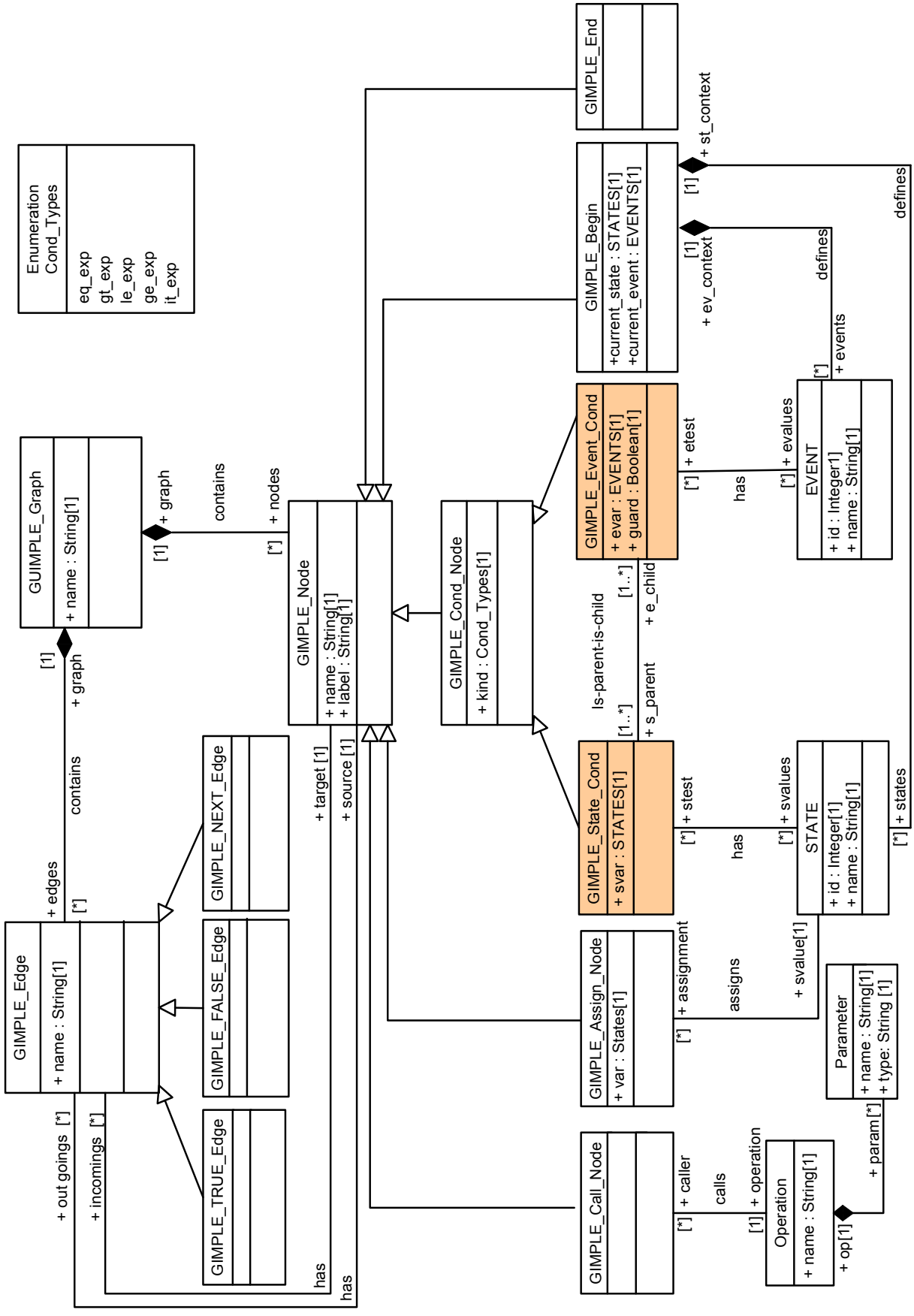


FIGURE 5.10 – Le méta-modèle de la forme G-Graph : 2 types de nœuds de Test GIMPLE_State_Cond et GIMPLE_Event_Cond

Ce méta-modèle a été utilisé pour implémenter la transformation des machines à états UML vers leurs formes G-Graph équivalentes. Cette transformation de modèles est implémentée sous forme de plugin Eclipse nommé « SM2GGRAPH ». La figure 5.11 montre la représentation graphique de la forme G-Graph produite à partir de la machine à états UML de la figure 5.1 en utilisant la transformation « SM2GGRAPH ».

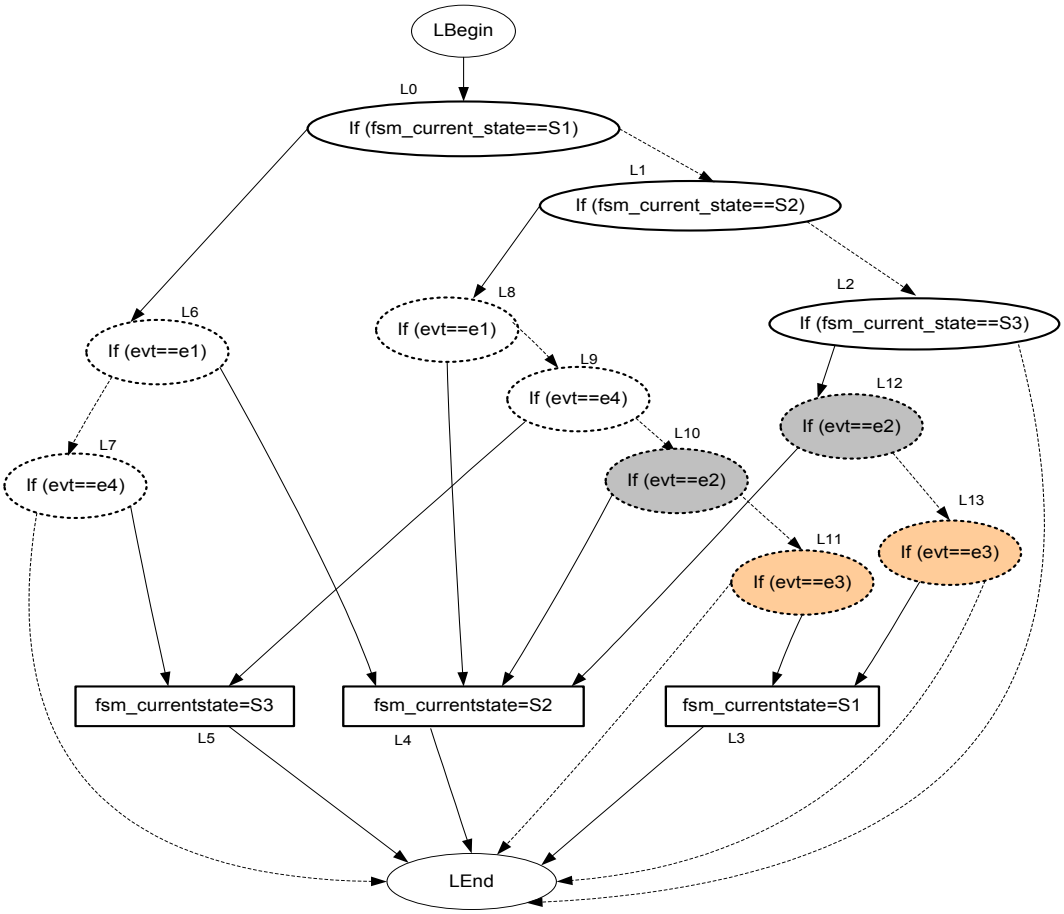


FIGURE 5.11 – La forme G-GRAPH généré à partir de la machine à états UML de la figure 21

Cette forme contient 16 nœuds différents : 2 pseudo nœuds (L*Begin* et L*end*), 3 nœuds de test de la valeur de l'état courant (`L0`, `L1` et `L2`), 3 nœuds de changement de valeur de l'état courant (`L3`, `L4` et `L5`) et 8 nœuds de test de la valeur de l'événement reçu (de `L6` à `L13`). Nous pouvons remarquer que contrairement à la forme S-Graph produite à partir de la même machine à états UML (Figure 5.9(a)) en utilisant l'algorithme de construction de S-Graph décrit dans l'approche Polis [11], la représentation G-Graph ne contient pas de nœuds d'affectation (*Assign Nodes*) *isomorphes*. En effet, dans notre algorithme de construction de la forme G-Graph, les premiers nœuds à ajouter dans le graphe, après les pseudo-nœuds sont les nœuds d'affectation. Vu que tous les états de la machine à états UML sont atteignables (la suppression des états inatteignables est effectuée dans le premier niveau d'optimisation de GUML), un *seul* nœud d'affectation est créé pour chaque état présent dans la machine à états UML. Il est à noter que dans [64], l'auteur a choisi de produire directement la forme S-Graph optimisée (Figure 5.9(b)) à partir de la machine à états de la figure 5.1. Ainsi, il a modifié l'algorithme de construction des S-Graph défini dans

[11] en testant juste avant la création d'un nouveau nœud dans le graphe, si un nœud lui étant *isomorphe* existe dans ce graphe. Pour ne pas compliquer l'implémentation de l'outil « SM2GGRPAH », nous avons choisi d'implémenter les optimisations dans un autre plugin Eclipse nommée « GGRAPH_OPTIMIZER ». Mise à part l'élimination des nœuds *isomorphes* (autres que les nœuds de type *Assign_Node*) et *semi-isomorphes*, une nouvelle optimisation qui consiste à combiner les nœuds de test de même type «*Factorisation des nœuds de test*» est implémentée dans la forme G-Graph. Le résultat de l'exécution des optimisations de « GGRAPH_OPTIMIZER » sur la forme G-Graph, l'apport de la nouvelle optimisation *Factorisation des nœuds de test* et son impact sur les autres formes intermédiaires du compilateur GUML seront étudiés dans la section suivante.

5.3 Optimisations du niveau G-Graph

L'objectif de l'outil d'optimisation « GGRAPH_OPTIMIZER » est l'élimination des expressions redondantes présentes dans la forme G-Graph produite à partir des machines à états UML. Pour ce faire 3 optimisations différentes ont étés implémentées. La première consiste à éliminer les nœuds *isomorphes* du graphe de départ, la deuxième élimine les nœuds *semi-isomorphes* et la troisième combine les nœuds de test de même type.

Dans ce qui suit, nous considérons la forme G-Graph de la figure 5.11 produite directement à partir de la machine à états UML de la figure 5.1.

5.3.1 Élimination des nœuds isomorphes

Cette transformation s'appuie sur l'algorithme 1. La première phase de cet algorithme consiste à détecter tous les nœuds *isomorphes* selon la définition donnée par [11] : Deux nœuds sont *isomorphes* s'ils ont la même étiquette (même expression) et que leurs descendants sont aussi *isomorphes*. Par construction, la forme G-Graph produite ne contient pas de nœuds *isomorphes* de type *Assign*, ni de nœuds *isomorphes* de types *State_Cond*. Ainsi, l'algorithme 1 cherche les nœuds *isomorphes* uniquement dans l'ensemble des nœuds de test de type *Event_Cond* [1]. L'ensemble des nœuds *isomorphes* trouvés sont stockés dans la liste L_{iso}. Chaque élément de cette liste est un `p-uplet` qui contient un nœud et les `p-1` nœuds qui lui sont *isomorphes* (p>1 : au moins un nœud isomorphe doit exister). La deuxième phase de cet algorithme consiste à éliminer pour chaque élément de la liste L_{iso} les `p-1` nœuds *isomorphes*. Avant de supprimer un nœud, ses liens (de type *Gimple_Edge*) avec les autres nœuds existants sont supprimés : les arcs entrants sont redirigés vers le nœud à conserver (le premier élément du `p-uplet`) et les arcs sortants sont supprimés. Ainsi, le nœud à conserver possède au moins 2 arcs entrants (chaque élément du `p-uplet` à supprimer entraîne l'ajout d'un arc entrant au nœud à conserver). L'ajout d'un nouvel arc entrant entraîne à son tour l'ajout d'une nouvelle valeur (l'état parent de l'élément à supprimer) à la propriété *S_parent* du nœud de test de type *Event_Cond*. La spécialisation de deux types différents de nœuds de test : *Event_Cond* et *State_Cond* permet d'exprimer clairement la relation entre ces nœuds en gardant une trace de l'état parent (S_parent).

1. Pour les nœuds de type *Call* (Figure 5.10) , utilisés uniquement pour appeler des fonctions (effets de transitions, les actions des états, etc.) le même algorithme est utilisé

Algorithme 1: Algorithme de l'optimisation : élimination des nœuds *isomorphes*

Entrées :
L_{ev} : l'ensemble de tous les nœuds de type *Event_Cond* présents dans G-Graph
Résultat :
Détecter et supprimer tous les nœuds isomorphes de type *Event_Cond* trouvés dans G-Graph.
Données :
L_{iso} : la liste des nœuds isomorphes présents dans la forme G-Graph. Chaque élément de cette liste est lui-même de type liste : il contient un premier nœud et tous les autres nœuds isomorphes à lui.
L_{boo} : une liste de booléen de même taille que la liste L_{ev}. Chaque élément (nœud) de la liste L_{ev} lui correspond une valeur booléenne qui indique si l'élément a été déjà testé ou pas encore.
Initialisations :
$L_{iso} \longleftarrow \emptyset$;
pour *i allant de 0* **à** L_{ev}*.size()* **faire** L_{boo}*(i)* = faux;
Phase 1 : détecter les nœuds isomorphes et les placer dans la liste L_{iso}
début
  pour *i allant de 0* **à** L_{ev}*.size()* **faire**
    si L_{boo}*.get(i) == faux* **alors**
      L_{new}.add(L_{ev}(i));
      pour *j allant de i+1* **à** L_{ev}*.size()* **faire**
        si *is_isomorphic*(L_{ev}*(i),* L_{ev}*(j))* **alors**
          L_{new}.add(L_{ev}.get(j));
          L_{boo}*(j)=vrai*;
        fin
      fin
      si L_{new}*.size() > 1* **alors**
        k++;
        L_{iso}*.add(k,* L_{new}*)*;
      fin
    fin
  fin
  retourner L_{iso};
fin
Phase 2 : parcourir les éléments de la liste L_{iso}. Pour chaque élément (de type liste de nœuds), garder le premier nœud et supprimer les autres nœuds isomorphes à lui. Commencer par éliminer les nœuds isomorphes les plus proches du nœud End (parcourir la liste dans l'ordre décroissant).
début
  pour *j allant de* L_{iso}*.size()-1* **à** *0* **faire**
    $L_{current}$ = L_{iso} *.get(j);*
    node_to_keep = $L_{current}$*.get(0) ; //garder le premier élément de la liste*
    pour *i allant de 1* **à** $L_{current}$*.size()* **faire**
      node_to_delete = $L_{current}$*.get(i);*
      incoming_edges = node_to_delete.getIncomings();
      pour chaque $e \in incoming\_edges$ **faire**
        e.setTarget(node_to_keep);
      fin
      node_to_keep.getS_parent().addAll(node_to_delete.getS_parent());
      delete(get_true_edge(node_to_delete));
      delete(get_false_edge(node_to_delete));
      delete(node_to_delete);
    fin
  fin
fin

La figure 5.12 présente la forme G-Graph optimisée ne contenant pas de nœuds *isomorphes*. Le nombre de nœuds est ainsi passé de 16 à 14.

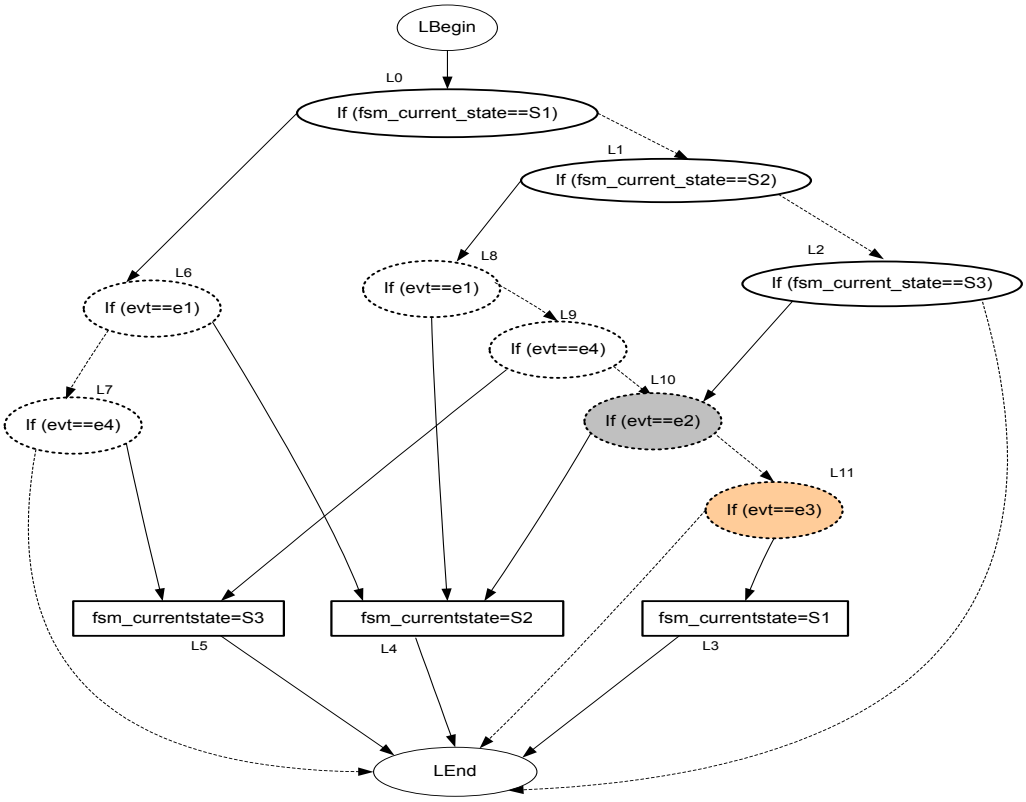


FIGURE 5.12 – La forme G-Graph réduite : résultat de l'exécution de l'optimisation élimination des nœuds *isomorphes*

Cette optimisation diminue également le nombre de blocs de base dans le GFC produit à partir de la forme GIMPLE générée. Ainsi, les deux blocs de bases (`bb17` et `bb19`) de la figure 5.8 ne vont pas être générés puisqu'ils correspondent aux deux nœuds `L12` et `L13` de la forme G-Graph de la figure 5.11. Ces deux nœuds ont été éliminés suite à l'exécution de l'optimisation de l'élimination des nœuds *isomorphes* (figure 5.12). L'impact de cette optimisation dépend du nombre de nœuds *isomorphes* présents dans la forme G-Graph. Ce nombre varie selon l'ordre de construction des nœuds de test de type *Event_Cond*. En effet, un simple changement de l'ordre de construction de ces nœuds de test peut changer le nombre de nœuds *isomorphes* présents dans la forme G-Graph et par la suite changer l'impact de cette optimisation sur la forme G-Graph.

