État de l'Art et Positionnement

SOMMAIRE

2.1 Introduction

Les systèmes TR²E retiennent de plus en plus l'attention du fait de leur introduction massive dans les domaines de notre quotidien, que ce soit dans nos loisirs, pour notre sécurité ou dans les transports (baladeurs mp3, géolocalisation mobile, voiture, contrôleur de vitesse d'un train, etc).

L'augmentation de leurs domaines d'application, de la complexité des architectures logicielles proposées, des critères de qualité de service attendue mais surtout du niveau de sécurité et de sûreté de fonctionnement requis pour la fiabilité du système, définit un ensemble

de besoins qui justifient un intérêt tant auprès des chercheurs que des industriels. Ainsi, de nombreux travaux de recherche ont porté sur l'amélioration des méthodes de conception des systèmes TR^{2}E critiques et, plus particulièrement, se sont intéressés à l'automatisation de leur analyse, de leur production et de leur déploiement, ceci afin d'augmenter la productivité des équipes de développement et d'assurer la fiabilité du système, notamment en limitant les interventions humaines lors des étapes *sensibles* du développement logiciel.

L'objectif principal de ces travaux vise la séparation des préoccupations et la décomposition claire des exigences fonctionnelles - *i.e.* ce que réalise le système - et non-fonctionnelles- *i.e.* de quelle manière il le réalise - de l'application. Ainsi, la partie applicative du système est développée indépendamment de la plate-forme cible sur laquelle il s'exécute.

Pour ce faire, des techniques de conception logicielle ont été développées tels les *design patterns*, les profils architecturaux, les preuves de code, les tests définissant la construction d'application déterministe et *correcte-par-construction*. Les intergiciels ont été conçus pour faciliter la mise en œuvre de cette séparation et définissent une couche intermédiaire d'adaptation entre le code applicatif et le code spécifique au support d'exécution.

L'introduction de l'approche par composants a permis d'accroître la décomposition fonctionnelle en offrant une séparation claire entre spécification et implantation des composants. La conception est ainsi facilitée par la construction d'un assemblage de composants préexistants, déployés, configurés et réalisant un sous-ensemble des fonctionnalités du système.

Les descriptions architecturales établies à partir des langages de descriptions d'architectures (ADLs) ont permis d'automatiser les étapes de déploiement et de configuration des composants. Pour améliorer cette solution, les techniques induites par l'*Ingénierie Dirigée par les Modèles* (IDM) ont permis l'automatisation des étapes d'analyse, de la validation du système et de la production du code source des composants.

De nombreuses approches visent à combiner ces solutions pour la production de systèmes TR^{2}E déterministes, *correcte-par-construction* et fiables. De nouveaux processus de conception, d'analyse et d'implantation sont ainsi proposés utilisant des standards, définissant des règles de conception et de développement rigoureuses, proposant la spécification et la vérification de contraintes de réalisation, utilisant des procédures de test exhaustives et plus récemment introduisant de nouvelles méthodes d'analyse basées sur des méthodes formelles.

Toutefois, cette combinaison n'est pas triviale et se heurte à différentes limites que nous abordons dans ce chapitre.

Ce chapitre présente les éléments majeurs requis pour la conception, l'implantation et la validation de systèmes TR^{2}E critiques. Nous présentons et évaluons différentes approches qui ont abordé une ou plusieurs problématiques décrites précédemment. Puis, nous discutons des limites de ces approches vis-à-vis de notre problématique générale. Enfin, nous présentons les grandes lignes de nos travaux de recherche.

2.2 Conception logicielle des systèmes TR^{2}E critiques

Cette section présente différentes techniques pour la conception, l'analyse et l'implantation des architectures logicielles d'un système TR^{2}E critique selon une approche dirigée par les modèles et assistée par ordinateur.

La modélisation d'une application permet de raisonner *à priori* sur celle-ci - *i.e.* avant son implantation. Le "modèle" sert de point d'entrée pour les analyses architecturales et comportementales et la validation du système vis-à-vis de sa spécification. Enfin, il permet la génération

du code source des composants logiciels modélisés et automatise leur configuration et leur déploiement.

2.2.1 Description des systèmes critiques

Différentes approches pour la description architecturale des systèmes distribués ont déjà été présentées dans des travaux antérieurs de notre équipe [Zalila, 2008; Borde, 2009]. Parmi les différentes techniques pour la modélisation de système, nous nous intéressons, dans cet état de l'art, essentiellement à celles dédiées aux systèmes TR²E critiques.

Langages de description d'architecture

Les Langages de Description d'Architecture (ADLs) ont pour objectif de décrire un système sous une forme structurée par assemblage de composants logiciels et matériels ainsi que leurs interactions. Ils s'appuient sur trois concepts fondamentaux [Medvidovic and Taylor, 2000] :

- le composant, qui représente une unité indépendante de calcul, de stockage de données ou d'encapsulation ;
- le connecteur, qui définit explicitement l'interaction et la sémantique de l'interaction entre un ou plusieurs composants ;
- la configuration, qui définit les configurations structurelles (ensemble de composants et de connecteurs d'une architecture propre à une application) et des configurations comportementales (évolution des liens, des communications et des propriétés non fonctionnelles entre composants et connecteurs).

Nous pouvons distinguer différents types d'ADLs en fonction des objectifs qu'ils visent :

- les ADLs génériques, pour la conception des systèmes logiciels en général ;
- les ADLs formels, pour l'analyse des systèmes ;
- les ADLs pour la conception d'un système spécifique à un domaine.

Bien que les ADLs formels (ACME [Garlan *et al.*, 1997], RAPIDE [Luckham *et al.*, 1995], WRIGHT [Allen, 1997] soient dédiés à l'analyse des systèmes, la plupart d'entre eux expriment des limites communes pour la configuration de l'application à partir de leur description et supportent difficilement une démarche de production automatique du code de l'application. Dans le cadre de nos travaux, nous nous intéressons uniquement aux ADLs spécifiques à un domaine comme le langage AADL dans sa première version [SAE Aerospace, 2004].

AADL 1.0. Le langage AADL (Architecture, Analysis and Design Language) [SAE Aerospace, 2004], dans sa première version, a été développé pour décrire l'architecture logicielle et matérielle d'un système TR²E au sein d'une unique représentation - *i.e.* un même modèle. Les motivations principales du langage sont l'analyse du système et la représentation du déploiement de l'architecture logicielle sur la plate-forme matérielle. Les travaux de [Zalila, 2008] étendent le périmètre d'activité du langage AADL en proposant un processus de conception autorisant la production automatique du code source des composants logiciels de l'application à partir de leur description architecturale en AADL 1.0.

Discussion. Le langage AADL dans sa première version fait partie de la catégorie des ADLs et est particulièrement adapté à la conception des systèmes TR²E critiques. Le modèle architectural final ne contient que des composants concrets modélisant des composants tels qu'ils apparaissent physiquement dans le système final. Un mécanisme efficace de propriétés

permet de spécifier pour chaque composant un ensemble d'exigences fonctionnelles ou non fonctionnelles notamment liées au caractère temps-réel, distribué et critique de l'application. Ainsi, il est possible d'automatiser la configuration et le déploiement des composants à partir de l'analyse d'un modèle AADL.

Langages de modélisation spécifique à un domaine

Aujourd'hui, l'ingénierie du logiciel s'oriente vers l'*Ingénierie Dirigiée par les Modèles* (IDM) et le principe du "tout est modèle". L'IDM vise de manière plus radicale que pouvaient l'être les approches de *design-patterns* [Gamma *et al.*, 1995] et des *aspects* [Kiczales and Hilsdale, 2001], à fournir des modèles complémentaires exprimant séparément une préoccupation (point de vue sur le système, étape de modélisation, etc) [Combemale, 2008].

Si la notion de *modèle* est le concept central de l'IDM, celle-ci introduit la méta-modélisation qui consiste à définir un ensemble de règles et de concepts visant la modélisation d'un ensemble de problèmes déterminés voir même spécialisés. L'IDM favorise ainsi la définition de langages de modélisation dédiés à un domaine particulier (*Domain Specific Modeling Languages* - DSML), offrant ainsi aux utilisateurs des concepts propres à leur métier.

L'approche MDA. L'OMG a adopté une approche de modélisation et a défini le MDA (*Model Driven Architecture* [Soley, 2000; OMG, 2003b]) pour promouvoir les bonnes pratiques de modélisation, exploiter les avantages des modèles en terme de pérennité, de productivité, d'interopérabilité et de prendre en compte certaines caractéristiques des plate-formes d'exécution. Pour cela, le MDA s'appuie sur plusieurs standards UML [OMG, 2007c; OMG, 2007d], MOF [OMG, 2006c] et XMI [OMG, 2007b]. L'objectif majeur du MDA est l'élaboration de modèles indépendants des détails techniques des plate-formes d'exécution (*Platform Independent Model* - PIM) et de générer de manière automatique la totalité des modèles de code (*Platform Specific Model* - PSM) par transformation de modèle [Blanc, 2005; Kleppe *et al.*, 2003].

Le profil UML/MARTE. Dans le contexte des systèmes temps-réel, le profil UML/MARTE (*Modeling and Analysis of Real-Time and Embedded systems* [OMG, 2007e]) a été défini et standardisé pour modéliser et analyser les systèmes TR^{2}E à l'aide du langage UML.