La figure 5.13 présente la même forme G-Graph de la figure 5.11 à l'exception du changement de l'ordre des deux nœuds de test `L12` et `L13` (tester si l'événement reçu est égal tout d'abord à `e3` et par la suite à `e2` contrairement à l'ordre établi par la figure 5.11 qui teste l'égalité de la valeur de l'événement par rapport à `e2` puis à `e3`). Nous pouvons remarquer que la figure 5.13 (contrairement à la figure 5.11) ne contient pas de nœuds *isomorphes*. Ainsi, l'optimisation de *l'élimination de nœuds isomorphes* n'a aucun effet sur cette forme. Bien que les deux nœuds `L11` et `L12` ont la même expression (`if (evt==e3)`) et leurs descendants issus de leurs *true_branches* sont identiques, ces deux nœuds ne sont pas *isomorphes*. Leurs descendants issus de leurs *false_branches* sont en effet différents.

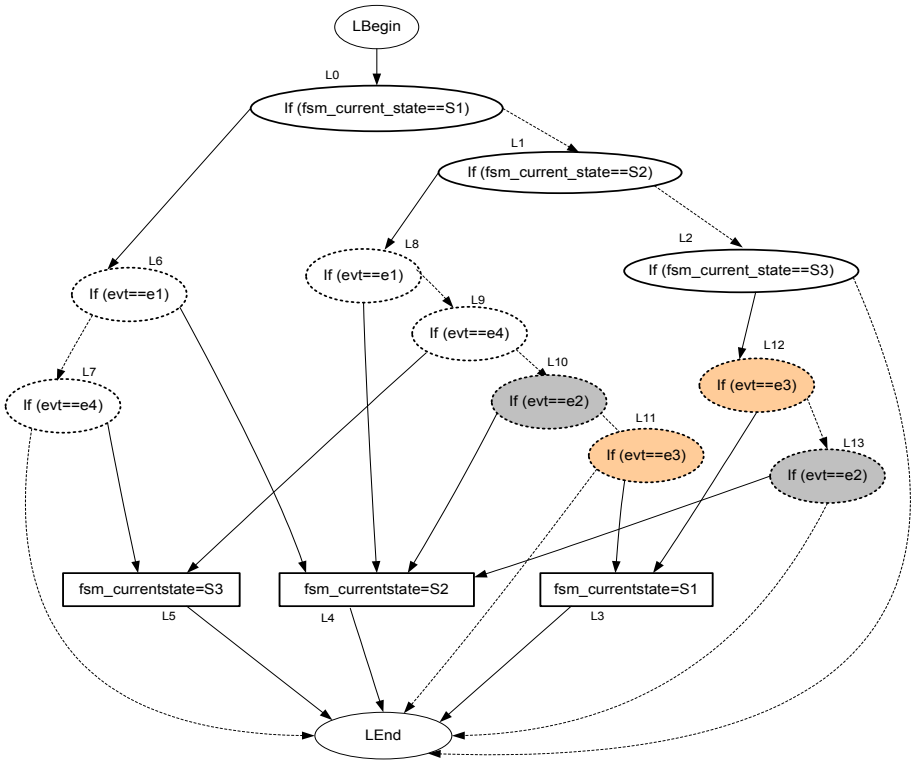


FIGURE 5.13 – La forme G-Graph contenant des neouds *semi-isomorphes* : (L10, L13) et (L11, L12)

Pour pouvoir éliminer l'expression redondante (`if (evt==e3)`) engendrée par la présence des deux nœuds L11 et L12, [64] a définit la notion de nœuds de test *semi-isomorphes* : Deux nœuds de test sont *semi-isomorphes* s'ils ont la même expression, les mêmes *true_branches* mais deux false_branches différentes. Les nœuds formant les *false_branches* doivent avoir exactement un seul parent

Ainsi, *l'élimination des nœuds semi-isomorphes* permettra d'éliminer les expressions redondantes que la première optimisation (*élimination des nœuds isomorphes*) est incapable d'éliminer à cause du changement d'ordre de construction des nœuds de test de type *Event_Cond*. Nous allons voir dans la section suivante que grâce à cette nouvelle optimisation (*élimination des nœuds semi-isomorphes*), la même forme G-Graph optimisée (figure 5.12) pourra être produite à partir de la forme G-Graph de la figure 5.13.

5.3.2 Élimination des nœuds *semi-isomorphes*

Cette transformation s'appuie sur l'algorithme défini dans Algorithme 2. La première phase de cet algorithme consiste à détecter tous les nœuds *semi-isomorphes*. De la même manière que l'algorithme 1, l'algorithme 2 cherche les nœuds *semi-isomorphes* uniquement dans l'ensemble des nœuds de test de type *Event_Cond*. L'ensemble des nœuds *semi-isomorphes* trouvés sont stockés dans la liste $L_{semiiso}$. Chaque élément de cette liste est un `p-uplet` qui contient un nœud et les `p-1` nœuds qui lui sont *semi-isomorphes*. La deuxième phase de cet algorithme consiste à éliminer pour chaque élément de la liste $L_{semiiso}$ les `p-1` nœuds *semi-isomorphes*.

Algorithme 2: Algorithme de l'optimisation : élimination des nœuds semi-isomorphes

Entrées :
L_{ev} : l'ensemble de tous les nœuds de type *Event_Cond* présents dans G-Graph
Résultat :
Détecter et supprimer tous les nœuds *semi-isomorphes* de type *Event_Cond* trouvés dans G-Graph.
Données :
$L_{semiiso}$: la liste des nœuds textitsemi-isomorphes présents dans la forme G-Graph. Chaque élément de cette liste est lui-même de type liste : il contient un premier nœud et tous les autres nœuds *semi-isomorphes* à lui.
L_{boo} : une liste de booléenne de même taille que la liste L_{ev}. Chaque élément (nœud) de la liste L_{ev} lui correspond une valeur booléenne qui indique si l'élément a été déjà testé ou pas encore.
Initialisations :
$L_{semiiso} \longleftarrow \emptyset$;
pour *i allant de 0* **à** L_{ev}*.size()* **faire** L_{boo}*(i) = faux ;*
Phase 1 : détecter les nœuds *semi-isomorphes* et les placer dans la liste L_{iso}
début

```
pour i allant de 0 à L_ev.size() faire
    si L_boo.get(i) == faux alors
        L_new.add(L_ev(i));
        pour j allant de i+1 à L_ev.size() faire
            si is_semiisomorphic(L_ev.get(i), L_ev.get(j)) alors
                L_new.add(L_ev.get(j));
                L_boo(j)= vrai;
            fin
        fin
        si L_new.size() > 1 alors
            k++;
            L_semiiso.add(k, L_new);
        fin
    fin
fin
retourner L_semiiso;
```

fin
Phase 2 : Parcourir les éléments de la liste $L_{semiiso}$. Pour chaque élément (de type liste de nœuds), faire descendre tous les nœuds pour les rendre isomorphes. Faire ensuite appel à l'algorithme 1 pour détecter et supprimer les nœuds isomorphes. Refaire cette phase jusqu'à ce que la liste $L_{semiiso}$ soit vide.
début

```
tant que L_semiiso ≠ ∅ faire
    L_current = L_semiiso.get(L_semiiso.size()-1);
    pour j allant de 0 à L_current.size()-1 faire
        down_node(L_current.get(j)) ; // rendre isomorphes les neouds semi_isomorphes
    fin
    L_iso = get_all_isom_nodes(L_ev) ; //Phase 1 de l'Algorithme 1
    si L_iso ≠ ∅ alors
        remove_iso_nodes(L_iso) ; //Phase 2 de l'Algorithme 1
    fin
    L_semiiso = get_all_semi_isom_nodes(L_ev) ; //Phase 1 de l'Algorithme 2
fin
```

fin

Si l'élimination des nœuds *isomorphes* consiste simplement à supprimer les nœuds ainsi que leurs arcs sortants de la forme G-Graph après avoir changé la cible des arcs entrants du nœud à supprimer et ajouter la valeur de son état parent à la propriété *S_parent* du nœud à garder, l'élimination des nœuds *semi-isomorphes* est un peu plus compliquée. Elle requiert une transformation de la forme G-Graph de telle manière à rendre les nœuds *semi-isomorphes* des nœuds *isomorphes*, puis à faire appel à la même fonction *remove_iso_nodes* implémentée dans la première optimisation (Phase 2 de l'algorithme 1).

La différence entre deux nœuds *isomorphes* et deux nœuds *semi-isomorphes* de type *Event_Cond* est le fait que les deux *false_branches* des nœuds *semi-isomorphes* sont différentes. Puisque l'ordre suivi pour faire les tests sur la valeur de l'événement reçu (ordre d'ajout des nœuds de type *Event_Cond* dans la forme G-Graph) n'a pas d'impact sur le comportement de la machine à états UML représentée par la forme G-Graph, ajouter les nœuds *semi-isomorphes* de type *Event_Cond* à la fin (leurs *false_branches* seront liées au dernier nœud End) ne changera pas le comportement de la machine à états UML. Ainsi, ces deux nœuds *semi-isomorphes* auront les mêmes *falses_branches* et par la suite deviennent *isomorphes*. La suppression des nœuds *semi-isomorphes* consiste alors à les faire descendre jusqu'à atteindre le nœud final (End), supprimer les nœuds *isomorphes* récemment ajoutés et détecter de nouveau les nœuds *semi-isomorphes* et ceci jusqu'à ce que cette transformation de la forme G-Graph n'entraîne plus l'apparition de nouveaux nœuds *semi-isomorphes* (Algorithme 2).

A titre d'exemple, la descente des deux nœuds *semi-isomorphes* `L11` et `L12` de la figure 5.13 entraîne l'apparition de deux nouveaux couples de nœuds *isomorphes* `(L11, L12)` et `(L10, L13)`. L'exécution de l'optimisation *d'élimination des nœuds semi-isomorphes* produit la même forme G-Graph optimisée de la figure 5.12 (contenant uniquement 14 nœuds).

Lors de la construction de la forme G-Graph, nous avons choisi d'ajouter les nœuds de test de type *Event_Cond* d'une manière à suivre l'ordre de l'apparition des événements déclenchant les transitions de la machine à états et ceci pour favoriser l'obtention d'un maximum de nœuds *isomorphes*. Par la suite, nous obtiendrons la forme G-Graph de la figure 5.11 qui contient directement deux couples de nœuds *isomorphes* `(L10, L12)` et `(L11, L13)` et dont l'exécution de la première passe d'optimisation (élimination des nœuds *isomorphes*) produit la forme G-Graph réduite de la figure 5.12 (ne contenant pas les deux expressions redondantes : ≪`if (evt==e3)`≫ et ≪`if (evt==e2)`≫.

Cependant, cette forme réduite est encore source d'optimisation puisqu'elle contient d'autres nœuds susceptibles d'être transformés en expressions redondantes. En effet, les deux nœuds `L6` et `L8` ont la même expression (`if (evt==e1)`), les mêmes *true_branches* et des *false_branches* différentes. De même pour les deux nœuds `L7` et `L9` qui partagent la même expression (`if (evt==e4)`). Les deux nœuds de chacun de ces deux couples `(L6, L8)` et `(L7, L9)` ne sont pas *semi-isomorphes* puisque le nœud `L10`, qui fait partie des deux *false_branches* des nœuds `L8` et `L9` possède deux parents (Figure 5.12). Ainsi, l'exécution des deux optimisations définies par [11] et [64] n'élimine pas les expressions redondantes des deux nœuds `L8` et `L9`. La présence du deuxième parent du nœud `L10` a empêché la deuxième passe d'optimisation (*élimination des nœuds semi-isomorphes*) d'optimiser encore plus la forme G-Graph. Cette optimisation a été définie par [64] sur la forme S-Graph qui ne fait pas la différence entre les nœuds de test de type *Event_Cond* et *State_Cond*. C'est cette spécialisation des nœuds de test introduite dans la forme G-Graph qui va permettre de réduire encore plus

le nombre de nœuds de la forme G-Graph de la figure 5.12 et éliminer par la suite toutes les expressions redondantes.

Si [64] a amélioré les optimisations de la forme S-Graph en introduisant la notion de nœuds *semi-isomorphes*, dans ce travail de recherche, nous améliorons encore plus les optimisations de la forme S-Graph en définissant une nouvelle optimisation (*Factorisation des nœuds de test*) qui consiste à dupliquer puis factoriser des nœuds de test de type *State_Cond*.

5.3.3 Factorisation des nœuds de test

Cette transformation de la forme G-Graph élimine les nœuds *semi-isomorphes* même en présence d'un nœud de la branche *false_branche* possédant deux parents de type différent (un de type *Event_Cond* et un de type *State_Cond* (exemple du nœud `L10` de la figure 5.12)). L'idée consiste à détecter le nœud de la *false_branche* posant problème et de l'écarter de cette branche. Cette transformation du G-Graph ne modifie pas le comportement initial de la machine à états de départ (une vérification de la conservation du comportement initial du G-Graph après la transformation est présentée en Annexe C). L'algorithme 3 décrit le déroulement de cette transformation. La première étape consiste à détecter le premier nœud (*n_current*) de la liste $L_{not-semi-iso}$ ainsi que le premier nœud (*n_diff*) de sa *false_branche* possédant deux parents différents (si les parents sont tous deux de type *State_Cond* une factorisation directe de ces deux nœuds parents est possible en utilisant l'opérateur logique OU). Le nœud (ou les nœuds) de test de type *State_Cond* (référencé par la propriété `S_parent`) origine du nœud *n_diff* est dupliqué pour le lier de nouveau (mais cette fois-ci directement) au nœud *n_diff* écarté de la *false_branche* de tous ses nœuds entrant de type *Event_Cond*en liant ses arcs entrants au nœud dupliqué. Cette phase de duplication est temporaire puisqu'elle est directement suivie par un mécanisme de factorisation des nœuds de test de type *State_Cond* (l'ancien nœud et le ou les nouveaux nœuds ajoutés par duplication) en utilisant l'opérateur logique OR. Suite à cette transformation, de nouveaux nœuds *isomorphes* ou *semi-isomorphes* apparaissent et seront éliminées en exécutant les deux premières optimisations (*élimination des nœuds isomorphes* et *élimination des nœuds semi-isomorphes*). De la même manière, l'élimination des nœuds *isomorphes* ou *semi-isomorphes* peut entraîner l'apparition de nouveaux nœuds à deux parents de types différents dans la forme G-Graph qui peut être optimisée d'avantage en exécutant de nouveau l'optimisation de la *factorisation des nœuds de test*. Cette séquence d'appels est exécutée tant que la liste $L_{not-semi-iso}$ n'est pas vide (Algorithme 3). On note que grâce à la distinction des deux types State_Cond et Event_Cond, nous considérons une nouvelles définition des nœuds *isomorphes* de type Event_Cond. En effet, lier la false_branche d'un nœud de type Event_Cond au nœud final (End) exprime le fait qu'il n'existe pas d'autres événements déclenchant des transitions sortantes du nœud parent de type State_Cond. De la même manière, lier la *false_branche* d'un autre nœud de type Event_Cond à un nœud de type State_Cond exprime la même chose (à savoir le passage vers un autre test sur la valeur de l'état courant vu qu'il n'existe pas d'autres événements déclenchant des transitions sortantes du nœud parent de type State_Cond. Ainsi, nous considérons ces deux *false_branche* comme équivalentes et nous considérons ces deux nœuds de type Event_Cond comme isomorphes s'ils possèdent mise à part deux *false_branches* équivalentes, des *true_branches* équivalentes et une même expression.

Algorithme 3: Algorithme de l'optimisation : factorisation des nœuds de test

Entrées :
L_{ev} : l'ensemble de tous les nœuds de type *Event_Cond* présents dans G-Graph
$L_{not-semi-iso}$: l'ensemble des nœuds qui n'ont pas pu être éliminés par Algortihme 2 à cause de la présence dans leurs *false_branches* d'un nœud de test ayant plus qu'un seul parent.
Résultat :
Eliminer les nœuds de la liste $L_{not-semi-iso}$.
Données :
$L_{devided}$: une liste à deux éléments, le premier contient les nœuds parents de type *State_Cond* et le deuxième les nœuds parents de type *Event_Cond* du nœud de test ayant plus qu'un seul parent.
$L_{collapse}$: liste de nœuds de type *State_Cond* à factoriser.
Initialisations :
$L_{devided} \longleftarrow \emptyset$; $L_{collapse} \longleftarrow \emptyset$; $L_{devided} \longleftarrow \emptyset$;
début
  tant que $L_{not-semi-iso} \neq \emptyset$ **faire**
    current_n = $L_{not-semi-iso}$.*get(0)*;
    détecter le premier nœud *n_diff* à deux parents différents dans la *false_branche* de *current_n*
    séparer les parents de *n_diff* selon le type (State_Cond et Event_Cond) dans la liste $L_{devided}$
    si $L_{devided}$.*size() > 1* **alors**
      si $L_{devided}.get(0) \neq \emptyset$ **alors**
        pour *k* ***allant de*** *0* **à** $L_{devided}$.*size()-1* **faire**
          the_child = ($L_{devided}$*(0))(k)*;
          the_parent = *the_child.getS_parent().get(0)*;
          duplicate_node(the_parent, L_{ev}*(i))*;
          $L_{collapse}$.*addAll(get_S_incomings(n_diff))*;
          get_false_edge(the_child).setTarget($L_{collapse}$*(0))*;
        fin
      sinon
        $L_{collapse}$.*addAll(get_S_incomings(n_diff))*;
      fin
    fin
    si $L_{collapse}$.*size() > 1* **alors**
      node_to_keep = $L_{collapse}$*(0)*;
      pour *l* ***allant de*** *1* **à** $L_{collapse}$.*size()-1* **faire**
        node_to_delete = $L_{collapse}$*(l)*;
        $Lincomings$ = *node_to_delete.getIncomings()*;
        pour *m* ***allant de*** *0* **à** $L_{collapse}$.*size()-1* **faire**
          $Lincomings$*(m).setTarget(node_to_keep)*;
        fin
        node_to_keep.getSvalues().addAll(node_to_delete.getSvalues());
        supprimer(get_true_edge(node_to_delete));
        supprimer(get_false_edge(node_to_delete));
        supprimer(node_to_delete);
      fin
    fin
    L_{iso} = *get_all_isom_n(*L_{ev}*)* ; *//Phase 1 de l'Algorithme 1*
    si $L_{iso} \neq \emptyset$ **alors** *remove_iso_n(*L_{iso}*)* ; *// Phase 2 de l'Algorithme 1*
    $L_{semiiso}$ = *get_all_semi_isom_nodes(*L_{ev} *)* ; *//Phase 1 de l'Algorithme 2*
    si $L_{semiiso} \neq \emptyset$ **alors** *remove_semi_iso_nodes(*$L_{semiiso}$*)* ; *// Phase 2 de l'Algorithme 2*
    $L_{not-semi-iso}$ = *get_not_semi_isom_nodes(*L_{ev}*)*;
  fin
fin

La complexité (dans le pire cas) des 3 algorithmes présentés dans cette section est polynomiale (`O(n^p)`) avec `p>0` et `n` représente le nombre de nœuds de type *Event_Cond* dans le G-Graph. Le nombre maximal de n est atteint dans le cas ou tous les états du G-Graph possèdent des transitions déclenchées par tous les événements (`nmax=`n_s `*` n_e avec n_s= nombre des états et n_e = nombre des événements). Le tableau 5.2 présente les caractéristiques des 3 algorithmes en termes de complexité et de terminaison (algorithme ne bouclant pas infiniment).

	Algorithme 1	**Algorithme 2**	**Algorithme 3**
Calcul de la Complexité	Phase1 : deux boucles imbriquées de n répétitions chacune → O(n^2) Phase2 : deux boucles imbriquées de n répétitions au total → O(n) **Complexité totale → O(n^2)**	Phase1 : deux boucles imbriquées de n répétitions chacune → O(n2) Phase2 : 3 boucles imbriqués → O(n3) (nombre max d'itérations de la boucle tant que est n/2) **Complexité totale → O(n3)**	4 boucles imbriqués → O(n4) nombre max d'itérations de la boucle tant que est n/2) **Complexité totale → O(n4)**
Absence de boucles infinies	Présence de 4 boucles dont le nombre de répétitions est connue à l'avance	Une seule boucle dont le nombre de répétitions est inconnu : la condition d'arrêt dépond de la taille de la liste $L_{semi\text{-}iso}$ qui atteindra forcément la valeur *nulle* (appel de *remove_iso_nodes*)	Une seule boucle dont le nombre de répétitions est inconnu : la condition d'arrêt dépond de la taille de la liste $L_{not\text{-}semi\text{-}iso}$ qui atteindra forcément la valeur *nulle* (appels d'algorithmes 1 et 2)

Tableau 5.2 – Les caractéristiques des 3 algorithmes en termes de complexité et de terminaison

Il ne s'agit pas d'une preuve par récurrences pour la terminaison ni d'un calcul détaillé de la complexité (qui peut être réduite pour l'algorithme 1 en évitant les boucles imbriqués, etc.). Toutefois, il est important de considérer ces caractéristiques et s'assurer que ces algorithmes répondent bien au but de l'utilisateur qui est la réduction du nombre des nœuds (une condition acquise pour algorithme 1 et 2 : on ne fait que diminuer le nombre de nœuds en appelant respectivement les méthodes *remove_iso_nodes* (L_{iso}) et *remove_semi_iso_nodes* ($L_{semiiso}$)). Pour l'algorithme 3, l'ajout d'un nœud de test de type *State_Cond* est immédiatement suivi d'une factorisation avec un autre nœud existant du même type. Algorithme 2 n'ajoute et supprime de nœuds que si et seulement si cette opération entraînera au moins l'apparition d'un couple de nœuds (*isomorphes* ou *semi-isomorphes*) qui à son tour entraînera la suppression d'au moins un nœud en évoquant les algorithmes 1 et 2.

Nous rappelons que le but de cette optimisation (*factorisation des nœuds de test*) est l'élimination des nœuds de type *Event_Cond* qui partagent la même expression mais qui ne sont ni *isomorphes*, ni *semi-isomorphes*. A titre d'exemple, les deux couples de nœuds `(L6, L8)` et `(L7, L9)` de la figure 5.12 forment deux couples de nœuds qui partagent la même expression (`if (evt==e1)` pour le premier couple et `if (evt==e4)` pour le deuxième), mais qui ne sont ni *isomorphes* ni *semi-isomorphes*. C'est en exécutant l'optimisation de *factorisation des nœuds de test* que les nœuds `L8` et `L9` sont éliminés. La figure 5.14(a) présente la forme G-Graph temporaire produite suite à la duplication du nœud de test `L1` (ajout du

nœud `L12`) et l'écartement du nœud de test `L10` qui avait deux parents de type différents (`L9` de type *Event_Cond* et `L2` de type *State_Cond*).

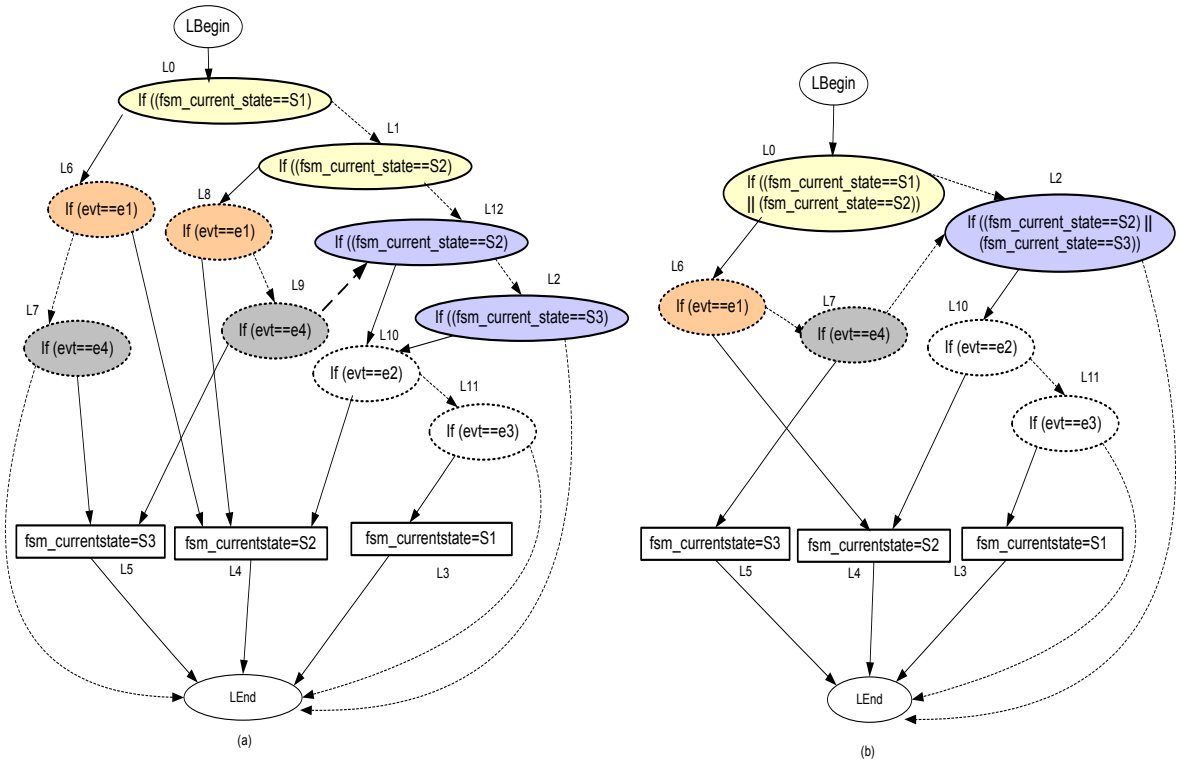


FIGURE 5.14 – (a) La forme G-Graph temporaire : la duplication du nœud de test L1 (b) La forme G-Graph optimisée : résultat de l'exécution de l'optimisation : factorisation de nœuds de test

La forme G-Graph de la figure 5.14 (b) est le résultat final de l'optimisation de l*a factorisation des nœuds de test*. Elle est produite suite à la factorisation des deux nœuds de test de même type (L12 et L2) en utilisant l'opérateur logique OU et l'exécution de l'optimisation de l'élimination des nœuds *isomorphes* qui a éliminé les nœuds L8 et L9. Cette optimisation est exécutée une seconde fois (algorithme 3) pour factoriser les deux nœuds L0 et L1 qui représentent les deux parents de même type (*State_Cond*) du nœud L6. Suite à cette transformation, la forme G-Graph ne contient plus d'expressions redondantes. Le nombre de nœuds de cette forme est passé de 14 (figure 5.12) à 11 (figure 5.14(b)). Il est à noter que la forme G-Graph produite directement à partir de la machine à état UML de la figure 5.1 contient 16 nœuds (figure 5.11). L'outil d'optimisation ≪GGRAPH_OPTIMIZER≫ a réduit progressivement ce nombre jusqu'à atteindre 11 nœuds. La diminution du nombre de nœuds entraîne la production d'une forme GIMPLE différente à partir de la quelle un GFC optimisé est produit. Par conséquent, la taille du fichier assembleur généré par le compilateur GUML en activant ce deuxième niveau d'optimisation (optimisation de la forme G-Graph) est plus compact que celle produite par G++. Le tableau 5.3 reprend les mesures effectuées auparavant (tableau 5.1) sur la taille du fichier assembleur produit par G++ et GUML en compilant la machine à états UML de la figure 5.1.

Ces mesures ont été obtenues en activant uniquement les optimisations du *middle-end* (niveau SSA) et du *back-end* (niveau RTL). Ces mesures ont donc changé en exécutant les nouvelles optimisations (les optimisations du niveau G-Graph) du *front-end* GUML. La taille du fichier assembleur a diminué d'environ 11% pour l'exemple de la machine à états UML de la figure 5.1 (5ème colonne du Tableau 5.3) et d'environ 20% par rapport à la taille du fichier assembleur produit par G++ à partir de la même machine à états UML.

Type de fichier	Taille (octets) du fichier produit par G++	Taille (octets) du fichier produit par GUML	Taille (octets) du fichier produit par GUML+ G-Graph optimisations	Taux d'optimisation	Gain par rapport à G++
Assembleur (.s)	2021	1815	1601	11,79 %	20,78%

Tableau 5.3 – Impact des optimisations du niveau G-Graph sur la taille du fichier assembleur généré par G++ et GUML à partir de la machine à états UML de la figure 5.1

5.3.4 Synthèse

Nous avons vu dans cette section que le compilateur GCC était incapable d'éliminer toutes les expressions redondantes présentes dans un fichier C++ généré à partir d'une machine à états UML. Cette limitation du compilateur GCC vient du fait que la forme GIMPLE générée à partir du code C++ n'est pas optimisée. Ainsi, nous avons implémenté un outil d'optimisation ≪GGRAPH_OPTIMIZER ≫ qui permet de produire une forme GIMPLE optimisée (ne contenant pas des expressions redondantes). Nous avons ainsi amélioré l'architecture du *front-end* GUML en ajoutant une nouvelle forme intermédiaire (la forme G-Graph) sur laquelle 3 optimisations ont été implémentées pour améliorer l'optimisation de fusion de blocs (*block merging*) implémentée au niveau RTL du compilateur GCC. Ces 3 optimisations sont : *l'élimination des nœuds isomorphes*, *l'élimination des nœuds semi-isomorphes* et *la factorisation des nœuds de test* (Figure 5.15). L'ajout de ce niveau d'optimisation (le niveau G-Graph) produit un code assembleur plus compact que le code assembleur produit par G++ à partir de la même machine à états UML (Tableau 5.3).

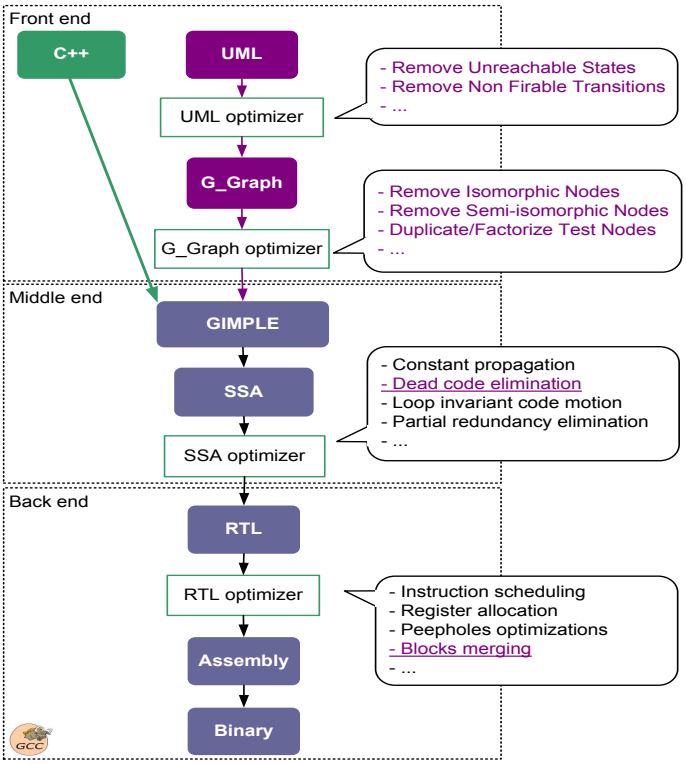


FIGURE 5.15 – L'ajout du deuxième niveau d'optimisation dans le *front-end* GUML, le niveau G-Graph

5.4 Conclusion

Dans ce chapitre, nous avons présenté le deuxième niveau d'optimisation de GUML, le niveau G-Graph. G-Graph est une forme intermédiaire adaptée pour les optimisations des machines à états UML. Cette forme, est très proche du GFC du compilateur GCC sur le quel sont basées toutes ses optimisations de haut niveau. Cependant, GCC construit le GFC à partir du langage C++ perdant ainsi les informations liées au flot de contrôle original des machines à états UML. Le GFC construit par GCC à partir du code C++ comporte des nœuds (des blocs de base) de même expression. GCC bien qu'il optimise le nombre de blocs de base du GFC en effectuant plusieurs passes d'optimisation telles que l'*élimination d'expressions redondantes* (au niveau SSA) et la *fusion de blocs identiques* (au niveau RTL) ne génère pas un assembleur compact. Nous avons montré qu'effectuer des optimisations au niveau G-Graph en se basant sur les informations liées à la machine à états (des nœuds de test de type *State_Cond*, d'autres de type *Event_Cond*, des relations (*S_parent*, *child*) entre un état et les événements déclencheurs de leurs transitions, etc.) résulte à un code assembleur plus compact que le code assembleur généré par GCC (Tableau 5.3). Pour évaluer GUML par rapport à GCC, nous avons généré du code C++ à partir de la machine à état UML de la figure 5.1 en se basant uniquement sur le NSC pattern et en utilisant un générateur de code codé à la main. Dans le chapitre suivant, nous évaluons GUML par rapport à des générateurs de code commerciaux qui utilisent des patterns de génération de code différents et ceci en compilant un exemple concret de modèle UML : le modèle d'une baignoire électronique.

CHAPITRE 6

Evaluation

Ce chapitre présente l'évaluation de la taille du code assembleur produit par le compilateur de modèles GUML par rapport à la taille du code assembleur produit en utilisant l'approche de génération de code. Les trois générateurs de code les plus utilisés du commerce pour le développement des systèmes embarqués Rhapsody, BridgePoint et iUML ont été évalués. Pour cela, la première section de ce chapitre introduit le modèle de la baignoire électronique, qui servira de support pour la compilation. La seconde présente le contexte et l'outillage des expérimentations menées. La troisième section présente et discute les résultats obtenus.

Dans le chapitre précédent, nous avons présenté GUML, le nouveau *front-end* du compilateur GCC pour le langage UML. En se basant sur le *middle* et le *back-end* du GCC, GUML constitue le premier compilateur de modèles qui génère directement de l'assembleur à partir de 3 diagrammes d'UML : le diagramme de classes, le diagramme d'activité et le diagramme de machines à états. GUML représente la mise en œuvre d'une nouvelle approche (*la compilation directe de modèles UML*) dirigée par les modèles pour la conception des systèmes embarqués. Dans ce chapitre, nous évaluons cette nouvelle approche par rapport à l'approche usuelle consistant à générer du code dans un langage de 3ème génération (C/C++/Java, etc.) à partir des modèles UML et à compiler ce code généré (Figure 6.1).

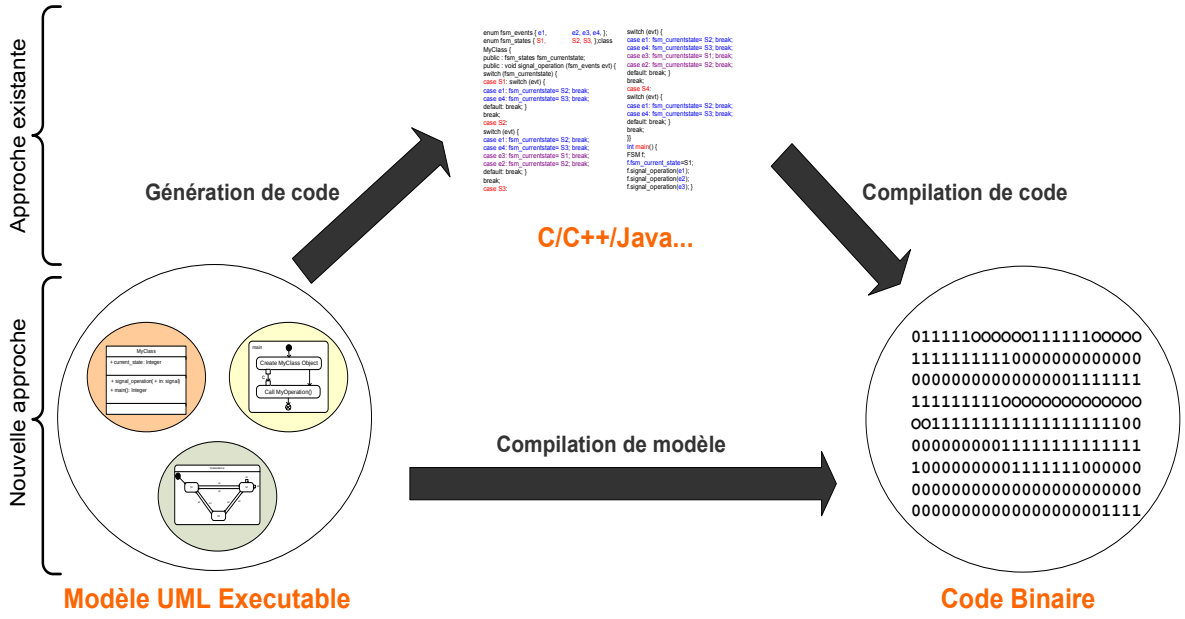


FIGURE 6.1 – L'ancienne et la nouvelle mise en œuvre de l'approche dirigée par les modèles pour le développement des systèmes (La nouvelle mise en œuvre que nous proposons élimine l'étape de la génération de code)

Cette évaluation repose sur la comparaison de la taille des fichiers assembleur produits par (1) GUML en compilant directement un modèle UML (2) GCC en compilant le code généré à partir du même modèle. Il est à noter que dans les deux chapitres précédents, nous avons évalué d'une manière indépendante les deux niveaux d'optimisation ajoutés dans le *front-end* GUML (Tableau 4.2 et Tableau 5.3). Cependant, nous avons utilisé un générateur de code que nous avons implémenté à la main. C'est dans ce chapitre que nous allons manipuler des générateurs de code commerciaux disponibles sur le marché. Nous allons également présenter un modèle réel (une baignoire électronique) sur lequel une évaluation des deux niveaux d'optimisations du *front-end* GUML est possible.

6.1 Modèle de la baignoire électronique

Nous considérons la baignoire électronique de la figure 6.2. Ce système est lié à un robinet et une pompe électroniques. Quatre boutons sont à la disposition de l'utilisateur de la baignoire afin de contrôler le niveau d'eau. Les deux premiers boutons «*fill*» et «*fill_all*» permettent d'activer le robinet électronique qui laisse écouler de l'eau. Avant d'appuyer sur le bouton «*fill*», l'utilisateur devra fixer la quantité d'eau en tournant le bouton «*fix quantity*». La quantité affichée par le bouton tournant «*fix quantity*» n'est pas prise en compte en appuyant sur le bouton «fill_all» qui laisse écouler l'eau jusqu'à atteindre le niveau maximum Lm. Les deux derniers boutons «*remove*» et «*remove_all*» permettent d'activer la pompe électronique qui évacue l'eau présente dans la baignoire en suivant respectivement le même fonctionnement des deux boutons «*fill*» et «*fill_all*» du robinet.