Définition 2.1 (Profil UML [OMG, 2007c]) *Un profil est un moyen d'extension d'un méta-modèle permettant de spécialiser une méta-classe pour répondre à des besoins de modélisation spécifiques.*

Selon la définition de la notion de profil UML donnée ci-dessus, MARTE spécialise UML et introduit les concepts de bases issus du domaine des systèmes TR^{2}E. Il favorise l'interopérabilité entre les différents outils utilisés pour la spécification, la conception, la vérification et la génération de code du système. Notamment, MARTE permet la modélisation des propriétés non fonctionnelles propres aux systèmes TR^{2}E. Enfin, MARTE introduit un modèle d'allocation des composants logiciels du système sur les composants matériels de la plate-forme d'exécution tout en respectant les contraintes temporelles et topologiques de l'application.

AADLv2. Récemment, une nouvelle version du langage AADL a été standardisée [SAE Aerospace, 2009b] et de nombreuses améliorations ont ainsi été apportées à savoir : l'introduction de nouveaux composants architecturaux, des précisions sur la sémantique des composants, une syntaxe graphique normalisée, un méta-modèle et diverses annexes (modélisation des données, comportementale...) permettant d'enrichir la modélisation et la qualité des outils associés.

Si le langage AADL 1.0 présenté prédécemment fait partie de la catégorie des ADLs, nous pouvons classifier le langage AADLv2 dans la catégorie des DSMLs considérant les caractéristiques propres à ces langages et les nombreuses améliorations citées ci-dessus et détaillées dans le chapitre 4 de ce manuscrit.

Discussion. Combinés à l'utilisation des techniques de modélisation, les DSMLs permettent d'élaborer des outils de qualité (représentation graphique, méta-modèle, édition de modèles...) qui facilitent particulièrement la représentation visuelle et la compréhension de systèmes complexes. De plus, l'utilisation des techniques liées à la transformation de modèle (présentées section 2.5) amène des perspectives intéressantes en termes d'expressivité, de traçabilité des exigences et d'automatisation.

A la différence des ADLs présentés précédemment, l'un des avantages d'UML/MARTE est la modélisation graphique de l'application TR²E mais celle-ci n'apporte que peu de bénéfices concernant la sémantique des concepts modélisés. Des travaux récents [Salazar, 2010; Vidal *et al.*, 2009] ont montré que le profil MARTE souffre des limites récurrentes du langage UML notamment pour le support d'une démarche de génération. La sémantique des concepts définis en UML et MARTE est trop riche et il est ainsi possible de modéliser un même concept de différentes manières, augmentant la complexité des patrons de génération et rendant la réutilisation des outils assez complexe. Enfin, plus généralement, les approches basées sur UML utilisent des syntaxes différentes pour décrire chacun des aspects d'une application, nécessitant la consolidation de toutes les représentations si le modèle de l'application est modifié ce qui s'avère couteux en terme de développement (ressources et temps).

Les récentes améliorations du langage AADLv2 et de nombreux travaux ont montré les multiples emplois possibles du langage AADL pour la modélisation, l'analyse et la conception de systèmes TR²E critiques. Outre le fait que le langage AADL a été développé précisément pour les systèmes TR²E et non pas adapté, l'avantage majeur du langage est la spécification architecturale et comportementale des composants logiciels et matériels de l'application au sein d'une unique représentation -*ie.e* un unique modèle.

De plus, des travaux ont montré l'intérêt de la communauté UML pour AADL et ont abouti à la définition d'un profil (intégré à MARTE) permettant le passage d'une spécification MARTE vers une description architecturale AADL [Faugere *et al.*, 2007] afin de profiter des nombreux outils basés sur le langage AADL.

Ces éléments nous amènent à penser que le langage AADL est un candidat plus approprié pour la modélisation, l'analyse et la conception de systèmes critiques selon un processus automatisé, ce qui fait l'objet de ce manuscrit. Intéressons-nous maintenant à l'utilisation faite des modèles à des fins d'analyse et de génération du code source du système.

2.2.2 Modèles d'analyses, vérification et validation

Dans notre introduction générale, nous avons vu la nécessité pour un système TR²E critique de satisfaire des exigences fortes en termes de sécurité et de sûreté de fonctionnement.

Différents modèles d'un système sont ainsi réalisés dans le but de couvrir et de permettre la vérification d'un aspect du système. Ainsi, de nombreux techniques et outils ont été développés afin d'établir des modèles d'analyse pour la vérfication et la validation du système vis-à-vis de ces spécifications.

Dans les sections suivantes, nous présentons les analyses les plus communes effectuées dans le contexte des systèmes critiques à savoir l'analyse d'ordonnançabilité et le *model checking*. Ces analyses permettent de conclure sur la faisabilité du système et de vérifier certaines propriétés comportementales. Leur objectif est d'augmenter *la confiance que l'on peut avoir dans ce système*.

Il est important de noter que la vérification du système s'effectue uniquement sur le système modélisé et non sur le système exécuté. Il s'agit d'un complément aux techniques de tests, permettant de vérifier certaines propriétés de manière exhaustive (comportement...) et d'effectuer des vérifications en amont du cycle de développement logiciel.

Ordonnançabilité

L'analyse d'ordonnancement d'un système vise à garantir que l'ensemble des tâches réalisant les fonctions du système finiront toujours leur exécution dans une échéance donnée. Pour ce faire, il est nécessaire de modéliser les tâches s'exécutant sur le processeur et de tenir compte des propriétés fonctionnelles et non fonctionnelles de celles-ci. Ainsi, une tâche doit être caractérisée par son comportement (périodique, sporadique ou apériodique), son pire temps d'exécution (WCET), sa période et/ou son échance (déterminée par son comportement) et une politique d'ordonnancement (RMS, EDF...) pour l'ensemble des tâches.

Divers travaux comme RMS ou EDF [Liu and Layland, 1973] proposent des algorithmes dont l'implantation a permis l'automatisation de l'analyse d'ordonnançabilité d'un système. Ces outils extraient les informations associées aux tâches décrites à l'aide d'un ADL ou d'un DSML.

CHEDDAR [Singhoff *et al.*, 2004] et MAST [Universidad de Cantabria, 2010] sont des outils d'analyse d'ordonnancement dédiés aux systèmes TR^{2}E. Ils permettent aussi de simuler cet ordonnancement.

Model checking

Les techniques de *model checking* consistent à modéliser et à analyser le comportement d'un système afin de valider une propriété donnée. Les propriétés standards généralement exprimées sont l'accessibilité, la sûreté, la vivacité, l'équité et l'absence de blocage. Il est aussi possible de vérifier des propriétés plus spécifiques sous réserve que celles-ci soient exprimables dans une logique compatible avec les algorithmes et les outils de vérification.

Les formalismes les plus utilisés pour représenter et analyser les systèmes TR^{2}E critiques sont les réseaux de Petri colorés (analyse des flots de messages dans le système) ; les réseaux de Petri temporisés [Gorrieri and Siliprandi, 1994] et les automates temporisés [Alur and Dill, 1994] prenant en compte les caractéristiques temporelles de l'application (ordonnancement, dimenssionnement de ressources).

Les outils CPN-AMI [LIP6 - MoVe, 2012], TINA [LAAS, 2012] et UPPAAL [Uppsala University - Aalborg University, 2012] réalisent l'analyse du comportement du système TR^{2}E modélisé respectivement en réseaux de Petri colorés, temporisés et en automates temporisés.

Discussion

Dans le cas d'ensembles de tâches périodiques, les tests de faisabilité sont suffisants pour analyser et valider l'ordonnancement du système. Cependant, ceux-ci s'avèrent limités dans le cadre de la vérification de système TR²E critiques mettant en jeu des mécanismes de synchronisation et de communication complexes (ressources partagées et protégées...).

Les méthodes formelles et les techniques de *model checking* complètent ces analyses pour garantir la faisabilité du système notamment pour la vérification de propriétés de sûreté spécifiques (interblocage, famine...).

Les techniques et les outils que nous venons de présenter permettent de vérifier et de valider l'ordonnancement et le comportement attendu d'un système TR²E critique modélisé. Dans notre contexte, nous n'entendons pas développer des outils d'analyses basés sur ces techniques, mais plutôt, dans le cas, possible réutiliser les outils existants et utilisant notamment le formalisme AADL comme (CHEDDAR pour l'ordonnancement et TINA ou CPN-AMI pour le *model checking* à partir de réseaux de Petri [Renault, 2009]).

Après ces étapes de vérification et de validation du système, nous nous intéressons maintenant à l'implantation du système et plus précisément aux étapes de déploiement et de configuration des systèmes critiques qui, combinés à une démarche de génération de code, autorisent la production automatique du code source de l'application.

2.2.3 Génération de code

La production automatique du code source de l'application est une étape importante de l'automatisation du processus de conception des systèmes TR²E. Outre les gains de productivité, elle augmente la fiabilité (élimination des erreurs introduites manuellement au sein du code, production de code déterministe à partir de patron de génération, etc.) et le niveau de confiance que l'on peut avoir dans le système. De nombreux outils notamment commerciaux tel SCADE [Esterel Technologies, 2012] utilisent ce concept.

Description

Un formalisme source est utilisé pour abstraire les constructions complexes et les difficultés d'implantation dans un langage de programmation (élaboration d'un modèle de code). Le processus de génération transforme alors ces composants abstraits en construction du langage cible (Ada, C, etc). En fonction de l'expressivité du formalisme source et de sa sémantique associée, il est alors possible de produire un squelette (exemple de l'utilisation d'UML) à compléter ou l'implantation complète (exemple de l'utilisation de LUSTRE) de l'application. Les aspects complexes (manipulation de pointeurs, gestion des types de données, etc.) sont ainsi pris en charge par le générateur de code et masqués à l'utilisateur.