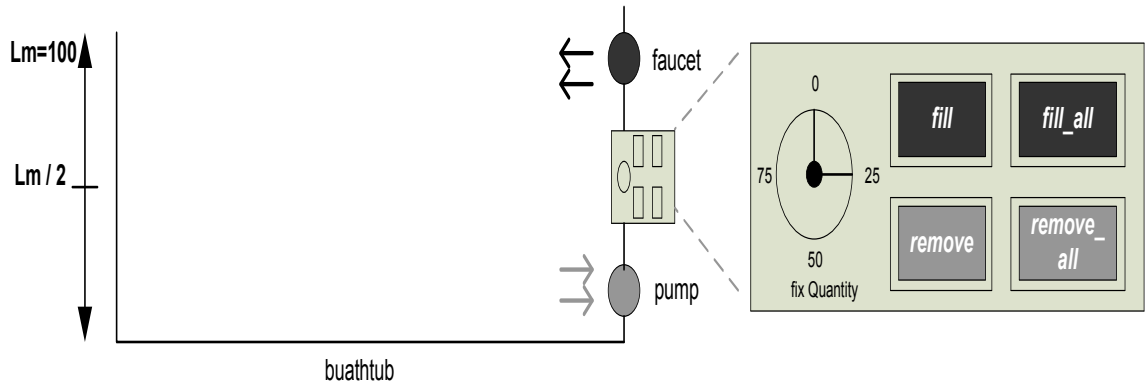


FIGURE 6.2 – Système de la baignoire électronique

Le modèle exécutable de cette baignoire a été réalisé en utilisant le modeleur Papyrus. Ce modèle est représenté par la figure 6.3. Il comporte 4 diagrammes : un diagramme de classes, deux diagrammes d'activité et une machine à états.

Le diagramme de classes contient la classe *Bathtub* possédant deux attributs (*Lm* : le niveau maximum de la baignoire fixée à 100 et *l* : le niveau actuel de l'eau dans la baignoire) et 3 opérations (*main()*, *fill_in(Integer)* et *remove(Integer)*). Le comportement de l'opération *main()* (création d'un objet de type Bathtub et l'envoi des signaux à cet objet) ainsi que celui de l'opération *fill_in(Integer)* sont modélisés chacun à l'aide d'un diagramme d'activité. Nous avons choisi de spécifier le comportement de l'opération *remove(Integer)* en ALF. Nous pouvons spécifier les corps des 3 opérations en ALF ou bien choisir de les spécifier toutes en utilisant des diagrammes d'activité. Nous avons choisi d'utiliser les deux alternatives dans cet exemple, bien que la transformation d'ALF vers GIMPLE requière une analyse syntaxique des expressions ALF que nous n'avons pas encore intégré dans notre compilateur de modèles (GUML prend en entrée l'activité équivalente à la notation ALF présente dans la figure 6.3). Toutefois, pour compiler les expressions ALF, un parseur sous forme de plugin Eclipse est en cours de développement dans notre équipe, il permet de transformer les expressions ALF vers leurs diagrammes d'activité équivalents.

Le cycle de vie d'un objet de type baignoire est modélisé par la machine à états *BathtubSM* contenant un pseudo-état (l'état initial), 4 états simples (*empty*, *filled*, *filled_under_half* et *filled_over_half*) et 17 transitions (figure 6.3).

La classe *Bathtub* déclare la réception des 4 signaux suivant «*fill*», «*fill_all*», «*remove*» et «*remove_all*». Ces 4 signaux ont chacun un paramètre x (de type entier) qui fixe la quantité de l'eau à remplir ou à vider. La réception d'un signal envoyé par l'utilisateur déclenche les transitions qui déclarent ce même signal comme événement déclencheur et change ainsi l'état du système de l'état source de la transition (l'état stable qu'occupait le système avant la réception du signal) vers l'état cible de cette même transition et ceci après avoir vérifié la condition du franchissement de la transition (Figure 6.3). Le franchissement d'une transition entraîne l'exécution de son effet (un comportement lié à la transition). A titre d'exemple, si l'utilisateur choisi d'envoyer le signal «*fill(40)*» et que le système est à l'état *empty*, la transition de l'état *empty* vers l'état *filled_under_half* est franchise. Ceci implique l'exécution de l'opération *fill_in (40)* qui représente l'effet de cette transition et le changement de l'état courant de la baignoire de *vide* à *filled_under_half*. Ce changement d'états n'est possible que si la condition $[l + x <= Lm/2]$ est valide (dans le cas où $x = 40$, cette condition est vérifiée).

Pour exécuter ce modèle UML, nous utilisons le compilateur GUML qui prend en entrée ce modèle exécutable (l'activité modélisant le corps de l'opération *remove(Integer)* est utilisé à la place de la notation ALF) et produit le code assembleur équivalent. Grâce aux deux niveaux d'optimisation ajoutés au *front-end* (le niveau UML et le niveau G-Graph), GUML produit un code assembleur optimisé en termes de taille. Cependant, plusieurs générateurs de code UML assurent la génération d'un code assembleur optimisé à partir des modèles UML. Nous allons ainsi, en nous basant sur l'exemple de la baignoire, comparer la taille des fichiers assembleur produits en utilisant GUML et les autres générateurs de code qui génèrent du C/C++ à partir des modèles UML et utilisent G++ (le *front-end* GCC pour le langage C++) pour produire de l'assembleur optimisé. La section suivante présentera les générateurs de code choisis pour assurer cette évaluation ainsi que les modifications apportées au modèle de la baignoire (figure 6.3) pour pouvoir évaluer les deux niveaux d'optimisation de GUML.

6.2 Contexte et outils de l'évaluation

Les générateurs de code que nous considérons dans cette évaluation sont : Rhapsody 7.5.3 d'IBM, BridgePoint 3.2.4 de Mentor Graphics et iUML 2.4r.5 de Kennedy Carter. Nous les avons choisis pour les raisons suivantes :

1. Ils permettent de générer du code (C/C++/Java etc.) à partir des modèles UML et plus précisément les machines à états.
2. Ce sont les générateurs de code les plus utilisés pour implémenter les systèmes embarqués temps réel, contraints par leurs ressources exigeants ainsi un binaire le plus compact possible. Le code assembleur produit par ces 3 générateurs de code est sensé être optimisé.
3. Dans leurs implémentations des machines à états, ils couvrent les 3 patterns de génération de code les plus utilisés (section 2.1.2.2) à savoir, le SP (*State Pattern*) et le NSC (*Nested Switch Cases*) utilisés par Rhapsody et le STT (*State Transition Table*) utilisé par iUML et BridgePoint.

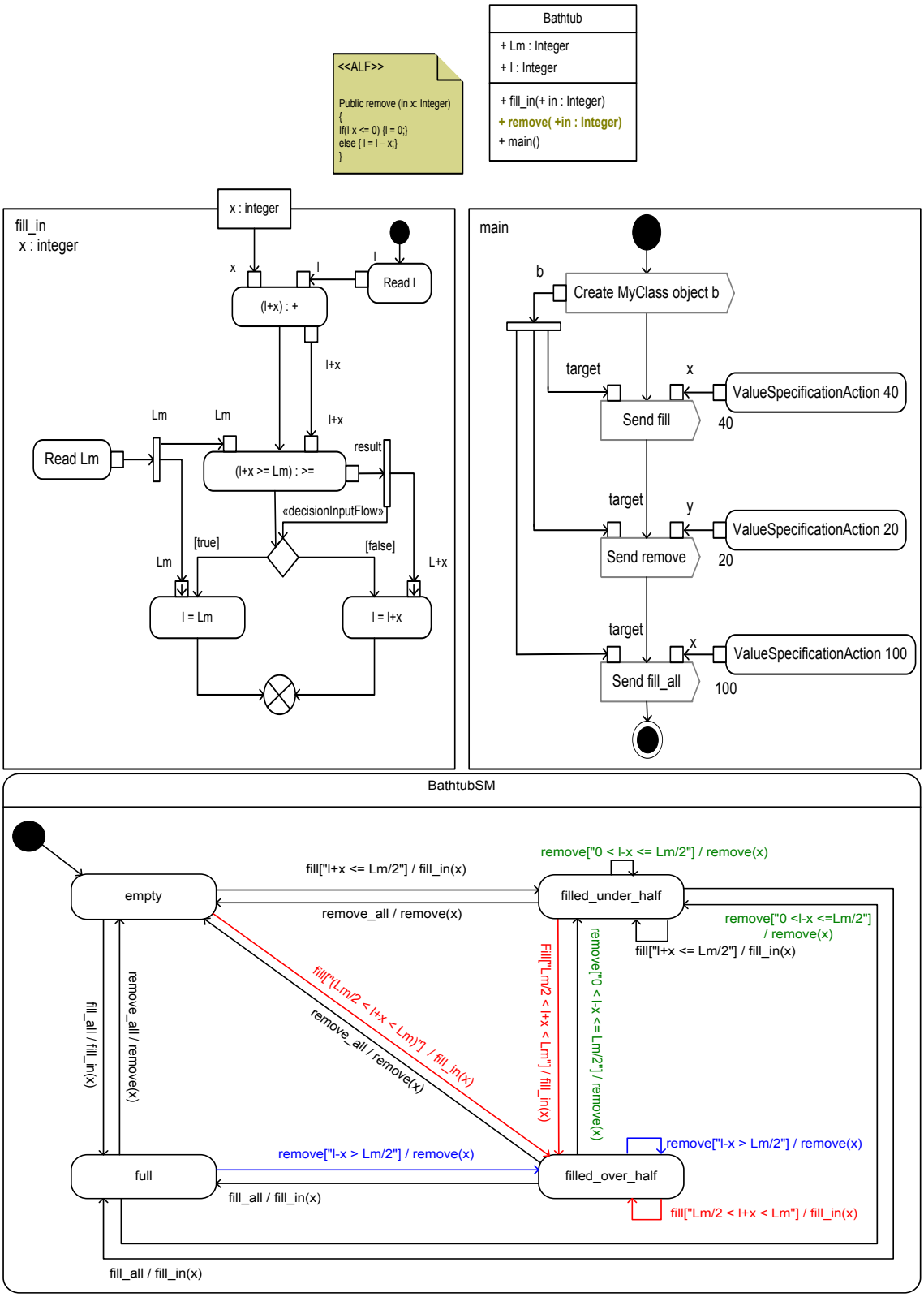


FIGURE 6.3 – Le modèle exécutable du système baignoire : 4 diagrammes UML : un diagramme de classe (Bathub) deux diagrammes d'activité (main et fill_in) et une machine à états (BathtubSM)

Le but de cette évaluation est de comparer la taille du code assembleur généré par GUML à partir du modèle de la baignoire avec les tailles des fichiers assembleur produits par ces 3 générateurs de code à partir du même modèle. Nous avons enrichi le pourvoir d'optimisation de GUML en améliorant deux optimisations de haut niveau du compilateur GCC : *l'élimination du code mort (dce)* et *l'élimination des expressions redondantes (block merging)*. Cette étude expérimentale est l'occasion d'évaluer le pouvoir d'optimisation de GUML d'autant plus que les deux outils BridgePoint et iUML sont connus par le fait qu'ils intègrent des modules d'optimisation de code assurant la production d'un assembleur compact à partir des modèles UML (section 2.1.2.2).

En réalité Rhapsody, BridgePoint et iUML sont beaucoup plus que des générateurs de code. Ce sont des AGLs qui intègrent des modeleurs (pour concevoir les modèles UML), des générateurs de code (pour générer du code à partir des modèles) et même des compilateurs (pour produire l'exécutable à partir du code généré). Par contre, GUML étant un compilateur de modèles UML n'intègre qu'une seule fonctionnalité : la production de l'exécutable à partir du modèle UML. GUML prend en entrée un fichier.*uml* généré par n'importe quel modeleur UML. Nous avons choisi d'utiliser le modeleur Papyrus pour modéliser l'application de la baignoire. Pour les 3 AGL, nous avons utilisé leurs modeleurs intégrés (Figure 6.4).

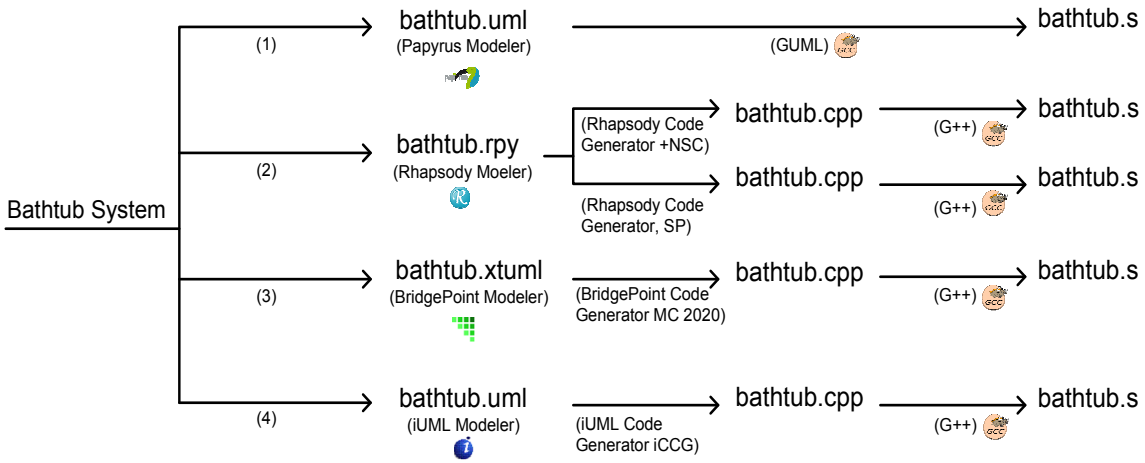


FIGURE 6.4 – Les 4 expériences réalisées pour évaluer la taille du fichier assembleur (bathtub.s) produit par GUML par rapport aux fichiers assembleur produits par les 3 AGLs : Rhaspdody, BridgePoint et iUML

Les compilateurs intégrés dans les 3 AGLs que nous avons téléchargés sont des versions anciennes de GCC (*Cygwin 3.4* pour iUML et *Mingw 5.1.3* pour BridgePoint 3.2.4 et Rhapsody 7.5.3). Puisque GUML utilise le *middle* et le *back-end* de la version 4.5.2 du compilateur GCC, nous avons compilé le code généré par les 3 AGLs en utilisant cette même version. Nous avons également spécifié (pour les 4 étapes de compilation présentées dans la figure 6.4) l'option -Os qui active les optimisations de GCC assurant la production d'un assembleur compact.

Les 3 AGLs que nous avons choisis pour évaluer GUML sont des outils matures. Leurs environnements de génération de code sont eux aussi matures et font appel à plusieurs bibliothèques statiques et dynamiques. Ainsi, la comparaison de la taille de tout le code exécutable généré à partir de tout le modèle de la baignoire (les 3 diagrammes : de

classes, d'activité et d'états transitions) n'est pas trop significative vu que les tailles des fichiers assembleurs générés par ces 3 AGLs sont beaucoup plus grandes que la taille du fichier assembleur généré par GUML, qui représente la première version du compilateur de modèles proposé dans ce travail de recherche. De plus, les méthodes de la génération de code à partir d'un diagramme de classes ou d'un diagramme d'activité sont pratiquement les mêmes pour les 3 AGLs choisis. En effet, une classe en UML est toujours implémentée en utilisant la notion de classe d'un langage orienté objet, les opérations UML se transforment en méthodes de la classe et les propriétés en attributs etc. Pour un diagramme d'activité, bien que l'implémentation varie d'un outil à un autre, le code assembleur généré est le même. A titre d'exemple, le corps de l'opération *fill_in(x)* modélisé par un diagramme d'activité (Figure 6.3) est spécifié en Rhapsody, iUML et BridgePoint en utilisant leurs langages d'action respectifs AL (Action Language), ASL (Action Specific Language) et OAL (Object Action Language). Bien que la syntaxe de ces 3 langages soit différente (figure 6.5), le code assembleur généré à partir des 3 spécifications de la figure 6.5 est le même.

Rhapsody AL

```
if (l + params->x >= Lm)
      l=Lm;
else
      l= l+ param->x;
```

BridgePoint OAL

```
if (self.l + param.x >= self.Lm)
      assign self.l=self.Lm;
else
      assign self.l = self.l+ param.x;
end if;
```

iUML ASL

```
b= find-only Bathtub
if (b.l + x >= b.Lm) then
      b.l=b.Lm
else
      b.l= b.l+x
endif
```

FIGURE 6.5 – La spécification du corps de l'opération *fill_in(x)* en utilisant les 3 langages d'action (AL, OAL et ASL) à syntaxes concrètes différentes

Par conséquent, nous avons choisi de comparer uniquement le code assembleur qui correspond à l'implémentation de la machine à états UML de la figure 6.3. En effet, et comme le montre le tableau 6.1, les 3 AGLs implémentent les machines à états de façons différentes (n'utilisent pas les mêmes patrons de génération de code ni les mêmes structures de données). Il est donc plus intéressant de comparer le code assembleur généré à partir des machines à états UML que celui généré à partir d'un diagramme de classes ou d'un diagramme d'activité. Cette restriction nous permet d'évaluer les deux niveaux d'optimisation ajoutés à GUML (le niveau UML et le niveau G-Graph). En effet, les optimisations implémentées dans le *front-end* GUML influent uniquement sur la taille du code assembleur généré en compilant les machines à états. Le premier niveau d'optimisation améliore l'optimisation de l'*élimination du code mort (dce)* en supprimant les états inatteignables présents dans la machine à états à compiler. La machine à états de la baignoire (figure 6.2) ne contient pas d'états inatteignables. Pour évaluer le premier niveau d'optimisation de GUML, nous avons ajouté quelques contraintes sur le système de la baignoire électronique qui modifient la condition de franchissement de certaines transitions résultant ainsi à l'apparition d'un état inatteignable.

		Rhapsody		**BridgePoint**	**iUML**
implémentation	Pattern de Génération de code	NSC	SP	STT	STT
	Etats	Enumération	Classes	Enumération	Enumération
	Evénements	Enumération	Classes	Enumération	Classes
	Langage d'Action	AL		OAL	ASL
Concepts pris en comptes	Type des états : (composites, orthogonaux, historiques)	+		-	-
	Actions des états (entry, doActivity, exit)	+		+	+
	Transition avec gardes	+		-	-
	Transition avec Effet	+		+	+
	Evénement avec paramètres	+		+	+
	Gestion des événements (consommés, perdus, ignorés)	+		+	+

Tableau 6.1 – Etude de l'implémentation des machines à états UML par Rhapsody, BridgePoint et iUML

La section suivante présente le nouveau modèle de la baignoire électronique utilisé pour évaluer la taille de l'assembleur produit par GUML et celui produit par Rhapsody. Les deux autres AGLs (BridgePoint et iUML) sont évalués en utilisant un modèle plus simplifié de la baignoire électronique vu qu'ils ne supportent pas les *gardes* (tableau 6.1). Les résultats de toutes les expérimentations seront décrits et discutés dans la section suivante.

6.3 GUML vs 3 AGLs : Rhapsody, BridgePoint et iUML

Nous supposons que la baignoire électronique (figure 6.2) est utilisée pour effectuer des expériences chimiques mélangeant plusieurs liquides. Nous supposons également que la quantité d'eau à remplir pour un liquide particulier ne peut excéder la moitié du volume maximal autorisé de la baignoire (pour obtenir un mélange concentré). Cette première hypothèse est donc liée au paramètre x envoyé par le signal «*fill*» qui doit respecter la contrainte suivante :

$$\text{C1 : contrainte sur la valeur } x \text{ du signal «}\textit{fill}\text{» : } [l + x <= Lm/2]$$

Cette contrainte est à respecter uniquement en envoyant le signal «*fill*». En effet, l'utilisateur peut diluer complètement la solution obtenue en envoyant le signal «*fill_all*». Le signal «*fill_all*» comme nous l'avons déjà expliqué ne prend pas en compte la valeur fixée par le bouton tournant (la quantité x à remplir). Il remplit la baignoire jusqu'au bout $l = Lm$.

Une autre contrainte est à respecter par ce système mais cette fois-ci concernant le paramètre x du signal «*remove*». Cette contrainte est en rapport avec le débit de l'évacuation du liquide. Une fois remplie ($l = Lm$), l'utilisateur doit vider la moitié supérieure. Il pourra par la suite vider peu à peu jusqu'à atteindre l'état vide de la baignoire.

C2 : contrainte sur la valeur x du signal «*remove*» : $[l - x <= Lm/2]$

En d'autres termes, la première contrainte ne pose pas de condition sur le débit du remplissage avant la moitié mais exige que le remplissage après la moitié soit total et non interrompu (en une seule fois). Par analogie à la première contrainte, la deuxième contrainte ne pose pas de condition sur la vidange une fois que le niveau du liquide est inférieur à la moitié, cependant, une fois remplie, la vidange jusqu'à la moitié se fait en une seule fois.

La prise en compte de ces deux contraintes (C1 et C2) dans la machine à états UML de la figure 6.3 se traduit par la modification des conditions du franchissement des transitons sortantes de l'état *full* et déclenchées par la réception du signal «*remove*» et celles sortantes des états *empty* et *filled_under_half* et déclenchées par la réception du signal «*fill*». La machine à états UML qui modélise le comportement de la baignoire électronique en respectant les deux contraintes C1 et C2 est présentée par la figure 6.6. Nous pouvons remarquer que l'état *filled_over_half* devient inatteignable vu que ces transitions entrantes ont toutes des *gardes* fausses.

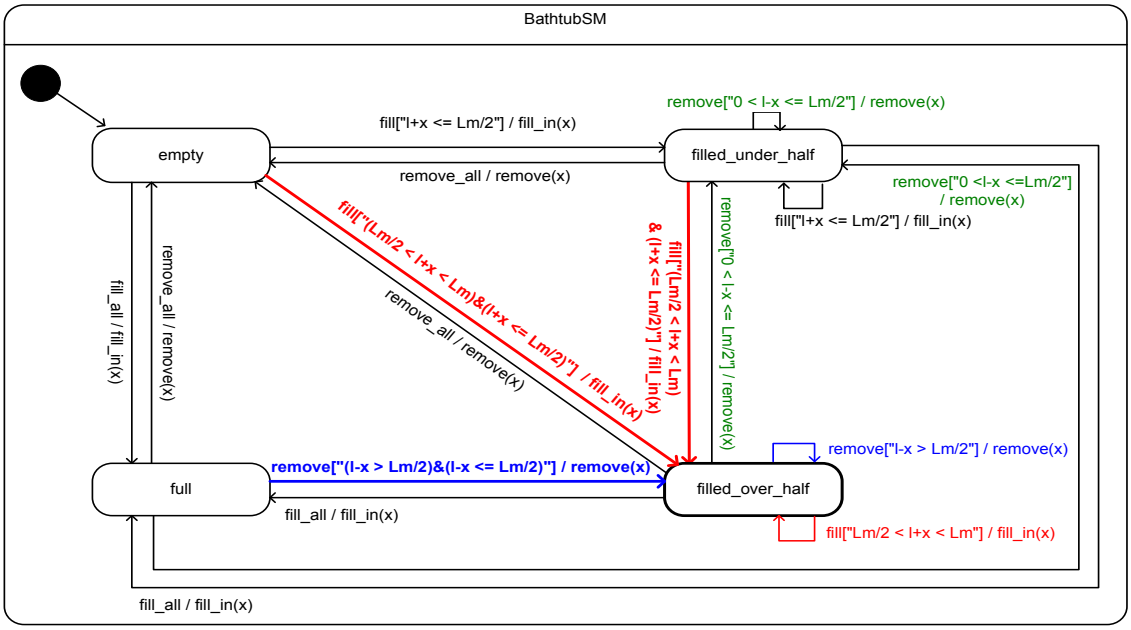


FIGURE 6.6 – La machine à états UML de la baignoire en respectant les contraintes C1 et C2 : les 3 transitions entrantes de l'état *filled_over_half* provenant des 3 autres états (*empty*, *full* et*filled_under_half*) ont des gardes toujours fausses

Ce type d'états inatteignables peut être détecté par GUML grâce à la passerelle de l'outil *Diversity* qui, en analysant l'automate de la figure 6.6 détecte l'état inatteignable *filled_over_half*. Bien que le compilateur GCC est capable de détecter les conditions toujours fausses et d'éliminer le code mort résultant, il reste toujours incapable de d'éliminer le code mort lié à l'état inatteignable *filled_over_half*. La section suivante présente le résultat de la compilation de la machine à états UML de la figure 6.6 en utilisant GUML et Rhapsody.

6.3.1 GUML vs Rhapsody

La machine à états de la figure 6.6 a été modélisée en Papyrus. Elle représente la vue graphique (fichier.di) générée par Papyrus. Mise à part la vue graphique, Papyrus génère aussi le fichier .uml (*baignoire.uml*) qui contient toutes les informations relatives au modèle de la baignoire. C'est ce fichier que le compilateur GUML prend en entrée pour générer l'assembleur équivalent en lançant tout simplement la commande : `guml baignoire.uml -Os -S`. Avant l'ajout des deux niveaux d'optimisation de GUML, le compilateur *uml1* appelé par le pilote *guml* fait appel directement à Acceleo pour générer la forme GIMPLE équivalente au fichier *baignoire.uml*. En réutilisant le *middle* et le *back-end* du compilateur GCC, GUML produit le fichier assembleur à partir de la forme GIMPLE (Figure 3.7). Après l'ajout des deux niveaux d'optimisation de GUML (le niveau UML et le niveau G-Graph), le compilateur *uml1* lance un script dans lequel les deux plugins d'optimisations sont appelés (Figure 5.15)).

Le résultat du lancement du premier plugin d'optimisation sur le fichier *baignoire.uml* est l'élimination de l'état inatteignable *filled_over_half*. La représentation graphique de la machine à états optimisée est présentée par la figure 6.7.

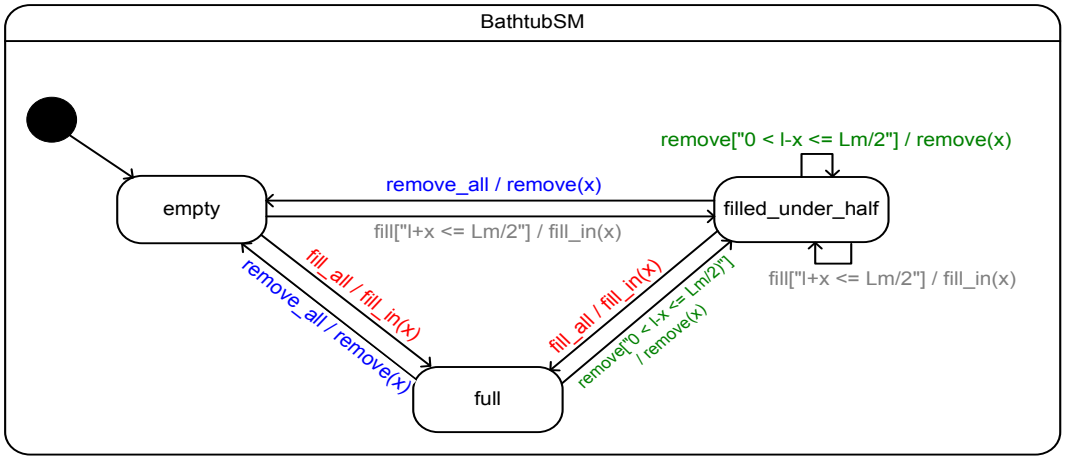


FIGURE 6.7 – La machine à états UML de la baignoire après avoir éliminé l'état inatteignable filled_over_half

La forme G-Graph (figure 6.8) équivalente à la machine à états UML de la figure 6.7 est produite en lançant le plugin « SM2GGRRAPH ». Les nœuds de type Assign et *Call* partagent la même forme (forme rectangulaire). En effet, dans la forme S-graph (à partir de la quelle la forme G-Graph a été défini), les nœuds de type Assign peuvent représenter des expressions d'appels d'opérations. Nous avons choisi d'ajouter un autre type de nœud (GIMPLE_*Call*_Node) afin de faciliter la transformation des nœuds G-graph vers les instructions Gimple équivalentes (un nœud de type Assign se transforme vers l'instruction *gimple_assign* et un nœud de type *Call* se transforme vers l'instruction *gimple_call*). La forme G-Graph de la figure 6.8 est optimisée en appelant le plugin « GGRAPH_OPTIMIZER ». En effet, elle contient 6 couples de nœuds isomorphes qui sont les couples `(L8, L13)`, `(L9, L11)`, `(L14, L18)`, `(L15, L19)` `(L16, L20)` et `(L17, L21)`. En exécutant l'optimisation de *l'élimination des nœuds isomorphes*, les nœuds `L13`, `L11`, `L18`, `L19`, `L20` et `L21` seront éliminés.

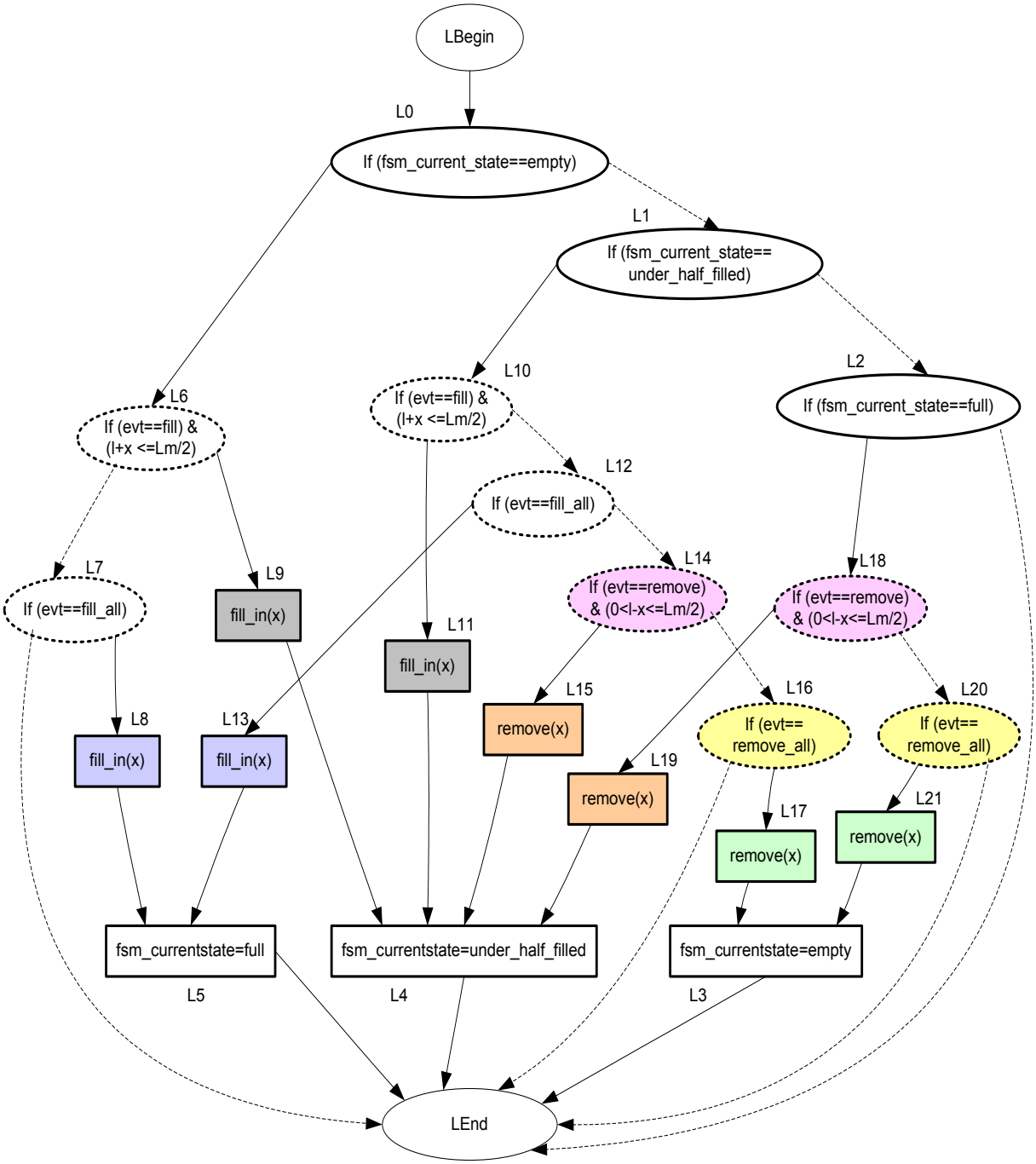


FIGURE 6.8 – La forme G-Graph produite à partir de la machine à états UML de la figure 6.7. Les nœuds colorés forment les couples des nœuds isomorphes présents dans cette forme

L'exécution de l'optimisation de *la factorisation des nœuds de test* a produit la forme G-Graph de la figure 6.9 (pour plus de détails, voir le chapitre précédant, section 5.3.3). C'est à partir de cette représentation intermédiaire que la forme GIMPLE correspondante à la machine à états UML de la figure 6.6 sera générée.

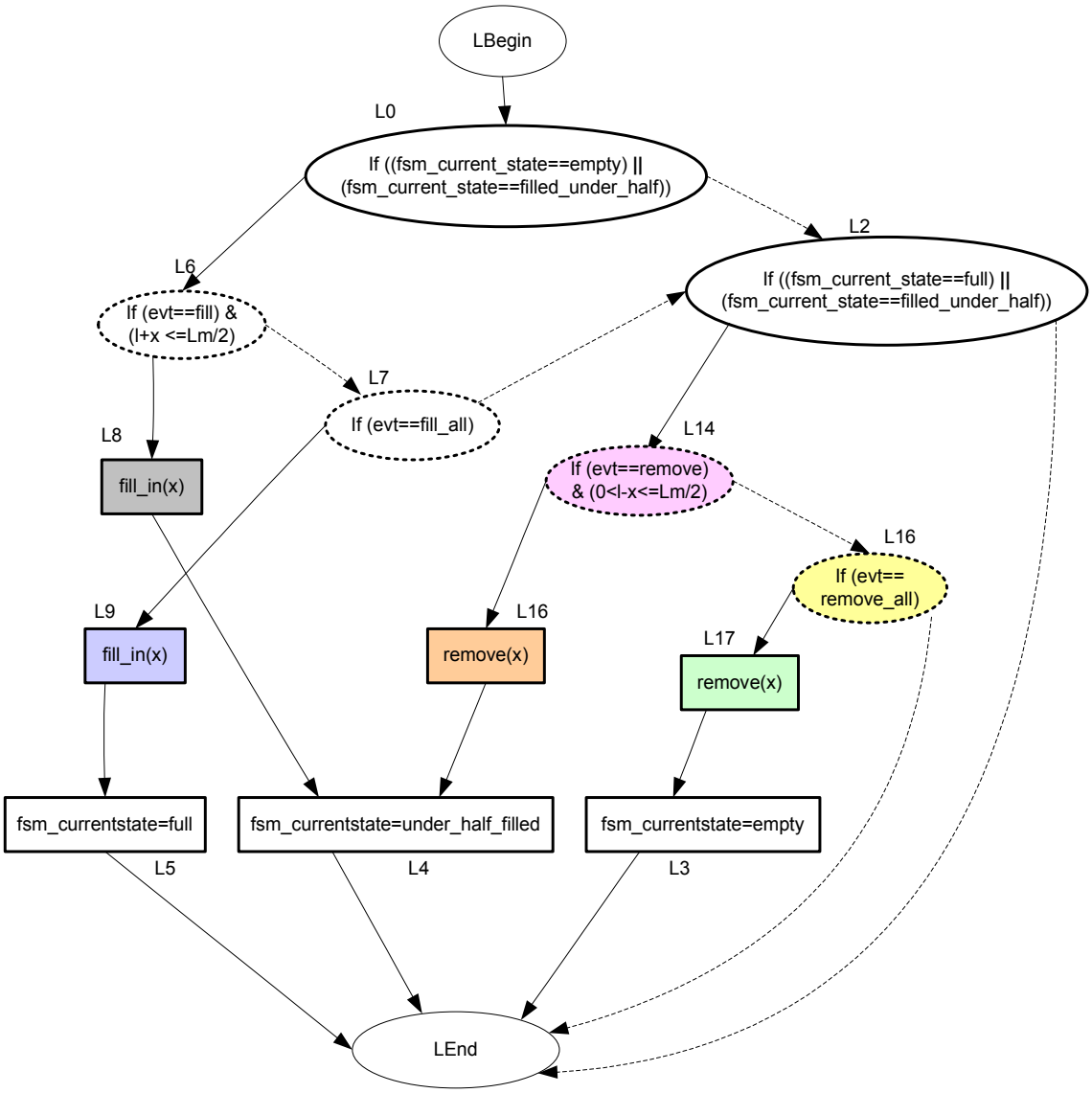


FIGURE 6.9 – La forme G-Graph produite suite à l'exécution de l'optimisation "factorisation des nœuds de test"

Le tableau 6.2 illustre l'impact des deux niveaux d'optimisation sur la taille du fichier assembleur produit par GUML à partir de la machine à états de la figure 6.6.

Compilateur	Taille (octets) code assembleur	Taux d'optimisation
G++	3254	--
GUML	2500	23.17 %
GUML + 1er niveau d'optimisation	2238	31.22 %
GUML + 1er et 2ème niveaux d'optimisation	2095	35.62 %

Tableau 6.2 – Impact des deux niveaux d'optimisation de GUML sur la taille du fichier assembleur généré

Pour cet exemple, le premier niveau d'optimisation de GUML (le niveau UML) a diminué la taille du code assembleur d'environ 20%. Cette taille a été encore réduite d'environ 10% en exécutant les optimisations définies dans le deuxième niveau d'optimisation de GUML (le niveau G-Graph). Par rapport à la taille du fichier assembleur généré par G++ en implémentant à la main (en suivant le NSC pattern) la machine à états UML de la figure 6.6, la taille du fichier assembleur produit par GUML est 3 fois plus compacte (dernière ligne du tableau 6.2). Cependant, nous ne devons pas nous contenter de ce résultat encourageant puisque plusieurs outils disponibles sur le marché assurent la production de code assembleur compact à partir des machines à états UML. Les 3 AGLs que nous avons choisis pour évaluer GUML sont utilisés pour le développement des systèmes temps réel embarqués qui intègrent souvent des mémoires de taille réduites. Ainsi, produire un code assembleur compact est l'une des contraintes à respecter par ces outils.