Discussion

Si la génération de code possède de nombreuses caractéristiques renforçant l'implantation correcte d'un système (limitation des bugs...) d'un système, elle introduit néanmoins une nouvelle complexité.

La sémantique du formalisme d'entrée doit permettre de décrire efficacement les exigences du système sous peine de nécessiter l'intégration manuelle de code compromettant

la sûreté de l'application. La transposition des problèmatiques d'implantation au modèle requiert une validation du modèle avant de procéder à la génération. De plus, ce modèle doit être analysable afin de vérifier les caractéristiques du système.

La difficulté de certifier l'ensemble des composants logiciels de l'application nécessite l'élaboration d'un processsus rigoureux (standards, documentation, traçabilité, méthodes de tests...) pour assurer un certain niveau de confiance dans le code généré. Enfin, le surplus de code généré à cause de stratégies de génération trop génériques voir inadaptées (nombreux accesseurs, dépendances du code vers des bibliothèques "lourdes"...) et l'intégration de code tiers (utilisateur) sont autant de difficultés à prendre en compte lors de l'élaboration du générateur de code.

2.2.4 Déploiement et configuration automatiques

Pour permettre une démarche de génération de code efficace, il est nécessaire d'automatiser les étapes de déploiement et de configuration des composants logiciels qui constituent l'application. Dans le cadre de ses travaux [Zalila, 2008] a adapté les étapes de déploiement et de configuration décrites par le standard D&C [OMG, 2006b] afin d'accentuer l'aspect statique qui caractérise les systèmes TR^{2}E critiques. Notamment, il donne les définitions suivantes :

Définition 2.2 (Déploiement) *Le déploiement d'une application répartie est la sélection des composants intergiciels requis pour réaliser une sémantique donnée. Le déploiement requiert aussi le placement d'entités qui envoient les messages à travers le réseau à partir des nœuds sources (souches) et d'autres entités qui reçoivent ces messages sur les nœuds de destination (squelettes).*

Définition 2.3 (Configuration) *La configuration d'une application répartie est l'opportunité de paramétrer individuellement chacun des composants intergiciels sélectionnés lors de la phase de déploiement et de les assembler par la suite avec les composants générés automatiquement ainsi que les composants fournis par l'utilisateur.*

Discussion

L'étape du déploiement est une tâche pouvant être source d'erreur. Il est donc nécessaire d'automatiser cette tâche notamment pour tirer bénéfice des descriptions architecturales réalisées lors de la phase de modélisation. Dans le contexte des systèmes critiques, les informations de déploiement doivent être connues à l'initialisation du système pour l'ouverture des canaux de communication. Dans le cadre des systèmes critiques, la phase de configuration est aussi importante. Celle-ci permet de gérer finement et d'allouer uniquement les ressources logicielles nécessaires à l'application et ainsi d'optimiser l'utilisation des ressources matérielles (souvent limitées).

Il existe de nombreux outils pour le déploiement et la configuration des composants logiciels d'un système TR^{2}E (CoSMIC [Gokhale *et al.*, 2003], FractalADL [Consortium, 2012], Ocarina [TELECOM ParisTech, 2012], etc). Dans la section 2.4, nous détaillons ceux visant précisément le contexte des systèmes TR^{2}E critiques.

Nous avons vu, dans cette section, les étapes importantes du processus de conception automatisé des systèmes TR^{2}E critiques. Dans les sections suivantes, nous nous intéressons aux spécificités liées à l'implantation des systèmes critiques et aux processus de conception et d'analyse existants.

2.3 Implantation des systèmes TR²E critiques

Nous pouvons trouver dans la littérature de nombreuses définitions du terme *système critique* [Rushby, 1994], mais intuitivement, nous pouvons qualifier un système TR²E de *critique* si une défaillance, qu'elle soit mécanique, électronique ou logicielle peut provoquer sa perte et engendrer des conséquences importantes voir dramatiques, dont l'impact peut se mesurer en pertes humaines, économiques ou environnementales.

Cette famille de systèmes est généralement présente dans des domaines tels que le spatial, l'aéronautique, le ferroviaire ou le médical nécessitant que le système respecte des exigences fortes en termes de sécurité et de sûreté de fonctionnement [Barnes, 2008].

De nos jours, il est difficile de garantir totalement la sécurité et le bon fonctionnement d'un système dans l'ensemble de ses conditions d'utilisation. Cependant, un certain nombre de méthodologies définissant techniques, règles de conception et outils a été mis en œuvre afin d'éliminer les risques de bugs, de détecter les incohérences et de prévenir des défaillances. Dans la suite, nous présentons certaines de ces méthodologies ou approches utilisées pour la conception et la réalisation des systèmes critiques.

2.3.1 Intergiciel critique

L'intergiciel est un élément cœur dans la conception des systèmes répartis. C'est une couche logicielle *intermédiaire* dont le rôle est d'assurer et de faciliter la communication entre les différents nœuds - *i.e.* applications distribuées - s'exécutant sur une architecture matérielle.

L'intergiciel s'intercale entre le code applicatif et la plate-forme d'exécution. Même si dans la plupart des cas, la plate-forme d'exécution est constituée de l'architecture matérielle et du système d'exploitation, certaines fonctionnalités requises d'un système d'exploitation (concurrence, interaction...) peuvent être implantées par l'intergiciel [Armstrong, 1996]. Par conséquent, celui-ci agit comme une interface en offrant des primitives de plus *haut-niveau*, évitant l'invocation directe des primitives du systèmes d'exploitation ou du matériel améliorant ainsi la portabilité.

De nombreux mécanismes et standards ont été définis afin de normaliser la conception d'applications dans le domaine distribué : de la simple invocation distante de procédures RPC [SUN, 1988], à l'utilisation d'objets répartis CORBA [OMG, 2004], la distribution des données DDS [OMG, 2007a] et même pour l'adaptation de ces mécanismes pour les systèmes temps-réel [OMG, 2005] et embarqués [OMG, 2006a].

Outre la minimisation des coûts de développement (temps, finances), l'intergiciel a pour objectif le renforcement de la "séparation des préoccupations" en satisfaisant l'ensemble des possibilités des exigences non fonctionnelles des systèmes répartis. Ainsi, il devient possible d'adapter la configuration de l'intergiciel en fonction des besoins en ressources intergicielles de l'application distribuée.

Ceci a pose la question fondamentale de savoir si un intergiciel doit satisfaire un caractère "générique" - *i.e.* de répondre à l'ensemble des besoins -, "configurable" - *i.e.* de répondre à un besoin donné - ou "schizophrène" - *i.e.* de répondre simultanément et uniquement à plusieurs besoins.

Dans la suite de cette section, nous présentons brièvement, à travers un exemple de leur implantation, deux stratégies de conception d'intergiciels : "configurable" et "schizophrène"[Quinot, 2003]. Puis, nous discutons de leur utilisation dans le contexte de développement des systèmes critiques.

Intergiciels configurables

Définition 2.4 (Intergiciel configurable [Quinot, 2003]) *Un intergiciel est dit configurable lorsque les composants qu'il intègre peuvent être choisis et leur comportement adapté en fonction des besoins de l'application et des fonctionnalités offertes par l'environnement.*

Dans le contexte des systèmes TR^{2}E critiques, les intergiciels configurables disposent de caractéristiques intéressantes. En effet, la sélection des composants uniquement requis par l'application permet d'optimiser sa taille mémoire et leur personnalisation renforce le respect des exigences.

TAO (The ACE ORB). est un intergiciel développé par l'université de Vanderbilt [Schmidt *et al.*, 1998]. Il s'appuie sur le standard CORBA [OMG, 2004] et a été conçu afin d'assurer la qualité de service de bout-en-bout des systèmes temps-réel répartis en autorisant la configuration de nombreux éléments (caractéristiques des tâches, politique d'ordonnancement, ressources requises pour l'exécution d'une opération, etc). Son architecture extensible autorise l'utilisateur à choisir un contexte (performance, faisabilité...) pour effectuer diverses opérations et assurer le fonctionnement de l'application. Le développement de cet intergiciel a contribué à la définition du standard RT-CORBA [OMG, 2005].

L'utilisation du canevas ACE, reposant sur des patrons de conception classiques des systèmes temps-réel [Schmidt and Cleeland, 2001] permet la configuration des mécanismes de communication (connexions, entités réagissant à la réception d'évènements...) et assure sa portabilité. Enfin, l'utilisation des mécanismes de communication spécifiés au sein du standard CORBA assure par construction l'interopérabilité des applications avec TAO.

Les travaux effectués autour de l'intergiciel TAO [Schmidt *et al.*, 1998; Schmidt and Cleeland, 2001] mettent en évidence l'intérêt de la construction d'un intergiciel dynamiquement configurable par intégration et personnalisation de composants pré-existants réduisant ainsi la complexité et augmentant la maintenabilité des fonctionnalités de base. L'utilisation d'une technologie de répartition reposant sur les patrons de conception simplifie considérablement le développement des applications réparties notamment en terme d'hétérogénéité.

Discussion. Des études montrent des incompatibilités d'utilisation de RT-CORBA pour les systèmes TR^{2}E critiques. En particulier, [Hugues, 2005] révèle la possibilité de modifier dynamiquement la priorité des tâches par l'utilisateur compromettant ainsi l'analysabilité statique d'ordonnancement. De plus, l'architecture orientée objet de TAO limite l'analyse du déterminisme de l'application.

Enfin, la configuration et le déploiement d'une application utilisant TAO s'effectuent de manière dynamique à l'initialisation du système. Si l'utilisation du standard CORBA assure l'interopérabilité, il limite les possibilités d'optimisation de certaines exigences non fonctionnelles comme, par exemple, lors de l'utilisation de mécanismes de communication spécifiques à une application ; ceux-ci ne nécessitant pas tous les mécanismes assurant la compatibilité entre les nœuds (cas des applications distribuées non hétérogènes).

Intergiciels schizophrènes

Définition 2.5 (Intergiciel schizophrène [Quinot, 2003]) *Un intergiciel est dit schizophrène lorsqu'il permet la cohabitation de plusieurs paradigmes de répartition au sein d'une même*

instance et met en place un mécanisme efficace d'interaction entre ces paradigmes (personnalités).

PolyORB. [Quinot, 2003] est un intergiciel développé par AdaCore qui implante l'architecture d'intergiciel définie par [Pautet, 2001]. Cette nouvelle architecture découpe les fonctionnalités de base d'un intergiciel en "services canoniques" assurant l'acheminement correcte d'une requête de l'expéditeur vers le destinataire. Ces services (adressage, liaison, interaction, typage, représentation, protocole, transport, activation et exécution) sont implantés dans un noyau cœur appelé *couche neutre*. Cette implantation est indépendante de tout standard de répartition.

L'architecture de PolyORB est définie en trois couches. Les *personnalités applicatives* correspondent à l'implantation des aspects applicatifs d'un standard de répartition donné (représentation des données, gestions des objets répartis...). Les aspects liés à la communication (protocoles, transport...) sont implantés au sein de *personnalités protocolaires*. *Personnalités applicatives* et *personnalités protocolaires* utilisent et spécialisent les services offerts par la *couche neutre*. Cette architecture "schizophréne" permet ainsi l'utilisation de personnalités différentes et leur coopération au sein d'une même instance d'intergiciel.

Discussion. La configuration de PolyORB s'effectue à l'initialisation du système et s'appuie sur des mécanismes d'aiguillage dynamique (héritage) pour la sélection de l'instance requise pour chaque service. Ceci est principalement dû à l'absence d'un mécanisme pour la configuration statique de l'application à partir d'une analyse fine de ses propriétés.

Pour générer automatiquement et configurer statiquement PolyORB, des travaux ont permis l'utilisation du langage de description d'architecture AADL [Vergnaud, 2006]. Enfin, des travaux plus récents [Zalila *et al.*, 2008] ont raffiné cette architecture pour la définition de l'intergiciel PolyORB-HI répondant aux besoins des systèmes TR²E. Celui-ci vise à fournir une instance d'intergiciel *dédiée* à l'application - *i.e.* seuls les services et les composants intergiciels requis par l'application sont sélectionnés et déployés - avec une empreinte mémoire plus réduite que celle obtenue avec PolyORB.

Conclusion

Nous venons de voir différentes méthodes de conception d'intergiciel pour les systèmes TR²E. L'une de nos contributions vise à proposer une architecture d'intergiciel afin de prendre en compte les composants de l'application complète (applicatif + intergiciel) lors des différentes étapes de vérification et de validation du système modélisé.

Dans le contexte de nos travaux de recherche, nous souhaitons ré-utiliser un intergiciel dédié aux systèmes TR²E. L'intergiciel PolyORB-HI, détaillé dans le chapitre 6 de ce manuscrit, est un candidat approprié car celui-ci répond directement aux spécificités des systèmes critiques. De plus, il supporte les composants logiciels modélisés en AADL, formalisme de description de systèmes critiques que nous avons retenu pour notre approche.

2.3.2 Le profil architectural Ravenscar

Une approche pour l'implantation d'un système critique est la limitation des constructions logicielles pouvant compromettre le respect de certaines exigences. Ainsi, le profil architectural Ravenscar [Burns *et al.*, 2004; Kwon *et al.*, 2002] dédié au langage Ada et Java propose de

restreindre les constructions du langage de programmation utilisé pour l'implantation d'applications monolithiques. Celui-ci n'autorise que les constructions qui garantissent une analyse statique d'ordonnancement, l'absence d'interblocage et une borne temporelle d'inversion de priorités entre les tâches. Cette approche facilite le développement d'entités logicielles déterministes et *correctes-par-construction*. Le respect de l'ensemble de ces restrictions est assuré lors de la phase de compilation.

Description

Concernant le langage Ada, le profil Ravenscar impose que :

- les constructions telles que les *rendez-vous*, les entrées dans les tâches et les entrées multiples d'objets protégés sont interdites car elles rendent complexe l'analyse statique du système (augmentation des mécanismes d'interaction entre les tâches, etc) ;
- les tâches ne doivent jamais finir et toute création de tâche ne peut survenir que lors de la phase d'élaboration (phase d'initialisation de l'application) afin de garantir que l'ensemble de tâches à analyser reste invariant ;
- l'accès aux objets protégés doit être utilisé selon le protocole de plafonnement de priorité PCP (*Priority Ceiling Protol* [Sha *et al.*, 1990]) afin de garantir l'absence d'interblocage et borner l'inversion de priorité ;
- l'utilisation exclusive des tâches périodiques ou sporadiques afin de garantir une analysabilité selon l'analyse par taux d'utilisation du processeur (RMA) ou l'analyse de temps de réponse des tâches (RTA).

Discussion

Le profil Ravenscar vise initialement le développement d'applications monolithiques et les aspects liés à la concurrence. Cependant, dans le cadre de communications asynchrones et de l'utilisation d'un protocole temps-réel entre les nœuds de l'application, celui-ci peut-être étendu pour les systèmes TR^{2}E.

Des restrictions supplémentaires autres que celles sur les aspects de concurrence peuvent aussi être appliquées. L'interdiction de l'utilisation d'instructions d'allocation dynamique de mémoire permet de prévenir des erreurs d'incohérence des pointeurs et d'éliminer des facteurs d'indéterminisme temporel. Cette restriction permet de vérifier statiquement - *i.e.* dès l'étape de compilation du code source - les opérations mémoire, le respect de la taille de la pile, etc. L'interdiction des constructions telles que l'aiguillage dynamique réduit aussi un facteur d'indéterminisme temporel et facilite l'implantation d'outils d'analyse. Enfin, l'interdiction de l'utilisation des nombres à virgules flottantes permet de prévenir d'un certaines erreurs sur les opérations calculatoires et les dépassements mémoire.

Ces restrictions ne sont pas obligatoires mais elles sont régulièrement appliquées au sein de diverses approches et font partie des bonnes recommandations pour la conception et l'analyse des systèmes TR^{2}E critiques. Elles ont, par exemple, été ajoutées au standard Ada 2005 [Working Group, 2005, Annexe H] et sont vérifiées à la compilation (par le compilateur GNAT [Miranda and Schonberg, 2004]). Dans le cadre de nos travaux, nous avons choisi de respecter l'ensemble des restrictions que nous venons de présenter.

Intéressons-nous maintenant aux processus automatisés pour l'analyse et l'implantation de systèmes TR^{2}E critiques existants et s'appuyant sur une démarche IDM.

2.4 Processus d'analyse et d'implantation de systèmes critiques

Dans cette section, nous présentons trois méthodes intégrées pour l'analyse et l'analyse de systèmes critiques. Par méthodes intégrées, nous entendons des approches définissant une méthode, un processus d'ingénierie et fournissant un canevas pour l'analyse et l'implantation (génération, déploiement et configuration) de systèmes critiques.

Ces approches intègrent notamment un ou l'ensemble des éléments présentés dans les sections précédentes. Après leur présentation, nous discutons des avantages et des limites de ces approches. Certaines de ces limitations font l'objet des motivations de ces travaux de recherche et du nouveau processus de conception de systèmes critiques que nous proposons.

2.4.1 HRT-UML/RCM

Description

HRT-UML/RCM (*Hard Real-Time UML for the Ravenscar Computational Model* [Mazzini *et al.*, 2009]) est une méthode intégrée pour le développement d'applications critiques temps-réel dur. Cette méthode a été développée dans le cadre du projet ASSERT [The Assert Project, 2012] et vise à transposer les principes de la méthode HRT-HOOD vers le langage UML selon une approche MDA.

HRT-UML/RCM fournit un canevas pour la conception de systèmes permettant :

- à l'utilisateur de définir des modèles du système indépendants de la plate-forme d'exécution (PIMs) ;
- la production automatique (par transformation) de modèles spécifiques à une plate-forme d'exécution (PSMs) respectant le profil Ravenscar ;
- l'analyse de faisabilité (des contraintes temporelles) et la vérification d'exigences non fonctionnelles sur le système ;
- la production automatique du code source du système à partir des PSMs.

La figure 2.1 décrit le processus de conception proposé par la méthode HRT-UML/RCM.