Une évaluation de GUML par rapport à des générateurs de code commerciaux (plutôt qu'un code codé à la main) s'avère plus intéressante. Nous commençons par évaluer GUML par rapport à Rhapsody. Nous avons ainsi modélisé la même machine à états UML de la figure 6.6 en utilisant Rhapsody. Pour générer du code à partir des machines à états UML, Rhapsody se base sur deux patterns : le NSC (le pattern utilisé par défaut) et le SP (*State Pattern*). Le NSC pattern est utilisé pour produire un assembleur compact sans se soucier du temps qu'il met pour être compilé et exécuté. Le SP pattern produit un code assembleur plus rapide à exécuter et si le modèle à compiler contient plusieurs machines à états héritées et hiérarchiques, ce pattern produit un code assembleur plus compact que le NSC pattern. Dans les deux cas, Rhapsody génère 3 fichiers C++ à partir du modèle de la baignoire : `MainDefaultComponent.cpp` (implémentant la fonction *main()* qui crée l'objet baignoire et envoi les signaux à cet objet), `BathtubModel.cpp` (l'implémentation de la classe *Bathtub*) et `BathtubSM.cpp` (l'implémentation de la machine à états UML *BathtubSM*). C'est la taille du fichier assembleur généré à partir du fichier `BathtubSM.cpp` que nous comparons avec celle du fichier assembleur produit par GUML. La figure 6.10 présente les différentes tailles des fichiers assembleur produits à partir de la machine à états UML de la figure 6.6 par GUML et Rhapsody (en utilisant les deux patterns NSC et SP).

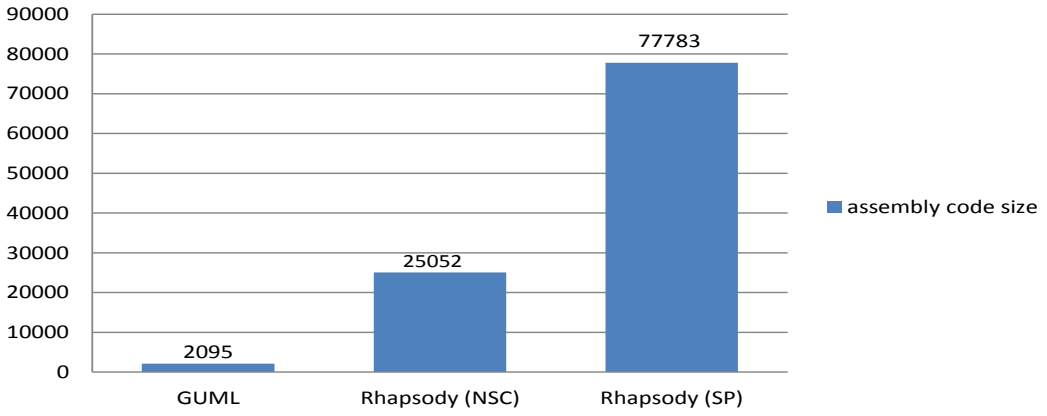


FIGURE 6.10 – La taille des fichiers assembleur générés par GUML (en compilant la machine à états UML de la figure 6.6) et Rhapsody (en générant du code à partir de la même machine à états en utilisant les deux patterns NSC et SP et en le compilant avec GCC)

Il y apparait clairement que l'assembleur produit par GUML est beaucoup plus compact que celui produit par Rhapsody. Cela peut être expliqué par le fait que le *Framework* de génération de code de Rhapsody fait appel à plusieurs bibliothèques statiques dont le code est inclus dans le fichier assembleur final. A titre d'exemple, la bibliothèque OXF (*Object eXecution Framework*) définit plusieurs services utilisés dans le code C++ généré tels que la classe `IOxfActive` pour la création d'objet actif, le type `TakeEventStatus` pour la gestion des différents états d'un événement (consommé, en attente, rejeté, ...), etc.

Une étude plus détaillée du code C++ généré par Rhapsody montre que même en ignorant le code lié aux bibliothèques, Rhapsody ne produit pas un code optimisé. A titre d'exemple, pour changer la valeur de l'état courant de $S0$ vers $S1$ (même en compilant une machine à états simple, ne contenant pas d'états composites), Rhapsody génère deux affectations : `rootState_subState = S1` et `rootState_active = S1`. La déclaration de deux variables pour stocker la valeur de l'état courant et la double affectation augmentent considérablement la taille de l'assembleur.

Avec l'ajout de ses deux niveaux d'optimisation (le niveau UML pour améliorer l'optimisation du *dce* et le niveau G-Graph pour améliorer celle de la fusion des blocs de bases *blocks merging*), GUML augmente le pouvoir d'optimisation de GCC afin de produire un code assembleur plus compact. Rhapsody, bien que souvent considéré comme l'environnement de développement dirigé par les modèles le plus utilisé pour développer les systèmes temps réel embarqués où la production d'un code optimisé est exigée, ne semble pas proposer de solutions à ce problème. En effet, une étude détaillée du code assembleur produit par Rhapsody montre que ce dernier se base uniquement sur les optimisations du compilateur G++ (les optimisations du niveau SSA (*middle-end* du G++) et du niveau RTL (*back-end* du G++)). Ainsi, le code relatif à l'état inatteignable *filled_over_half* et les expressions redondantes de test de valeurs des signaux reçus tel que (`if (evt==fill)`) existent dans le code C++ produit par Rhapsody et persistent dans le code assembleur généré, ce qui augmente encore plus l'écart entre les tailles des fichiers assembleurs produits par GUML et Rhapsody en utilisant le NSC pattern.

L'écart est encore plus visible si l'utilisateur choisi d'implémenter sa machine à états UML en utilisant le State Pattern. La figure 6.11 montre que le pattern NSC produit un code assembleur 3 fois plus compact que celui produit par le State Pattern. Ceci peut être expliqué par le fait que le SP implémente chaque état de la machine à états UML en utilisant une classe alors que le NSC utilise une seule énumération pour implémenter tous les états simples. Cependant, même si le NSC produit un code assembleur plus compact que le SP, la stratégie globale de génération de code adoptée par l'outil Rhapsody semble négliger l'aspect de l'optimisation et s'intéresser beaucoup plus à l'aspect de lisibilité du code généré pour assurer la rétro-ingénierie.

Les deux outils que nous évaluons dans la section suivante (BridgePoint et iUML) sont comme Rhapsody, dédiés au développement des systèmes temps réel embarqués, mais s'intéressent beaucoup plus que Rhapsody à la production d'un code assembleur compact.

6.3.2 GUML vs Rhapsody , iUML et BridgePoint

Comme le montre le tableau 6.1, BridgePoint et iUML ne supportent pas les gardes. Par la suite, nous ne pouvons pas modéliser la machine à états UML de la figure 6.6 en

utilisant ces deux outils. Néanmoins, puisque ces deux outils se basent sur un pattern de génération de code (le pattern STT) différent des patterns utilisés par Rhapsody (NSC et SP) et qu'ils effectuent des optimisations lors de la génération de code, il est intéressant d'évaluer la taille de leurs fichiers assembleurs produits à partir d'une machine à états UML sans gardes. Ainsi, nous considérons la machine à états UML de la figure 6.7 produite suite à l'exécution de l'optimisation de l élimination des états inatteignables. Les gardes des transitions de cette machine à états (contrairement aux gardes exprimées dans la machine à états de la figure 6.6) ne sont pas utiles pour les optimisations de GUML. Ainsi, nous avons modélisé la machine à états UML de la figure 6.7 en ignorant les gardes avec les 3 AGLs. La figure 6.11 présente les tailles des 5 fichiers assembleur générés respectivement par GUML, BridgePoint, iUML et Rhapsody en utilisant les deux patterns (SP et NSC). Nous considérons uniquement le code assembleur provenant de l'implémentation de la machine à états (les outils iUML et BridgePoint génère l'implémentation de la machine à états dans un fichier C++ indépendant).

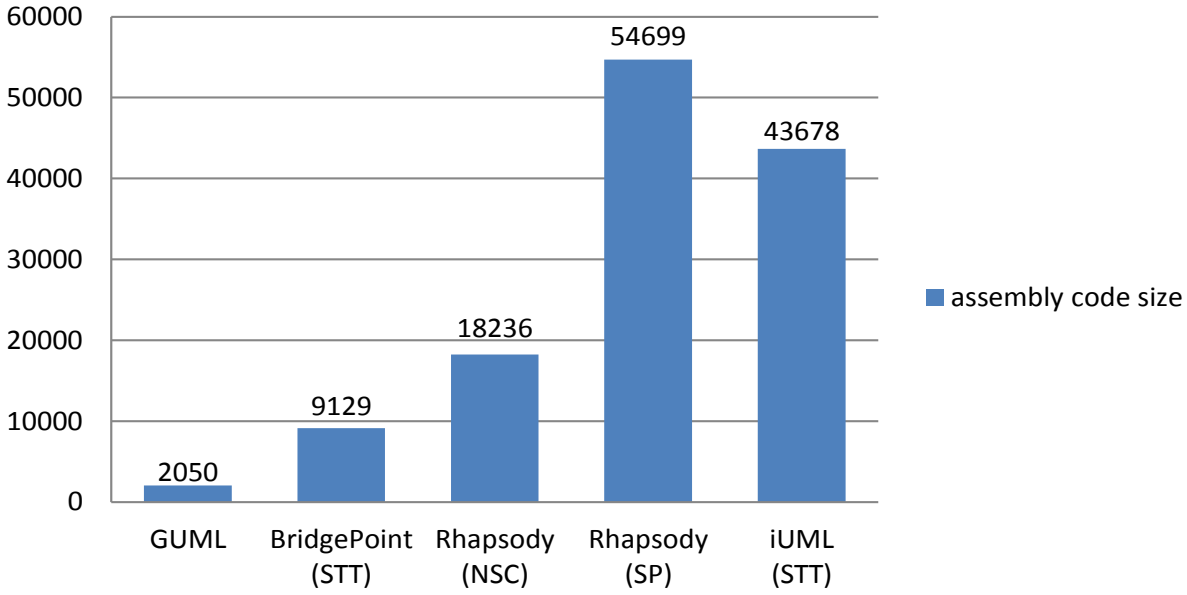


FIGURE 6.11 – Les tailles des codes assembleurs générés à partir de la machine à états UML de la figure 6.7 en utilisant GUML, BridgePoint, Rhapsody et iUML

L'exécution des optimisations définies dans le niveau UML de GUML n'ont pas d'effet sur la machine à états UML de la figure 6.7 (pas d'états inatteignables à éliminer). Cependant, les optimisations définies dans la forme intermédiaire G-Graph ont éliminé les expressions redondantes (figure 6.9) et par la suite diminué la taille du fichier assembleur produit.

En analysant les fichiers assembleur produits par les 3 autres AGLs, nous découvrons qu'aucun de ses 3 outils n'est capable d'éliminer les expressions redondantes que GUML élimine grâce à ses optimisations de la forme G-Graph. Ces 3 AGLs comptent sur les optimisations de G++ sans les améliorer. Cependant, l'utilisation du pattern STT par BridgePoint a diminué considérablement la taille du fichier assembleur par rapport à celle produite par Rhapsody. En effet, la figure 6.11 montre que pour l'exemple de la machine à états de la figure 6.7 , le pattern STT utilisé par BridgePoint a produit un fichier assembleur

deux fois plus compact que le fichier produit par le pattern NSC de Rhapsody et 6 fois plus compact que celui produit par le pattern SP. BridgePoint, outre l'utilisation du pattern le moins gourmand en espace mémoire, optimise le code C++ généré avant de le passer au compilateur G++. Cependant, les optimisations que BridgePoint effectue (section 2.2.2.1) telles que l'élimination du code mort présent dans le code C++ qui provient de la déclaration d'un attribut non utilisé sont déjà prises en compte dans le *middle-end* du G++.

Bien que l'implémentation des machines à états d'iUML se base tout comme BridgePoint sur le pattern STT, la taille du code assembleur produit par iUML est beaucoup plus grande que celle du fichier produit par BridgePoint. Ceci peut être expliqué par le fait que l'outil iUML (tout comme Rhapsody) implémente chaque événement à l'aide d'une classe.

Ainsi, le pattern de génération de code choisi (STT, NSC, SP,...), les structures de données utilisés (une classe, une énumération, ...) et le type de bibliothèque choisi (statique ou dynamique) ont beaucoup d'influence sur la taille du code assembleur produit. Cependant, et même en négligeant tous ces facteurs qui sont derrière la différence énorme entre les ordres de grandeur de la taille du code produit par GUML (milliers d'octets) et les autres générateurs de code (dizaines de milliers d'octets), GUML produit un code assembleur plus compact grâce à ses optimisations de haut niveau qu'aucun compilateur ou générateur de code supporte.

6.4 Conclusion

Dans ce chapitre, une évaluation de GUML, le compilateur de modèles optimisant proposé dans le chapitre 3, a été présentée. Ce compilateur définit 2 niveaux d'optimisations pour améliorer la qualité du code assembleur produit à partir des modèles UML et en particulier les machines à états. L'évaluation s'est donc portée sur la comparaison de la taille du code assembleur produit à partir des machines à états en utilisant GUML et 3 générateurs de code disponibles sur le marché et utilisés pour le développement des systèmes temps réel embarqués exigeant la production d'un code assembleur optimisé. Les expérimentations que nous avons menées révèlent que le code produit par GUML est plus compact que celui produit par les générateurs de code choisis et que les optimisations que GUML effectue ne sont implémentées dans aucun de ces générateurs. Ainsi, la nouvelle approche que nous proposons dans ce travail de recherche et qui consiste à compiler directement les modèles UML au lieu de générer du code et puis compiler le code généré est une approche prometteuse qui peut améliorer considérablement la taille du code assembleur et par la suite l'exécutable de l'application. Nous n'avons traité dans ce travail de recherche qu'un seul critère d'optimisation : «*la réduction de la taille du code*». En revanche, plusieurs autres critères d'optimisation sont à étudier tels que la réduction du temps d'exécution, du temps de compilation et de la consommation d'énergie. L'étude de ces critères d'optimisation de code fait partie des perspectives de ce travail de recherche. Les autres perspectives et la conclusion générale constituent l'objet du chapitre suivant.

CHAPITRE 7

Conclusion

7.1 Bilan

L'objectif de cette thèse était de traiter la problématique de la génération de code *exécutable* optimisé à partir des modèles UML de systèmes embarqués. Nous avons commencé nos travaux par l'étude de l'approche classique dirigée par les modèles pour le développement des systèmes embarqués. Nous avons remarqué que cette approche se base sur la génération de code en langage de programmation de 3ème génération à partir des modèles exécutables UML. Cette approche se base essentiellement sur les optimisations des compilateurs de code qui prennent en entrée le code généré et produisent en sortie l'exécutable correspondant. Nous avons choisi d'étudier le compilateur de code GCC très utilisé dans les travaux de recherche sur la compilation. Les critères d'optimisation de code sont variés : l'optimisation de temps d'exécution, de temps de compilation, de la taille du code, etc. Pour les systèmes embarqués souvent de taille mémoire réduite, le critère d'optimisation le plus important et celui de la réduction de la taille du code. Ainsi, nous nous sommes intéressés dans cette thèse à ce critère d'optimisation.

Notre étude [63] des optimisations du compilateur GCC assurant la production d'un code compact à partir des modèles UML a révélé des lacunes concernant la capacité de GCC à effectuer les optimisations liées à la sémantique propre aux modèles UML. En effet, au cours de la génération de code, les informations liées à la sémantique du langage de modélisation UML sont perdues. Ces informations sont donc inaccessibles au compilateur GCC, qui se trouve dans l'incapacité de les exploiter à des fins d'optimisation. Pour remédier à ce problème, nous avons proposé une nouvelle mise en œuvre de l'approche dirigée par les modèles qui supprime l'étape de la génération de code, et remplace cette étape par une compilation directe des modèles UML exécutables [77]. Nous avons ainsi proposé GUML le premier compilateur de modèles UML qui compile les classes, les activités et les machine à états. Etant un *front-end* GCC pour le langage UML, GUML réutilise toutes les optimisations de haut niveau (niveau SSA) et de bas niveau (niveau RTL) du compilateur GCC. Il produit ainsi au pire des cas un code binaire de mêmes performances que le code binaire produit par le compilateur GCC.

Par la suite, nous nous sommes intéressés à l'amélioration du pouvoir d'optimisation de notre compilateur de modèles. Pour ce faire, nous avons ajouté un premier niveau d'optimisation (le niveau UML) dans lequel nous avons implémenté des optimisations améliorant l'élimination du code mort (*dead code elimination*), une optimisation de la forme SSA de GCC. Nous avons également ajouté à GUML un deuxième niveau d'optimisation (le niveau G-Graph) dans lequel nous avons implémenté des optimisations améliorant la fusion des blocs (*blocks merging*), une optimisation de la forme RTL du compilateur GCC. Grâce à ses deux niveaux d'optimisation, GUML est capable de produire, dans certains cas, un code assembleur plus compact que le code assembleur produit par G++ (le *front-end* GCC pour le langage C++) à partir de la même machine à états UML. Les expérimentations que nous avons menées ont montré que le code produit par GUML est plus compact que celui produit par trois générateurs de code choisis parmi les générateurs de code les plus utilisés dans le développement des systèmes embarqués (Rhapsody, BridgePoint et iUML). Nos expérimentations ont révélé que les optimisations que GUML effectue ne sont implémentées dans aucun de ces générateurs. Ainsi, la nouvelle approche que nous avons présentée dans ce travail de recherche *la compilation directe des modèles UML* est une approche prometteuse

qui peut améliorer considérablement les performances du code assembleur et par la suite l'exécutable de l'application.

La mise en œuvre de l'approche proposée, qui consiste à ajouter un nouveau front-end pour le langage UML au compilateur GCC peut être considérée comme un avantage puisque le compilateur GCC cible plus de 40 architectures différentes. Cependant, cette dépendance à GCC, limitant l'utilisation de notre compilateur de modèles, peut être aussi considérée comme un inconvénient. Toutefois, cela ne remet aucunement en cause l'approche proposée. GUML, bien qu'il réutilise toutes les optimisations du compilateur GCC n'améliore qu'un seul critère d'optimisation : la réduction de la taille du code. Pour certains types de systèmes embarqués tels que les systèmes embarqués temps réel, le critère d'optimisation du temps d'exécution est aussi important que le critère d'optimisation de la taille du code. L'utilisation de GUML pour compiler les modèles exécutables des systèmes embarqués temps réel et les systèmes embarqués multitâches fait partie des perspectives de cette thèse.

7.2 Perspectives

Les perspectives de ce travail de recherche sont multiples. Nous pouvons penser à enrichir le langage source du compilateur proposé (compiler les machines à états hiérarchiques, les associations entre les classes, etc.), à améliorer l'implémentation du compilateur (l'interfaçage avec *Diversity et Acceleo*, etc.) ou encore à étudier le compromis *réduction temps d'exécution/réduction taille du code*. Cependant, nous avons choisi de développer deux autres perspectives intéressantes de ce travail : l'amélioration de l'optimisation du temps d'exécution et l'utilisation de GUML pour la compilation d'applications multitâches.

7.2.1 GUML et MARTE pour améliorer le temps d'exécution

La réduction du temps d'exécution est un critère d'optimisation très important pour les systèmes temps réel embarqués. L'*inlining* [93] est l'optimisation la plus connue, utilisée par le compilateur GCC pour réduire le temps d'exécution du code. Cette optimisation consiste à remplacer les appels d'opérations (couteux en termes de temps d'exécution) directement par les instructions formants leurs corps. L'*inlining* est l'une des optimisations qui, en diminuant la taille d'exécution d'un programme, peut augmenter de l'autre coté la taille du code binaire résultant. Pour gérer ce problème, le compilateur GCC n'applique pas cette optimisation pour toutes les fonctions. Il se base généralement sur la taille de la fonction appelée : si la fonction contient un nombre important d'instructions, elle ne sera pas *inlinée*. GCC définit ainsi un poids (entre 1 et 10) associé à chaque fonction (1 est la valeur par défaut, 10 est la valeur maximale associée généralement aux fonctions couteuses en terme de taille de code) [94].