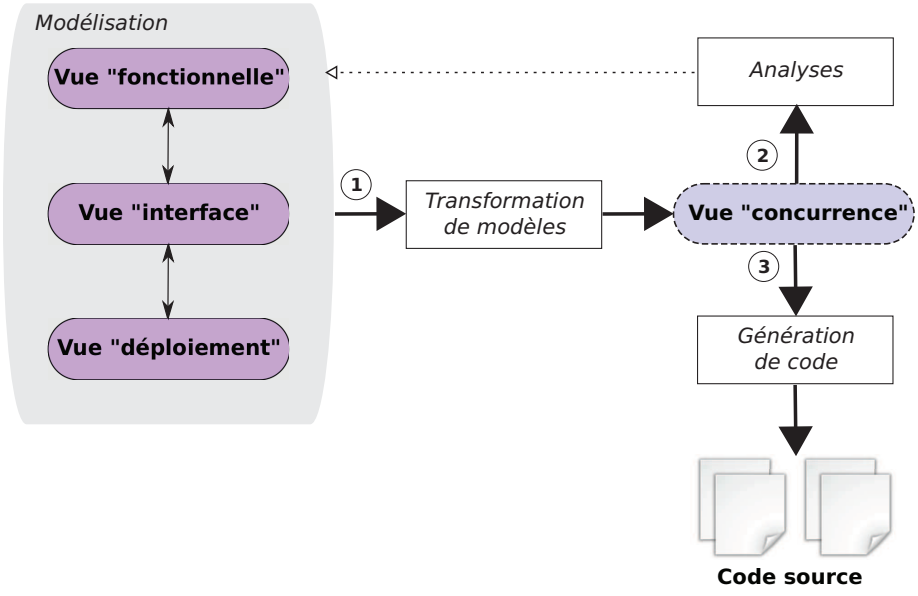

FIGURE 2.1 – Processus de conception de la méthode HRT-UML/RCM

Modélisation

La description architecturale suit les recommandations du standard IEEE 1471 [IEEE, 2000] renforcant la "séparation des préoccupations". Dans ce contexte, HRT-UML/RCM définit trois vues "fonctionnelle", "interface", "déploiement", spécifiées par l'utilisateur et une vue "concurrence", générée à partir des trois précédentes, pour décrire l'architecture logicielle du système.

Les "composants" sont spécifiés à l'aide de diagrammes de classe UML standard. Un méta-modèle définit l'ensemble des classes utilisables. La sémantique associée à une partie des éléments du méta-modèle respecte le modèle de concurrence du profil Ravenscar (RCM [Bordin and Vardanega, 2005]) assurant ainsi un assemblage de composants "correcte-par-construction" et garantissant l'analyse statique de l'application.

La vue "fonctionnelle" capture les informations structurelles et comportementales des différentes opérations réalisées par le système indépendamment des exigences non fonctionelles du domaine TR^{2}E.

La vue "interface" raffine les composants de la vue "fonctionnelle" en y attachant les informations liées à la concurrence, les interfaces (opérations) requises et fournies par les différents composants et leur interaction. Les composants sont instanciés et interconnectés à l'aide de point d'interaction (*PortCluster*). L'ensemble forme une instance de l'application TR^{2}E vérifiable.

La vue "déploiement" spécifie les composants physiques de l'architecture et exprime comment les composants logiciels s'exécutent sur ces derniers. La notion de "partition" y est introduite pour modéliser un nœud logique, encapsulant les instances de composants et assurant leur isolation spatiale et temporelle.

La vue "concurrence", générée à partir des trois vues précédentes (par transformation "prouvée" [Cancila *et al.*, 2010]), décrit l'architecture du système réalisée pour la plate-forme d'exécution spécifiée par la vue de "déploiement". Les composants de la vue "interface" sont ici traduits en des composants respectant la sémantique du RCM.

Analyses

Le PSM produit par la vue "concurrence" est le point d'entrée pour les différentes analyses. Des analyses temporelles et notamment d'ordonnançabilité sont effectuées à l'aide d'une extension (MAST+ [Bordin *et al.*, 2008; Panunzio and Vardanega, 2011]) de l'outil MAST [Universidad de Cantabria, 2010] autorisant notamment l'analyse de tâches sporadiques. La transformation du PSM vers un modèle MAST+ est assurée par le canevas de conception HRT-UML/RCM. Les résultats de l'analyse d'ordonnancement sont automatiquement remontés à l'utilisateur qui peut, le cas échéant, éditer ses modèles et relancer le processus. D'autres analyses sont possibles, utilisant notamment les méthodes formelles et des techniques de *model checking* pour la partie comportementale.

Génération de code

HRT-UML/RCM permet la génération d'une application TR^{2}E critique et sa plate-forme d'exécution dans les langages Ada 2005, Ada 2005/Ravenscar et C pour le standard OSEK/VDX RTOSs [Bordin and Vardanega, 2005]. La génération de code Ada 2005/Ravenscar semble être la plus aboutie, notamment par les choix de conception des auteurs de l'approche et le respect du profil Ravenscar. Elle s'effectue par transformation modèle-à-code implantée en

MOFscript. Les composants RCM modélisés sont utilisés pour paramétrer et instancier des patrons -*i.e.* des paquetages génériques Ada - implantant le comportement des composants.

Discussion

La méthode HRT-UML/RCM est basée sur l'approche MDA et intègre des outils de transformation de modèles. Le choix de contraindre la réalisation du méta-modèle à la sémantique de RCM renforce l'analysabilité statique du système et contraint la modélisation aux seules constructions autorisées par le profil Ravenscar. Enfin, les analyses temporelles effectuées sur le modèle de l'application tiennent compte d'une partie des ressources intergicielles souvent exclues dans les autres approches.

La principale limite de HRT-UML/RCM est l'utilisation du langage UML. En effet, la modélisation d'un système implique le développement de trois vues (fonctionnelle, interface et déploiement) nécessitant une consolidation des différentes représentations du système à chaque modification de l'un des modèles. De plus, si la notation des diagrammes de classes UML est très expressive pour visualiser interface, opérations et propriétés non-fonctionnelles des composants, celle-ci complique fortement la modélisation dans le cas d'un très grand nombre de composants (passage à l'échelle).

Une autre limitation vise les aspects comportementaux qui ne sont pas explicitement modélisés à l'aide d'un formalisme tels les *statecharts* UML. Ceux-ci sont implicitement décrits par la sémantique des composants RCM et enfouis au sein du méta-modèle par construction. L'analyse comportementale requiert alors une transformation (traduction du comportement des composants) plus complexe vers un formalisme d'un outil tiers.

Enfin, le processus de génération de code est basé sur l'instanciation et la configuration de paquetages génériques Ada implantant le comportement réel des composants. Si ce procédé renforce la fiabilité et augmente le niveau de confiance du système généré, ceci peut se faire au détriment de l'optimisation de l'empreinte mémoire.

2.4.2 OCARINA/POLYORB-HI

Description

OCARINA [TELECOM ParisTech, 2012] est un projet réalisé lors de différents travaux de recherche [Vergnaud, 2006; Zalila, 2008; Renault, 2009] de l'équipe S3 de TELECOM ParisTech, s'articulant autour de la conception d'intergiciel, de la production et de l'analyse automatisée de systèmes TR^{2}E critiques.

OCARINA s'appuie sur le langage de description d'architecture standardisé AADL, ses différentes annexes (modélisation des données, génération de code...) et sur le profil architectural Ravenscar. L'intergiciel POLYORB-HI [Zalila, 2008] a été défini et implanté dans différents langages cibles (Ada/Ravenscar, C, et Java/RTSJ) et sert de plate-forme d'exécution pour les composants architecturaux logiciels modélisés en AADL puis générés dans l'un des langages cibles.

La figure 2.2 décrit le processus de conception et d'analyse d'un système critique à partir de sa description architecturale réalisée en AADL. L'architecture du canevas de développement proposée par OCARINA reprend l'architecture classique d'un compilateur (par exemple GNAT).

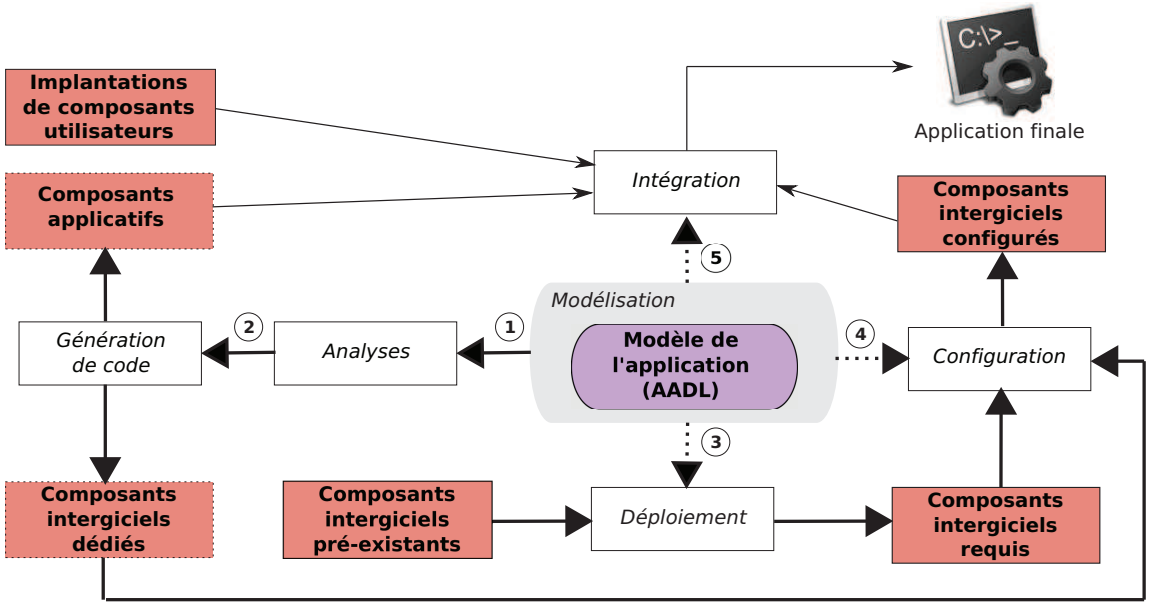

FIGURE 2.2 – Processus de conception de l'approche OCARINA

Modélisation

La modélisation d'un système TR^{2}E critique s'effectue uniquement à l'aide du langage AADL (AADL 1.0 ou AADLv2). L'ensemble des constructions architecturales AADL utilisées par OCARINA est restreint pour la génération de code aux seules constructions autorisées par le profil Ravenscar.

Analyses

Le point d'entrée des analyses par OCARINA est la représentation architecturale finale - *i.e.* complète et valide au sens "légal" AADL - du système. Différentes analyses sont alors possibles. REAL [Gilles, 2008] est un langage de contraintes qui permet de vérifier des propriétés non fonctionnelles sur une description AADL (dimensionnement, définition d'un plafond de priorité, etc). L'analyse d'ordonnancement du système est effectuée à l'aide de l'outil CHEDDAR et un générateur de code permet la transformation du modèle AADL vers les réseaux de Petri pour une analyse comportementale [Renault, 2009].

Génération de code

Une fois le modèle AADL analysé et le système critique validé, OCARINA permet la production automatique du code des nœuds de l'application distribués et des ressources intergicielles (POLYORB-HI) requises par ceux-ci. Les composants intergiciels sont ainsi sélectionnés, déployés, configurés et intégrés de manière automatique aux composants applicatifs pour former l'exécutable final de l'application. Différents générateurs de code sont fournis pour les langages Ada/Ravenscar, C, Java/RTSJ et C/ARINC 653.

Discussion

La principale limitation de l'approche OCARINA vient du fait que l'outil ne supporte que les descriptions architecturales AADL. OCARINA ne disposant pas d'un formalisme pour la spécification des aspects comportementaux (l'annexe comportementale AADL [SAE Aerospace, 2009a] n'étant pas à son niveau de maturité actuel), la spécification des aspects comportementaux s'effectue par l'implantation du comportement au sein de composants opaques (*subprogram*) et explique notamment l'intégration des composants de l'intergiciel POLYORB-HI uniquement lors de la phase de compilation. La non prise en compte des ressources intergicielles (dépendantes de la plate-forme d'exécution) lors des analyses de faisabilité est une limite récurrente des approches basées sur l'IDM. Celle-ci augmente la différence sémantique entre modèles d'analyse et l'implantation réelle du système et complexifie les techniques mises en place (traçabilité...) pour assurer la cohérence entre spécification, analyses et code généré.

Enfin, les transformations vers les modèles d'analyses (réseau de Petri...) et le processus de génération de code à partir de la description architecturale AADL sont implantés en Ada. Ceci complexifie à la fois le développement, la maintenance et l'évolution des générateurs de code comparé à une démarche de génération modèle-à-modèle ou modèle-à-code issue des techniques de transformation de modèles [Brun *et al.*, 2008]. Nous détaillons ces techniques dans les sections suivantes de ce chapitre.

2.4.3 MYCCM-HI

Description

MYCCM-HI [Borde, 2009; Borde *et al.*, 2009] est une méthode de conception dédiée aux systèmes TR^{2}E critiques et adaptatifs - *i.e* reconfigurables -, développée par Thales et TELECOM ParisTech dans le cadre du projet FLEX-EWARE [Flex-eWare Project, 2012]. Cette méthode vise la conception et l'analyse formelle des systèmes TR^{2}E reconfigurables et se base sur le standard Lightweight CCM [OMG, 2003a].

La figure 2.3 décrit le processus de conception de cette méthode. L'architecture générale du canevas de développement MYCCM-HI reprend celle des compilateurs classiques (GNAT, etc).

Modélisation

MYCCM-HI définit son propre langage de description d'architecture COAL (*Component-Oriented Architectecture Language*) pour modéliser le comportement adaptatif d'un système. Celui-ci étend une spécification OMG IDL avec des constructions issues des standards D&C et AADL 1.0. COAL permet de décrire conjointement l'architecture logicielle selon une approche à base de composants "génériques" et les exigences non fonctionnelles spécifiques aux systèmes TR^{2}E sous forme de propriétés.

Les différents comportements possibles du système sont représentés à l'aide de modes de fonctionnement, encapsulés dans des automates analysables.

Analyses

Le point d'entrée des analyses effectuées par le canevas de développement MYCCM-HI est la spécification du système dans le langage COAL. A partir de celle-ci, MYCCM-HI génére

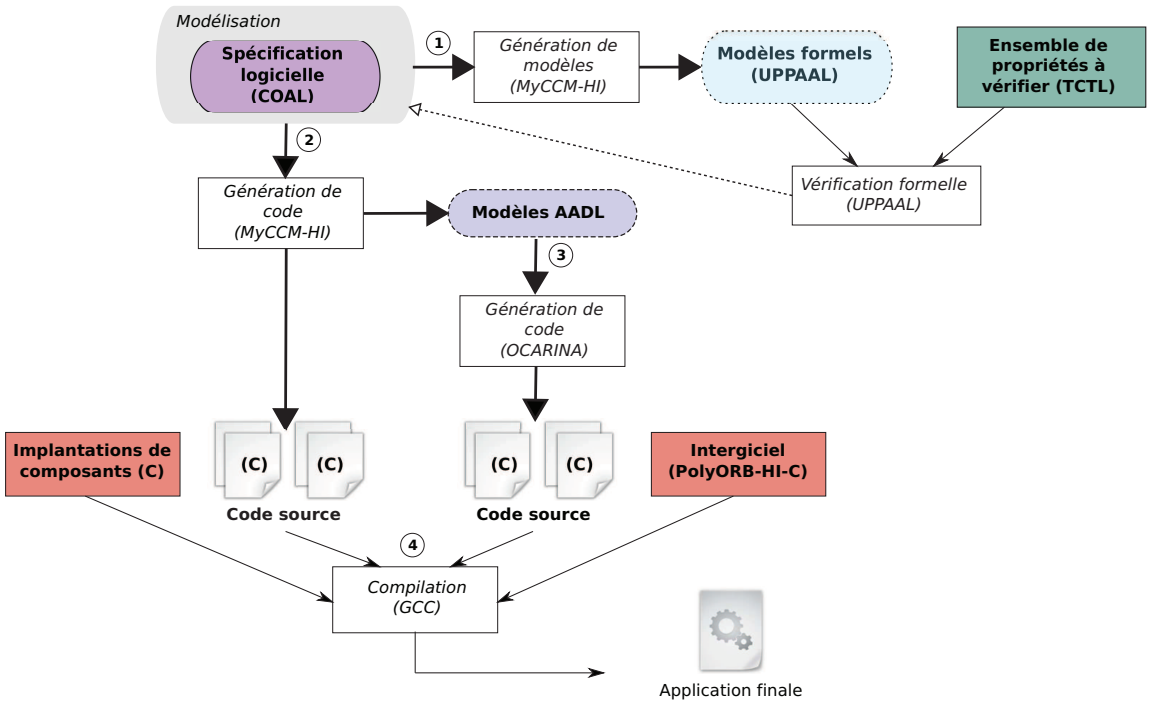

FIGURE 2.3 – Processus de conception de la méthode MyCCM-HI

des modèles formels analysés par l'outil UPPAAL [Uppsala University - Aalborg University, 2012].

Lorsque l'ensemble des propriétés de sûreté requises est validé, un modèle AADL représentant l'architecture logicielle et matérielle est généré à partir de la même spécification COAL. Les outils d'analyse (ordonnancement..) basés sur une représentation AADL et présentés dans le chapitre 4 sont ainsi exploités.

Génération de code

MyCCM-HI propose un processus de production automatique de systèmes critiques et adaptatifs en deux étapes. La première étape est réalisée par un générateur de code C implanté dans MyCCM-HI. Celui-ci génére les différents mécanismes d'adaptation "corrects-par-construction" respectant les contraintes de réalisation des systèmes critiques et requis par le système.

La deuxième partie concerne la génération des activités des tâches et des ressources de communication - *i.e.* des ressources intergicielles - requises par le système et supportant ainsi la partie fonctionnelle générée précédemment. Pour ce faire, MyCCM-HI intègre l'intergiciel PolyORB-HI et utilise le générateur de code C contraint Ravenscar de l'outil Ocarina [TELECOM ParisTech, 2012] pour générer les composants intergiciels.

Discussion

A notre connaissance, la méthode MyCCM-HI est l'une des premières solutions pour la conception et l'analyse de systèmes critiques avec modes. Néanmoins, celle-ci souffre de quelques limitations dues à différents choix de conception.

L'introduction du formalisme COAL et l'utilisation du langage AADL nécessitent l'utilisation d'une transformation dite exogène (voir définition 2.5.1) réalisée et enfouie au sein d'un générateur de code. Ceci complexifie la maintenance de l'outil et nécessite la définition de règles de transformation "correctes" afin de préserver la sémantique des composants modélisés. De plus, l'auteur [Borde, 2009] indique que les améliorations apportées au langage AADLv2 permettent maintenant de se passer de COAL.