Ainsi, pour gérer l'*inlining*, GCC se base sur la taille de la fonction appelée. Toutefois, *inliner* les fonctions périodiques (très fréquentes dans le code implémentant un système embarqué temps réel), de petite période (s'exécutent assez fréquemment) peut minimiser le temps d'exécution du système même si la taille de la fonction périodique dépasse la taille maximale (*max-inline-insns-auto*) à partir de laquelle GCC considère que la fonction

est très grande et qu'il ne faut pas l'*inliner* [94]. GCC se base généralement sur la taille (le nombre d'instructions) des fonctions vu que la période (bien qu'elle puisse être utile pour l'*inlining*) n'est pas facile à détecter à partir du code généré. En effet, implémenter une fonction périodique revient généralement (et selon le langage cible choisi) à ajouter une boucle infinie contenant l'appel à la routine *sleep()* en passant la période comme paramètre à cette routine).

Le compilateur GUML pourrait améliorer l'optimisation de l'*inlining* en prenant en compte la notion de période des fonctions facile à détecter au niveau UML grâce au stéréotype « *RtFeature* » du profil MARTE (*Modeling and Analysis of Real-Time Embedded Systems*) standardisé par l'OMG pour la modélisation et l'analyse des systèmes temps réel embarqués [95]. Le stéréotype « *RtFeature* » est appliqué à un élément UML pour spécifier les propriétés temps réel de cet élément telles que la période, la priorité, le deadline, etc.

7.2.2 GUML et OpenMP pour les applications multitâches

Par définition, les systèmes multitâches doivent gérer le parallélisme et la concurrence de tâches. En UML, deux tâches qui s'exécutent en parallèle se présentent sous formes de deux actions se trouvant entre deux nœuds particuliers d'une activité : le nœud *fork* et le nœud *join*. Actuellement, GUML ne compile que les actions qui s'exécutent séquentiellement. Cependant, pour les systèmes multitâches, la notion de parallélisme de tâches est primordiale. Concrètement, GUML génère du GIMPLE à partir des modèles UML. Pour le calcul parallèle, GIMPLE contient des instructions relatives aux clauses et directives de l'API OpenMP [96]. OpenMP étant une interface qui implémente le modèle « *fork/join*» pour le calcul parallèle, la compilation des *fork* et *join* des diagrammes d'activité reviennent simplement à faire un mapping entre ces deux éléments et les instructions GIMPLE dérivant des directives OpenMP correspondantes. La figure 7.1 présente un diagramme d'activité UML dans lequel deux opérations « op_2 » et « op_3 » s'exécutent en parallèle et un extrait de la forme GIMPLE équivalente.

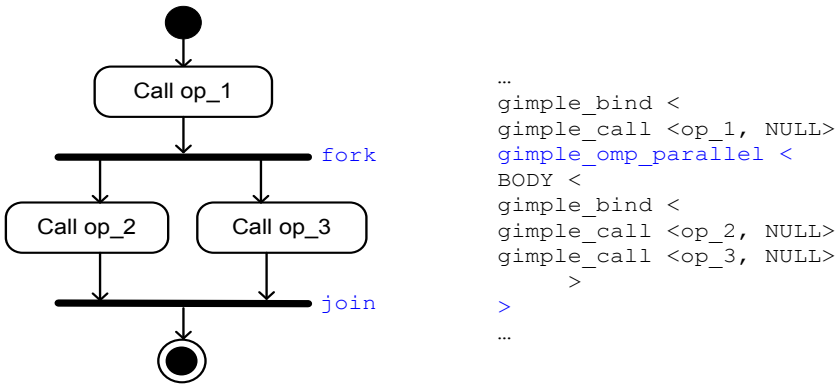


FIGURE 7.1 – Un diagramme d'activité UML contenant un *fork* et un *join* et leurs correspondant en GIMPLE l'instruction *"gimple_omp_parallel"*

L'instruction GIMPLE « *gimple_omp_parallel* » implémente les deux nœuds *fork* et *join* (le début représente le *fork* tandis que la fin représente le *join*). Le corps de cette instruction contient toutes les tâches qui peuvent s'exécuter en parallèle. Un des avantages de la

spécification du parallélisme de tâches en utilisant les activités UML est le fait que le concepteur ne gère ni la synchronisation des threads parallèles ni les mécanismes de gestion des ressources partagées. La compilation directe des diagrammes d'activités contenant des *forks* et des *joins* préserverait (en se basant sur les instructions GIMPLE relatives aux directives OpenMP) cette abstraction de gestion de threads et des resources partagées. La génération de code C++ ou Java à partir des *fork/join*, fera certainement appel à des routines de gestions de données partagées et de synchronisations des threads ce qui complique considérablement le générateur de code.

Dans cette thèse, nous avons proposé la première version du *front-end* GCC pour le langage UML. En réalisant les perspectives décrites ci-avant, GUML pourra bien s'intégrer à GCC à coté des 8 autres *front-ends* distribués par default dans la collection GCC. Cette perspective est une perspective à long termes vu la complexité du langage UML avec le quel, et en changeant les points de variation sémantique, plusieurs sémantiques d'exécution peuvent être associé au même modèle.

ANNEXE A

Implémentation d'une machine à états UML en utilisant les 3 patterns de génération de code NSC, STT et SP

La machine à états considérée est celle de la Figure A.1. Nous avons généré le code en utilisant le générateur de code Acceleo. La machine à états modélise le cycle de vie d'un objet de type MyClass La classe MyClass qui définit les 2 opérations action1() et action2(). Nous présentons ici le comportement de la machine à états implémenté par l'opération process_event(). (Nous ne présentons pas le code généré à partir de la classe ni la fonction main qui se charge d'envoyer les événements).

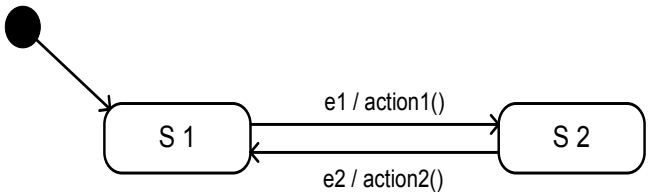


FIGURE A.1 – La machine à états UML implémentée en utilisant les 3 patterns de génération de code NSC, STT et SP

Code C++ généré en utilisant le NSC pattern

```
enum fsm_events          //les événements sont implémentés
{                        // à l'aide d'une énumération
e1,
e2,
};
enum fsm_states          //les états sont implémentés
{                        // à l'aide d'une énumération
S1,
S2,
};
public : fsm_states fsm_currentstate;
public : void process_event(fsm_events evt)
```

```
{
switch (fsm_currentstate)
{
case S1:
        switch (evt)
        {
        case e1:
                action1();
                fsm_currentstate=S2;
        break;
        default:
        break;
        }
break;
case S2:
        switch (evt)
        {
        case e2:
                action2();
                fsm_currentstate=S1;
        break;
        default:
        break;
        }
break;
default :
break;
}
}
```

Code C++ généré en utilisant le STT pattern

```
enum fsm_events
{
        e1,
        e2,
};
enum fsm_states
{
        S1,
        S2,
};
struct fsm_transition
{
        fsm_events trigger;
        fsm_states fromstate;
        fsm_states tostate;
        ptFunc func;
};
public : static const int state_num=2;
public : static const int event_num=2;
public : static const int tr_num=2;
public : fsm_states fsm_currentstate;
public : fsm_transition transitions[tr_num];
public : fsm_transition STT[state_num][event_num];//le tableau à 2 dimensions
transitions[0].trigger=e1;
transitions[0].fromstate=S1;
transitions[0].tostate=S2;
transitions[0].func= & MyClass::action1;
transitions[1].trigger=e2;
transitions[1].fromstate=S2;
transitions[1].tostate=S1;
transitions[1].func= & MyClass::action2;
STT[0][0]= transitions[0];                                    // S1 to S2
STT[1][1]= transitions[1];                                    // S2 to S1
```

```
public : void process_event (fsm_events evt)
{
        doAction(STT[fsm_currentstate][evt].func);
        fsm_currentstate = STT[fsm_currentstate][evt].tostate;}
```

Code C++ généré en utilisant le SP pattern

La Classe Contexte

```
#ifndef CONTEXTE_H_
#define CONTEXTE_H_

class State;
class Contexte
{
public:
        Contexte();
        void e1();              // les événements sont implémentés en méthodes
        void e2();
        void changeState(State* );

private:
        friend class State;
        State* current_state;
};

#endif /*CONTEXTE_H_*/

#include "Contexte.h"
#include "S1.h"
Contexte::Contexte()
{
        current_state = S1::Instance(); // S1 est l'état initial
}
void Contexte::e1()
{
        current_state->e1(this);
}
void Contexte::e2()
{
        current_state->e2(this);
}
void Contexte::changeState(State* s)
{
        current_state= s;
}
```

La Classe State

```
#ifndef STATE_H_
#define STATE_H_
#include "Contexte.h"

class State
{
public:
        virtual void e1(Contexte*)=0;
        virtual void e2(Contexte*)=0;
};

#endif /*STATE_H_*/
```

La Classe S1

```
#ifndef S1_H_
#define S1_H_
#include "State.h"
#include "Operations.h"

class S1 : public State, Operations // les états sont implémentés en classes
{
public:
        typedef void (Operations::*ptFunc)();

public:
        S1();
        static S1* Instance();
        virtual void e1(Contexte*);
        virtual void e2(Contexte*);

private:
        static S1* singleton;
        ptFunc func;

};

#endif /*S1_H_*/
#include "S1.h"
#include "S2.h"
#include "Operations.h"

S1::S1()
{
func= &Operations::action1;
}

void S1::e1(Contexte* t)
{
        (*this.*func)();
        t->changeState(S2::Instance());
}

void S1::e2(Contexte* t){}

S1* S1::singleton = 0;
S1* S1::Instance()
{
        if (singleton == 0)
                singleton = new S1();
        return singleton;
}
```

La Classe S2

```
#ifndef S2_H_
#define S2_H_
#include "State.h"
#include "Operations.h"

class S2 : public State, Operations
{
public:
        typedef void (Operations::*ptFunc)();

public:
        S2();
        static S2* Instance();
        virtual void e2(Contexte*);
        virtual void e1(Contexte*);
```

```
private:
        static S2* singleton;
        ptFunc func;
};
#endif /*S2_H_*/
#include "S2.h"
#include "S1.h"
#include "Operations.h"
S2::S2()
{
func= &Operations::action2;
}

void S2::e2(Contexte* t)
{
        (*this.*func)();
        t->changeState(S1::Instance());
}

void S2::e1(Contexte* t){}

S2* S2::singleton=0;
S2* S2::Instance()
{
        if (singleton == 0)
                singleton = new S2();
        return singleton;
}
```

ANNEXE B

Points de variation sémantique et Points de sémantique non définie fixés pour GUML

La norme UML contient plusieurs points de variation sémantique [29] [30]. Nous n'avons fixé que les points de variation sémantique qui sont en rapports avec les éléments syntaxiques du langage source de GUML notamment les machines à états et les activités. [30] a défini 29 ambigüités dans la définition de la sémantique des machines à états UML en considérant plusieurs éléments syntaxiques (les points d'entrée et sorties, les états composites, les pseudo-états : *History*, *Fork*, *Choice*, etc.). Nous n'avons fixé que 3 points de variation sémantique puisque la première version de GUML ne prend en compte que les machines à états simples ne contenant pas des pseudos états autre que l'état initial. Nous avons également défini la sémantique de 4 points de sémantique non définie dans UML (contrairement aux points de variation sémantique, UML ne fournit pas de choix d'interprétation pour les points de sémantique non définie).

Elément Syntaxique	Imprécision UML	Nature	Sémantique d'exécution GUML
Transition	UML ne fixe pas le choix de la transition à franchir si un événement déclenche plus qu'une transition de garde valide.	Point de variation	GUML prend la première transition de garde valide et ignore les autres.
StateMachine	UML ne fixe pas la politique de sélection (FIFO, LIFO, etc.) des évènements dans la pile à évènements attachée à chaque machine à états	Point de variation	GUML considère que tout événement envoyé est automatiquement reçu et traité par l'objet. La pile à événements est de taille 1.
StateMachine	UML ne fixe pas la politique de gestion des événements ne déclenchant pas de transitions.	Point de variation	GUML ignore les événements non déclencheurs de transitons
SendSignalAction	UML ne définit ni le protocole d'envoi du signal, ni le temps que celui-ci prend pour arriver à l'objet receveur.	Non définie	Le temps mis par le signal pour arriver à l'objet receveur est nul. L'envoi de signal est implémenté par un appel à une méthode de l'objet receveur.
Behavior (Transition Effect)	La manière de transmettre le comportement, les paramètres et le temps de transmission sont indéfinis	Non définie	GUML ne supporte que les appels d'opérations. L'implémentation est assurée par un appel à l'opération correspondante avec les bons paramètres. Le temps de transmission est nul.
CallOperationAction	Le mécanisme déterminant la méthode invoquée comme résultat d'un appel d'opération est indéfini.	Non définie	La méthode appelée comme résultat d'un appel d'opération porte le même nom que cette opération.
AddStructural FeatureValueAction	La sémantique est indéfinie si la valeur ajoutée viole la multiplicité de la propriété structurelle	Non définie	L'ajout d'une valeur violant la multiplicité n'est pas autorisé

Tableau B.1 – Les points de variation sémantique et les points de sémantiques non définie fixés par GUML

ANNEXE C

Conservation du comportement de la machine à états en exécutant l'optimisation *factorisation des nœuds de test*

L'optimisation de la factorisation des nœuds de test est une optimisation implémentée sur la forme intermédiaire G-Graph pour éliminer les nœuds *semi-isomorphes* même en présence d'un nœud de la *false_branche* possédant deux parents. Pour l'exemple de la machine à états de la figure 5.1, cette optimisation comporte 3 transformations intermédiaires de la forme G-Graph jusqu'à atteindre une forme G-Graph optimisée. Pour s'assurer que cette optimisation est valide (conserve le comportement de la forme G-Graph initiale), nous allons montrer que chaque étape intermédiaire de cette optimisation conserve à son tour le comportement du G-Graph. Le code C++ généré à partir du G-Graph initial de la figure 5.12 (avant l'exécution de l'optimisation « factorisation des nœuds de test » est le suivant :

Code 0 :

```
1       if (S1==S1)
2       {
3               If (evt==e1)
4                       S= S2; exit;
5               else if (evt ==e4)
6                       S= S3; exit;
7       }
8       else if (S1==S2)
9       {
10              if (evt==e1)
11                      S= S2; exit;
12              else if (evt ==e4)
13                      S= S3; exit;
14              else if (evt ==e3)
15                      S= S1; exit;
16              else if (evt ==e2)
```

```
17                      S= S2; exit;
18      }
19      else if (S1==S3)
20      {
21              if (evt ==e3)
22                      S= S1; exit;
23              else if (evt ==e2)
24                      S= S2; exit;
25      }
```

Nous allons considérer après chaque étape intermédiaire de l'optimisation le code C++ généré et montrer que le comportement du code généré n'a pas changé.

Etape 1 : La duplication d'un nœud de test de type State_Cond et l'écartement du nœud de test de type Event_Cond (ayant deux parents de type différents) en liant ses arcs entrants au nouveau nœud dupliqué. La figure 5.14 (a) correspond au résultat de cette étape intermédiaire. Le code C++ généré est le suivant :
Code 1 :

```
1       if (S1==S1)
2       {
3               if (evt==e1)
4                       S= S2; exit;
5               else if (evt ==e4)
6                       S= S3; exit;
7       }
8       else if (S==S2)
9       {
10              if (evt==e1)
11                      S= S2; exit;
12              else if (evt ==e4)
13                      S= S3; exit;
14      }
15      if (S==S2)
16      {
17              if (evt ==e3)
18                      S= S1; exit;
19              else if (evt ==e2)
20                      S= S2; exit;
21      }
22      else if (S1==S3)
23      {
24               if (evt ==e3)
25                      S= S1; exit;
26              else if (evt ==e2)
27                      S= S2; exit ;
28      }
```

Nous allons monter que Code 1 est équivalent à Code 0 en déterminant tous les chemins d'exécution possibles de Code 1 (en particulier les instructions des lignes 8 à 21) et ceux du Code 0 (en particulier les instructions des lignes 8 à 18).
Tous les chemins possibles d'exécution des instructions de la ligne 8 à la ligne 18 du Code 0 sont :
1) Cas où S ==S1 est vrai

```
S ==S1 vrai,  evt==e1 vrai, S= S2, exit
S ==S1 vrai,  evt==e1 fausse,  evt==e4 vrai,  S= S3, exit
S ==S1 vrai,  evt==e1 fausse,  evt==e4 fausse,  evt==e3 vrai, S= S1, exit
S ==S1 vrai,  evt==e1f ausse,  evt==e4 fausse,  evt==e3 fausse,  evt==e2 vrai, S= S2, exit
```

```
S ==S1 vrai, evt==e1 fausse, evt==e4 fausse, evt==e3 fausse,
evt==e2 fausse, évaluer (S==S2)
```

2) Cas où S ==S1 est fausse

```
S ==S1 fausse, évaluer (S==S2)
```

Tous les chemins possibles d'exécution des instructions de la ligne 8 à la ligne 21 du Code 1 sont :

1) Cas où S ==S1 est vrai

```
S ==S1 vrai, evt==e1 vrai , S= S2, exit
S ==S1 vrai, evt==e1 fausse, evt==e4 vrai, S= S3, exit
S ==S1 vrai, evt==e1 fausse, evt==e4 fausse, S ==S1 vrai , evt==e3 vrai, S= S1, exit
S ==S1 vrai, evt==e1 fausse, evt==e4 fausse, S ==S1 vrai , evt==e3 fausse,
evt==e2 vrai, S= S2, exit
S ==S1 vrai, evt==e1 fausse, evt==e4 fausse, S ==S1 vrai, evt==e3 fausse,
evt==e2 fausse, évaluer (S==S2)
```

2) Cas où S ==S1 est fausse

```
S ==S1 fausse, S ==S1 fausse, évaluer (S==S2)
```

On peut conclure que Code 1 est équivalent à Code 0 vu qu'ils ont tous les deux les mêmes chemins d'exécution.

Etape 2 : La fusion des deux nœuds de type State_Cond (if S==S2)) et (if (S==S3)) en utilisant l'opérateur logique OU. Le code C++ généré suite à cette transformation est le suivant :

Code 2 :

```
1       if (S1==S1)
2       {
3               if (evt==e1)
4                       S= S2; exit;
5               else if (evt ==e4)
6                       S= S3; exit;
7       }
8       else if (S==S2)
9       {
10              if (evt==e1)
11                      S= S2; exit;
12              else if (evt ==e4)
13                      S= S3; exit;
14      }
15      if ((S==S2) || (S==S3))
16      {
17              if (evt ==e3)
18                      S= S1; exit;
19              else if (evt ==e2)
20                      S= S2; exit;
21      }
```

Montrer que Code 2 est équivalent à Code 1 revient à montrer que le bloc d'instructions (de la ligne 15 à la ligne 28 du Code 1) est équivalent au bloc d'instructions (de la ligne 15 à la ligne 21 du Code 2), qui est évident par la définition même de l'opérateur logique OU.