Enfin, le processus de production de génération de code qui s'effectue en deux étapes (avec les générateurs MYCCM-HI et OCARINA) complexifie la génération mais aussi la traçabilité des composants requise dans le but de renforcer la cohérence entre modèles d'analyse et code généré.

2.4.4 Conclusion

Dans cette première partie de notre état de l'art, nous nous sommes intéressés aux processus de conception des systèmes TR^{2}E critiques. Certaines de ces solutions tirent bénéfice des techniques issues de l'IDM et notamment de la transformation de modèle pour l'automatisation des étapes d'analyses, de conception et d'implantation des composants logiciels, renforçant ainsi la sécurité et la sûreté de fonctionnement de l'application.

Les techniques de "modélisation" des différents aspects du système (notamment via la séparation des préoccupations) ont permis d'améliorer la productivité et la validation du système très tôt dans le cycle de développement logiciel (avant l'étape d'implantation). Cependant, celles-ci définissent et raisonnent sur des modèles à des niveaux d'abstractions différents introduisant de ce fait un biais sémantique entre modèles d'analyses, modèles de code et l'implantation finale du système. Cette différence de niveaux d'abstraction s'explique en partie, par certaines contraintes des techniques d'analyse (par exemple, les réseaux de Petri nécessitent la définition d'un modèle avec une forte abstraction afin de limiter l'explosion combinatoire) et d'autres résultent de l'inadaptation des outils ou du processus de conception développés.

Ainsi, nous avons identifié trois limites principales :

1. La modélisation implicite des aspects comportementaux et leur utilisation unique lors des étapes d'analyse. Ceci est dû à l'inadaptation du langage de modélisation nécessitant l'intégration ou la traduction vers un formalisme tiers pour expliciter le comportement. L'hétérogénéité des formalismes utilisés complexifie l'interopérabilité.
2. La prise en compte des composants intergiciels uniquement lors de la construction de l'exécutable final. Cette limite découle de la première et de l'incapacité du langage de modélisation à permettre une modélisation simple et efficace des ressources architecturales et comportementales de l'intergiciel. L'intergiciel n'intervient donc pas lors des analyses effectuées sur le modèle du système.
3. Un processus de génération de code inadapté. Dans la plupart des cas, le processus de génération consiste à sélectionner et à configurer un certain nombre de composants logiciels pré-existants. Ceci complexifie les mécanismes mis en œuvre pour assurer la cohérence.

Notre objectif principal vise à renforcer la fiabilité du processus de de réalisation des systèmes critiques et à rendre le code généré du système plus sûr. Pour ce faire, il est nécessaire de renforcer la cohérence entre modèles d'analyse, modèles de code et l'implantation finale du système.

A partir de ces trois limitations, nous proposons un nouveau processus de conception automatisé et détaillé dans le chapitre 3 de ce manuscrit. Ce processus repose sur la définition

d'une chaîne de transformation de modèles assurant le raffinement incrémental d'une description architecturale d'un système modélisé dans le langage AADL. Afin de mieux comprendre le rôle et les différentes étapes de nos transformations, il est nécessaire de s'intéresser aux différents techniques et outils de transformation de modèles.

2.5 Transformation de modèles : enjeux et solutions

L'état de l'art sur la conception et l'analyse des systèmes critiques nous a permis d'identifier deux limitations à savoir : 1) l'absence d'analyse des ressources de l'intergiciel et 2) la différence sémantique entre modèles d'analyse et code généré (limitant la traçabilité et le niveau de confiance du système généré).

L'étude des différents processus de conception de systèmes critiques (voir section 2.4) a montré le rôle central du modèle. Celui-ci se place au cœur du développement logiciel et représente des vues du système à des niveaux d'abstractions variés. Il capture des informations à différentes étapes du cycle de vie (spécification, conception...) et permet leur restitution lors de différentes activités de développement (analyses, génération de code, simulation, etc). La transformation de modèles assure l'évolution de ces représentations du système au sein des différentes étapes du cycle de vie logiciel.

Pour palier aux limites 1) et 2), nous proposons un nouveau processus d'analyse et d'implantation intégrant une étape de "raffinement" incrémental des modèles architecturaux et comportementaux (détaillée dans le chapitre 3) par transformation de modèles. Ce raffinement vise à intégrer les composants intergiciels, à expliciter le comportement des composants logiciels et à réduire le niveau d'abstraction du système critique modélisé tout en préservant la sémantique initiale des composants. Ainsi, nous renforçons la cohérence entre modèles d'analyse et code généré.

Dans cette section, nous présentons des définitions générales autour de la notion de transformation de modèles et nous explorons différents techniques et outils de transformation de modèles disponibles dans le monde de l'IDM.

2.5.1 Principe et concepts généraux

Dans la démarche de l'IDM, la transformation de modèles vise la génération et la manipulation de modèles. Elle trouve son intérêt dans de nombreuses activités du développement logiciel telles que le raffinement, la migration d'un langage, le changement d'espace technologique, la génération de code, etc.

L'OMG définit la transformation de modèles dans le contexte de l'approche MDA comme *"le processus de conversion d'un modèle en un autre modèle appartenant au même système"* [Kleppe *et al.*, 2003]. Mens et al. [Mens and Van Gorp, 2006] définissent la transformation de modèles comme *"la génération d'un ou plusieurs modèles cibles à partir d'un ou plusieurs modèles sources, selon une description de transformation"*. Nous retiendrons cette dernière dans le cadre de ces travaux de recherche.

Définition 2.6 (Transformation de modèles) *Génération d'un ou plusieurs modèles à partir d'un ou plusieurs modèles sources selon une description de transformation.*

Principe de la transformation de modèles

La figure 2.4 décrit le principe d'une transformation de modèles. Celle-ci assure la traduction d'un **modèle source** dans une représentation donnée vers un **modèle cible** dans la même représentation, mettant ainsi à jour les informations du modèle, ou dans une autre représentation, créant le modèle cible à partir d'informations extraites du modèle source.

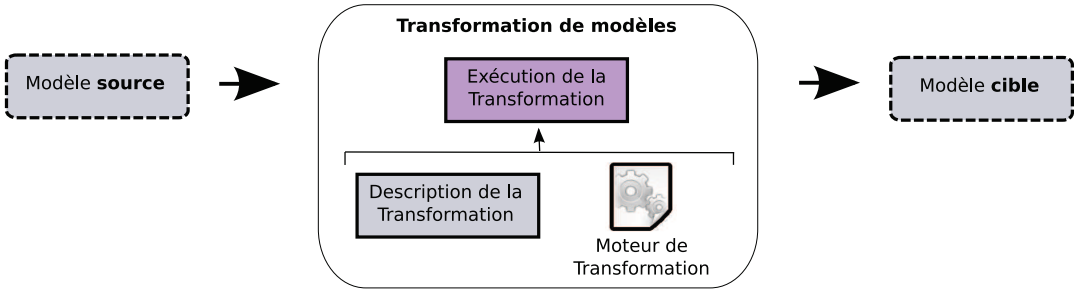

FIGURE 2.4 – Principe de la transformation de modèles

La **description de transformation** exprime comment un ou plusieurs modèles sources sont transformés en un ou plusieurs modèles à l'aide d'un langage de transformation de modèles. Elle définit l'ensemble des **règles de transformation** qui décrivent comment un ou des éléments (ou fragment) du modèle source sont transformés en un ou des éléments du modèle cible. Une règle de transformation contient un patron source et un patron cible. Pour chaque occurence du patron source du modèle source, un patron cible est créé dans le modèle cible.

Le **moteur de transformation** de modèles exécute ou interprète la description de transformation et applique celle-ci pour produire le modèle cible à partir du modèle source. L'automatisation d'une transformation "correctement définie" permet la réutilisation des informations du système et de renforcer la cohérence entre les différents modèles créés, raffinés ou maintenus.

Méta-niveaux d'une transformation de modèles

La figure 2.5 illustre les méta-niveaux de l'architecture d'une transformation de modèles et met en relation les concepts présentés précédemment et ceux issus de la méta-modélisation présentés en introduction (voir chapitre 1).

En particulier, nous observons qu'une description de transformation peut être représentée par un **modèle de transformation** conforme à un méta-modèle. Ainsi, il est possible d'utiliser une transformation de modèles comme modèle source et/ou modèle cible d'une autre transformation. Cette caractéristique offre des perspectives très intéressantes en terme d'évolution et de maintenance. Elle autorise la modification ou l'ajout d'informations (ou d'annotations) à la description de transformation (et par conséquent aux règles qu'elle définit) renforçant, par exemple, les techniques de traçabilité existantes.