Etape 3 : Éliminer les nœuds isomorphes (Figure 5.14 (b)) et factoriser les deux nœuds de type State_Cond. Le code généré après cette transformation est le suivant :

Code 3 :

```
1       if (S1==S1) || (S==S2)
2       {
3               if (evt==e1)
4                       S= S2; exit;
5               else if (evt ==e4)
6                       S= S3; exit;
7       }
8       if ((S==S2) || (S==S3))
9       {
10              if (evt ==e3)
11                      S= S1; exit;
12              else if (evt ==e2)
13                      S= S2; exit;
14      }
```

Montrer que Code 3 est équivalent à Code 2 revient à montrer que le bloc d'instructions (de la ligne 1 à la ligne 14 du Code 2) est équivalent au bloc d'instructions (de la ligne 1 à la ligne 7 du Code 3), qui est évident par la définition même de l'opérateur logique OU.

Bibliographie

[1] France, R. and Rumpe, B. *Model-driven Development of Complex Software : A Research Roadmap*. Future of Software Engineering (FOSE '07). IEEE Computer Society, Washington, DC, USA, 2007, (Cité page 6).

[2] Krall, A. *Efficient JavaVM just in time compilation*. Parallel Architectures and Compilation Techniques (PACT '98). IEEE Computer Society, Washington, DC, USA,1998, (Cité page 6).

[3] Von Hagen, W. *The Definitive Guide to GCC, Second Edition*. Apress, 2006, (Cité page 6).

[4] OMG. *Unified Modeling Language (OMG UML) Superstructure, Version 2.3*. formal/2010-05-05, (Cité pages 7, 13 et 37).

[5] Jacobson, I. *keynote address to CMP Media's UML World conference in New York City*. http ://drdobbs.com/architecture-and-design/184414764, 2001, (Cité page 9).

[6] Mellor, S. and Balcer, M. *Executable UML : A Foundation for Model Driven Architecture*. Addison Wesley, ISBN 0-201-74804-5, 2002, (Cité pages 9, 17 et 33).

[7] Selic, B. *New Methods and Tools for Developing Real-Time Software*. 12th IEEE Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, pp. 177-178, USA, 2009, (Cité page 9).

[8] Lugato, D et, a. *Validation and automatic test generation on UML models : the AGATHA approach*. Int.J. Softw. Tools Technol. Transf, pp124–139, 2004, (Cité pages 12, 37 et 67).

[9] Mraidha, C., Tucci-Piergiovanni, S., and Gerard, S. *Optimum : a MARTE-based methodology for schedulability analysis at early design stages*. SIGSOFT Softw. Eng. Notes, vol 36, issn 0163-5948, ACM, New York, USA, 2011, (Cité page 12).

[10] Woodside, C.M et, a. *Performance Analysis of Security Aspects by Weaving Scenarios Extracted from UML Models*. The Journal of Systems and Software Special Issue, Vol.82, pp.56-74, 2009, (Cité page 12).

[11] Balarin, F et, a. *Hardware-Software Co-Design of Embedded Systems : The POLIS Approach*. Kluwer Academic Publishers, 1997, (Cité pages 12, 33, 34, 35, 83, 86, 87 et 92).

[12] GASPARD2. *Graphical Array Specification for Parallel and Distributed Computing*. Equipe DaRT, INRIA Lille, http ://www.gaspard2.org, (Cité page 12).

[13] Aho, A., Lam, M., Sethi, R., and Ullman, J. *Compilateurs : principes, techniques et outils*. ISBN 2-7296-0295-X, InterEdition Paris, 1989, (Cité pages 12, 24 et 25).

[14] Herrington, J. *Code Generation in Action*. ISBN = 1930110979, Manning Publications Co. Greenwich, CT, USA, 2003, (Cité page 13).

[15] MacDonald, A., Russell, D., and Atchison, B. *Model-driven Development within a Legacy System : An industry experience report*. Software Engineering Conference, pp 14 - 22, 2005, (Cité page 13).

[16] Mraidha, C. *Modélisation exécutable et analyse de propriétés temps réel*. Thèse de doctorat, Université d'Evry, France, 2005, (Cité page 13).

[17] Schattkowsky, T. *Platform-independent modeling of synthesizable software systems using UML 2*. ISBN 9783899598520, Der Andere Verlag, 2009, (Cité pages 13 et 17).

[18] Bock, C. *Three Kinds of Behavior Model*. Journal Of Object-Oriented Programming Vol 12, No 4, July, 1999, (Cité page 13).

[19] Harel, D. *A visual formalism for complex systems*. Sci. Comput. Program. 8, 3, 1987, (Cité page 14).

[20] Wagner, F., Schmuki, R., Wagner, T., and Wolstenholme, P. *Modeling Software with Finite State Machines : A Practical Approach*. ISBN 0849380863, Auerbach Publications, 2006, (Cité page 14).

[21] Harel, D. and Naamad, A. *The STATEMATE semantics of statecharts*. ACM Trans. Softw. Eng. Methodol. 5, 4 pp 293-333, 1996, (Cité page 14).

[22] Martin, G. *UML for Embedded Systems Specification and Design : Motivation and Overview*. Design, Automation and Test in Europe (DATE), 2002, (Cité page 14).

[23] Davis, A. and Keller, R. *Data flow program graphs*. IEEE Computer. 15, 2, pp 26-41, 1982, (Cité page 15).

[24] Petri, C. A. *Communication with automata*. Volume 1 Supplement 1, AD0630125, 1966, (Cité page 15).

[25] Estrin, G. and Turn, R. *Automatic Assignment of Computations in a Variable Structure Computer System*. IEEE Transactions on Electronic Computers vol 12, no.6 pp 755-773, 1963, (Cité page 15).

[26] Lee, A. and Messerschmitt, D. G. *Static scheduling of synchronous data flow programs for digital signal processing*. IEEE Trans. Computers, 1987, (Cité page 15).

[27] OMG. *Semantics of a Foundational Subset for Executable UML Models (fUML), v1.0*. formal/2011-02-01, (Cité page 16).

[28] OMG. *Action Language for Foundational UML (Alf) Concrete Syntax for a UML Action Language, 2010*. (Cité page 16).

[29] Chauvel, F. and Jézéquel, J. M. *Code generation from UML Models with semantic variation points*. MoDELS, LNCS. Springer Verlag, 2005, (Cité pages 16 et 129).

[30] Fecher, H., Schönborn, J., Kyas, M., and Willem, P. D. *29 new unclarities in the semantics of UML 2.0 state machines*. Formal Engineering Methods Springer, Berlin, Allemagne, 2005, (Cité pages 16, 65 et 129).

[31] Raistrick, C., Francis, P., Wright, J., C.Carter, and Wilkie, I. *Model Driven Architecture with Executable UML*. ISBN 9780521537711, Cambridge University Press, 2004, (Cité page 17).

[32] Abstract-Solution. *ASL Abstract Specification Language*. http ://www.kc.com/XUML/asl.php, 2011, (Cité page 17).

[33] OAL. *Object Action Language*. http ://www.ooatool.com/docs/OAL08.pdf, 2002, (Cité page 17).

[34] Pathfinder-Solutions. *PAL : Platform Independent Action Language*. http ://www.ooatool.com/docs/ PAL04.pdf, 2004, (Cité page 17).

[35] Mraidha, C., Gérard, S., Tanguy, Y., Dubois, H., and Schnekenburger, R. *Action Language Notation for Accord/UML*. Technical report, CEA, 2005, (Cité page 17).

[36] Mens, T., Czarnecki, K., and VanGorp, P. *A taxonomy of model transformation*. Dagstuhl Seminar on Language Engineering for Model-Driven Software Development, 2005, (Cité pages 18 et 36).

[37] OMG. *MOF QVT final adopted specification*. http ://www.omg.org/docs/ptc/05-11-01.pdf, 2005, (Cité page 18).

[38] Jouault, F. *xModel Transformation with ATL*. MtATL Workshop, Nantes, France, 2009, (Cité page 18).

[39] OMG. *MOF Model to Text Transformation Language, v1.0*. formal/2008-01-16, (Cité page 18).

[40] Musset, J., Juliot, E., and Lacrampe, S. *AcceleoTM 2.2 : Guide utilisateur. Obeo, http ://www.acceleo.org/pages/accueil/fr, 2008*. (Cité page 18).

[41] Klatt, B. *Xpand : A Closer Look at the model2text Transformation Language*. Chair of Software Design and Quality at the University of Karlsruhe, 2007, (Cité page 18).

[42] Oldevik, J. *MOFScript Eclipse Plug-In : Metamodel-Based Code Generation*. Eclipse Technology eXchange workshop (eTX), Nantes, France, 2006, (Cité page 18).

[43] JET. *Java Emission Template*. http ://eclipse.org/articles/Article-JET/jet-tutorial.html, 2004, (Cité page 18).

[44] *http ://publib.boulder.com.ibm.help.download.rhapsody.doc/pdf/codegenc.pdf, 2008*. Rhapsody Code Generation Guide, (Cité page 18).

[45] Eichberg, M. *MDA and Programming Languages*. Workshop Generative Techniques in the context of Model-Driven Architecture, OOPSLA, 2002, (Cité pages 19 et 23).

[46] Samek, M. *Practical UML Statecharts in C/C++, second Edition Event-Driven Programming for Embedded Systems*. Newnes, 2008, (Cité page 19).

[47] Metz, P., O'Brien, J., and Weber, W. *Code Generation Concepts for Statecharts Diagram of UML v1.1*. Object Technology Group (OTG) Conference, University of Vienna, Austria, 1999, (Cité page 19).

[48] Martin, R. C. *UML Tutorial : Finite State Machines, C++ Report*. 1998, (Cité page 19).

[49] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns : Elements of Reusable Object-Oriented Software,*. Massachusetts : Addison-Wesley, 1995, (Cité pages 19 et 21).

[50] VanGurp, J. and Bosch, J. *On the implementation of finite state machines*. International Conference Software Engineering and Applications, 1999, (Cité pages 19 et 23).

[51] Gottschling, P. and Lumsdaine, A. *Integrating Semantics and Compilation Using C++ concepts to develop robust and efficient reusable libraries*. In Generative Programming And Component Engineering (GPCE), Nashville, Tennessee, USA, 2008, (Cité page 23).

[52] BridgePoint. *http ://www.mentor.com/products/sm/model-development/bridgepoint/*. (Cité pages 24 et 64).

[53] Rose-rt. *Rational Rose Real Time http ://www.ghs.com/partners/rational/rose-rt.pdf*. (Cité page 24).

[54] Clang. *LLVM and Clang : Advancing Compiler Technology*. http ://llvm.org/pubs/2011-02-FOSDEM-LLVMAndClang.pdf, 2011, (Cité page 24).

[55] Davidson, J. W. and Fraser, W. *The Design and Application of a Retargetable Peephole Optimizer*. ACM Transactions on Programming Languages and Systems, Volume2, 191-202, 1980, (Cité page 27).

[56] Novillo, D. *Tree SSA A New Optimization Infrastructure for GCC*. GCC Developers Summit, Ottawa, Ontario Canada, 2003, (Cité page 30).

[57] Cytron, R., Ferrante, J., Rosen, B., Wegman, M., and Zadeck, K. *Efficiently computing static single assignment form and the control dependence graph*. ACM Transactions on Programming Languages and Systems, vol 13(4), pp451-490, 1991, (Cité pages 30, 31 et 32).

[58] Merrill, J. *GENERIC and GIMPLE : A New Tree Representation for Entire Functions*. GCC Developers Summit, Ottawa, Ontario Canada, 2003, (Cité page 30).

[59] Hendren, L., Donawa, C., Emami, M., Gao, G., Justiani, B., and Sridharan, B. *Designing the McCAT Compiler Based on a Family of Structured Intermediate Representations*. Lecture Notes in Computing Science no. 757 pp 406-420, 1992, (Cité page 30).

[60] Naishlos, D. *Autovectorization in GCC*. GCC Developers Summit, Ottawa, Ontario Canada, 2004, (Cité page 32).

[61] Glek.T and Hubicka.J. *Optimizing real world applications with GCC Link Time Optimization*. journal CoRR vol. abs/1010.2196, 2010, (Cité page 32).

[62] Hubicka, J. *Interprocedural optimization framework in GCC*. GCC Developers Summit Ottawa, Ontario Canada, 2007, (Cité page 32).

[63] Charfi, A., Mraidha, C., Boulet, P., Gérard, S., and Terrier, F. *Toward optimized code generation through model-based optimization. :*. Design Automation and Test in Europe (DATE), EDAA , Belgium, 2010, (Cité pages 32, 58, 62 et 118).

[64] Bjorklund, D. *A Kernel Language for Unified Code Synthesis*. Thèse de doctorat, Abo Akademi University, 2005, (Cité pages 33, 35, 36, 83, 86, 90, 92 et 93).

[65] Lilius, D. B. J. and Porres, I. *Towards Efficient Code Synthesis from Statecharts*. Practical UML-Based Rigorous Development Methods, Countering or Integrating the eXtremists. Workshop of the pUML-Group Lecture Notes in Informatics P-7, Toronto,Canada, 2001, (Cité page 33).

[66] *SCOPE : A Statechart COmPilEr*. http ://www.itu.dk/people/wasowski/projects/scope/, (Cité page 33).

[67] Wasowski, A. *On efficient program synthesis from statecharts*. Language, Compiler, and Tool for Embedded Systems (LCTES), ACM, USA, 2003, (Cité page 33).

[68] Schlabe, D., Knostmann, T., and Bünte, T. *A Scade Suite to Modelica Interface*. Modelica Conference, Dresden, Germany, 2011, (Cité pages 34 et 64).

[69] M Chiodo et, a. *Synthesis of Software Programs for Embedded Control Applications*. ACM/IEEE Design Automation Conference, 1995, (Cité page 34).

[70] Rudell, R. *Dynamic variable ordering for ordered binary decision diagrams*. International Conference on Computer-Aided Design, 1993, (Cité page 35).

[71] Maddeh, M., Romdhani, M., and Ghedira, K. *Classification des approches de refactorisation des modèles*. 4èmes Journées sur l'Ingénierie Dirigée par les Modèles (IDM 08) Mulhouse, France, 2008, (Cité page 36).

[72] Demeyer, S. *Maintainability versus Performance : What's the Effect of Introducing Polymorphism*. Technical Report, University of Antwerp, Belgium, 2002, (Cité page 37).

[73] Biermann, E., Ermel, C., and Taentzer, G. *Precise Semantics of EMF Model Transformations by Graph Transformation*. MoDELS, Springer, Heidelberg, 2008, (Cité page 37).

[74] Dobrzanski, L. *UML Model Refactoring Support for Maintenance of Executable UML Models*. Master thesis, Blekinge Institute of Technology, Sweden, 2005, (Cité page 37).

[75] Berstel, J., Boasson, L., Carton, O., and Fagnot, I. *Minimization of Automata*. Journal CoRR vol . abs/1010.5318, 2010, (Cité page 37).

[76] Clarke, L. A. *A System to Generate Test Data and Symbolically Execute Programs*. J. IEEE Trans. Softw. Eng. 2. 3, pp. 215–222, 1976, (Cité pages 37 et 67).

[77] Charfi, A., Mraidha, C., Boulet, P., Gérard, S., and Terrier, F. *Does code generation promote or prevent optimizations ?* IEEE Symposium Object/Component/Service-Oriented Real-Time Distributed Computing, USA, 2010, (Cité pages 38 et 118).

[78] Riehle, D., Fraleigh, S., Bucka-Lassen, D., and Omorogbe, N. *The architecture of a uml virtual machine*. OOPSLA, ACM, New York, 2001, (Cité page 38).

[79] H Wada et, a. *Design and implementation of the matilda distributed uml virtual machine*. Software Engineering Applicattions (SEA), USA, 2006, (Cité page 38).

[80] Crane, M. L. and Dingel, J. *Towards a UML virtual machine : implementing an interpreter for UML 2 actions and activities.* Center for Advanced Studies on Collaborative research. pp. 96–110 ACM, USA, 2008, (Cité page 38).

[81] Gurov, V. S., Mazin, M. A., Narvsky, A. S., and Shalyto, A. *Tools for support of automata-based programming*. Journal. Program. Computer. Softw. 33, 6. 343–355, 2007, (Cité page 38).

[82] Schattkowsky, T. and Muller, W. *A UML virtual machine for embedded systems.* International Conference on Information Systems, USA, 2005, (Cité page 38).

[83] Harel, D. and Rumpe, B. X. *Meaningful Modeling : What's the Semantics of 'Semantics' ?* Computer 37(10), 64-72, 2004, (Cité page 42).

[84] OMG. *Object Constraint Language OMG Available Specification*. Version 2.0 formal/06-05-01, (Cité page 42).

[85] Ridoux, O. *Cours de Compilation*. http ://www.irisa.fr/lande/ridoux/ENS/COMP/, 2006, (Cité page 50).

[86] Charfi, A., Gamatié, A., Honoré, A., Dekeyser, J. L., and Abid, M. *Validation de modèles dans un cadre d'IDM dédié à la conception de systèmes sur puce*. 4èmes Journées sur l'Ingénierie Dirigée par les Modèles (IDM 08) Mulhouse, France, 2008, (Cité page 50).

[87] Grosu, R., Huang, X., Jain, S., and Smolka, S. A. *Open-source model checking*. Workshop on Software Model Checking (SoftMC), 2005, (Cité page 50).

[88] Donnelly, C. and Stallman, R. *Bison : The Yacc compatible Parser Generator Bison*. manuelle d'utilisation, ISBN 1-882114-44-2, Free Software Foundation, Boston, USA, 2011, (Cité page 52).

[89] Zeng, K., Guo, Y., and Angelov, C. K. *Graphical model debugger framework for embedded systems*. Design, Automation and Test in Europe (DATE). European Design and Automation Association, Belgium, 2010, (Cité page 64).

[90] J DeAntoni et, a. *RT-Simex : retro-analysis of execution traces*. Foundation of Software Engineering (FSE), 2010, (Cité page 64).

[91] Rodrigues, A. W., Aranega, V., Etien, A., Guyomarch, F., and Dekeyser, J. L. *Enabling Traceability in MDE to Improve Performance of GPU Applications*. Rapport de recherche N RR-7720, équipe DaRT INRIA Lille, 2011, (Cité page 64).

[92] Starynkevitch, B. *MELT a Translated Domain Specific Language Embedded in the GCC Compiler*. IFIP Working Conference on Domain-Specific Languages, Bordeaux France, 2011, (Cité page 64).

[93] Dean, J. and Chambers, C. *Towards better inlining decisions using inlining trials*. ACM conference on LISP and functional programming. ACM, New York, USA, 1994, (Cité page 119).

[94] Hubicka, J. *The GCC call graph module a framework for inter-procedural optimization*. GCC Developers'Summit Ottawa, Ontario Canada, 2004, (Cité pages 119 et 120).

[95] OMG. *UML Profile for MARTE : Modeling and Analysis of Real-Time Embedded Systems, version 1.1*. http ://www.omg.org/spec/MARTE/1.1, 2011, (Cité page 120).

[96] Novillo, D. *OpenMP and automatic parallelization in GCC*. GCC Developers' Summit Ottawa, Ontario Canada, 2006, (Cité page 120).