Caractérisation des transformations de modèles

Les modèles sources et cibles peuvent être conformes à des méta-modèles identiques ou différents, appartenir à un même niveau d'abstraction (raffinement) ou à des niveaux d'abs-

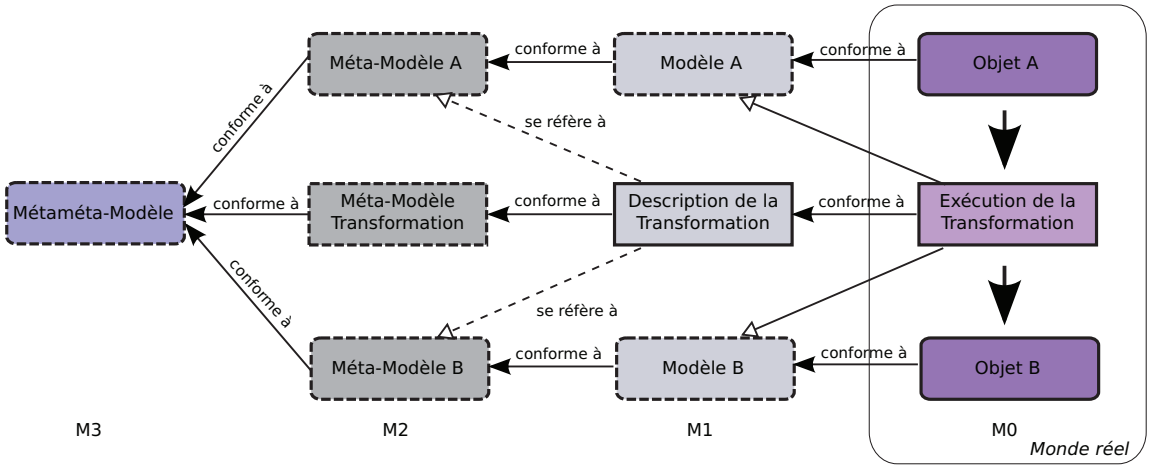

FIGURE 2.5 – Méta-niveaux de l'architecture de la transformation de modèles

traction différents (transformation de PIM vers PSM dans le MDA par exemple). Dans la littérature [Mens and Van Gorp, 2006], on parle alors de transformations *endogènes*, de transformations *exogènes*, de transformations *horizontales* ou de transformations *verticales*. Ainsi, nous pouvons distinguer quatre types de transformation de modèles, 1. et 2. par rapport à la différenciation des méta-modèles et 3. et 4. par rapport à la différenciation du niveau d'abstraction des modèles sources et cibles.

1. Une transformation est dite *endogène* si les modèles sources et cibles sont définis dans le même méta-modèle. Une optimisation, un refactoring ou une simplification sont des exemples de transformations endogènes.
2. Une transformation est dite *exogène* si les modèles sources et cibles sont définis dans des méta-modèles différents. Une synthèse (conversion d'un modèle source vers un niveau d'abstraction moins élevé), une rétro-ingénierie, une translation et une migration (programme Ada vers C) sont des transformations exogènes.
3. Une transformation est dite *horizontale* si les modèles sources et cibles appartiennent au même niveau d'abstraction. Le raffinement ou la translation est un exemple de transformation horizontale.
4. Une transformation est dite *verticale* si les modèles sources et cibles appartienent à des niveaux d'abstraction différents. Une synthèse, une rétro-ingénierie et la génération de code sont des exemples de transformation verticale.

Remarque. Nous pouvons aussi distinguer : la transformation *in-place* qui définit une transformation d'un modèle vers lui-même et assume que la source et la cible sont identiques excepté pour les fragments ciblés par la description de transformation ; et la transformation **higher-order** (HOT) qui définit une transformation de modèles ayant une transformation de modèles comme modèle source ou modèle cible.

Pour finir, les modèles sources et cibles peuvent également appartenir à des espaces technologiques. Un espace technologique est un contexte opérationnel de travail comportant un ensemble de concepts, un corps de connaissances, des outils, des compétences, etc. Il définit l'ensemble des technologies utilisées pour la représentation d'un modèle, les structures

de données, les parties-frontales, les outils nécessaires pour la manipulation des données et les formats de stockage des fichiers. Des exemples connus d'espace technologique pour la transformation de modèles sont EMF [Steinberg *et al.*, 2009], KERMETA [Fleurey, 2006], XML [Bray *et al.*, 2008] et le MDA [Blanc, 2005].

La section suivante présente brièvement les problématiques adressées par la transformation de modèles.

2.5.2 Problématiques adressées

Dans la littérature, nous pouvons trouver plusieurs outils traitant différentes problèmatiques par l'utilisation de la transformation de modèles. Dans cette section, nous présentons un aperçu des problématiques majeures adressées par celle-ci.

Changement du niveau d'abstraction

Selon la définition 1.5 (présentée dans l'introduction), le niveau d'abstraction est une mesure de la quantité d'informations - *i.e.* du niveau de détail - définie pour la description des fragments du modèle. La transformation de modèles permet soit (1) de réduire cette quantité d'informations (augmentation de l'abstraction du modèle), soit d'introduire de nouveaux détails et par conséquent de (2.1) réduire le niveau d'abstraction ou (2.2) de le laisser inchanger. Dans tous les cas (1, 2.1 et 2.2), cette propriété est indépendante des changements du méta-modèle et les méta-modèles source et cible peuvent être les mêmes ou différents.

Une transformation horizontale change l'espace technologique du modèle mais ne change pas le niveau d'abstraction initial du modèle (exemple de la translation). Une transformation verticale change le niveau d'abstraction du modèle. Celui-ci peut être augmenté par une synthèse réduisant le niveau de détail ou diminué par un raffinement ajoutant des détails au modèle source. Un exemple de transformation verticale de *raffinement* est la transformation d'un PIM en PSM [Blanc, 2005].

Changement de méta-modèle

Les transformations exogènes permettent la transformation des concepts d'un méta-modèle A vers des concepts d'un méta-modèle B. Ce type de transformation permet d'exhiber un aspect particulier du système modélisé et de le rendre plus compréhensible que ce soit par l'humain ou à l'aide d'un outil d'analyse spécifique (ex : représentation d'un système en réseau de Petri pour la vérification de propriétés de sûreté).

Changement d'espace technologique

Un espace technologique permet d'établir une représentation du modèle qui peut être stocké soit dans un format de fichier spécifique soit sous la forme de structures de données en mémoire (ex : XML [Bray *et al.*, 2008] ou XMI [OMG, 2007b]). Il fournit alors un ensemble de mécanismes permettant la manipulation de ces données. Ainsi, il est possible que l'espace technologique limite les moteurs de transformation pour diverses raisons telles que la scalabilité, le type de représentation (graphique vs textuelle), la traçabilité, la mise-à-jour incrémentale ou encore la stratégie d'exécution.

Aussi, le changement d'un espace technologique à un autre sous réserve d'une transformation "correctement définie" permet de contourner certaines limites.

Transformation modèle-à-code

La transformation *modèle-à-code* est utilisée pour générer du code dans un langage de programmation (Ada, Java, XML, etc.) à partir d'un modèle source. C'est un cas particulier de transformation *modèle-à-modèle* qui autorise un *mapping* arbitraire d'un élément du modèle source à une syntaxe particulière ou à un artefact du langage de programmation ciblé. Deux approches de transformation sont distinguées. L'approche basée sur le parcours de modèles (visiteurs) permet de traverser la représentation interne du modèle et de générer un code sous forme de texte dans un flux de sortie. L'approche basée sur l'utilisation de *templates* définit des fragments de code cible contenant des *méta-codes* qui assurent l'accès aux informations du modèle source, la sélection de morceaux de code et la réalisation d'expansions de manière itérative.

Cette transformation particulière est une alternative au processus de génération de code basé traditionnellement sur la théorie de la compilation.

Préservation des propriétés du système

Description de transformation avec préservation de la sémantique. Dans le cas des transformations endogènes, il est possible de définir une description de transformation qui préserve la sémantique des fragments du modèle. Ainsi, le modèle cible contient les mêmes informations provenant du modèle source mais celles-ci ne sont plus exprimées ou représentées sous la même forme. Elles sont soit traduites dans un espace technologique différent, soit avec une syntaxe abstraite différente.

Généralement, une description de transformation avec préservation de la sémantique améliore la description des éléments du modèle en affinant la spécification de ces attributs (par exemple, rendre explicite des informations implicites). Cependant, il peut s'avérer difficile d'exprimer un *mapping* qui préserve complètement la sémantique de l'élément ou d'un groupe d'éléments. Ainsi, il est possible de décrire une description de transformation définissant une *approximation* des propriétés du système. Celle-ci préserve alors uniquement les propriétés essentielles du modèle.

Un exemple de transformation avec préservation de la sémantique est l'optimisation du modèle à des fins de simulation ou d'amélioration de performances.

Description de transformation avec préservation du comportement. Une transformation préserve le comportement si les aspects comportementaux (i.e. les contraintes) spécifiés dans le modèle source sont également spécifiés de manière implicite ou explicite dans le modèle cible.

Remarque. Une description de transformation peut préserver le comportement (resp.la sémantique) et/ou la sémantique (resp. le comportement) d'un modèle (ex : la transformation de *modèle-à-code*).

Dans la section suivante, nous présentons brièvement une sélection des techniques et des outils de transformation de modèles des milieux universitaires et industriels.

2.6 Approches et outils de transformation de modèles

La transformation de modèles est l'une des clés du succès de l'IDM. Nous pouvons trouver aujourd'hui de nombreux techniques et outils de transformation de modèles dans les milieux universitaires et industriels. Dans la littérature, ces techniques et outils ont été classés selon différents critères [Czarnecki and Helsen, 2003; Czarnecki and Helsen, 2006; Mens and Van Gorp, 2006; Muller, 2006; Srivastava *et al.*, 2006; Jézéquel, 2008; Biehl, 2010]. Dans le cadre de nos travaux de recherche, nous nous contenterons de brosser un rapide aperçu des approches de transformation de modèles et des outils les plus pertinents pour notre solution. Nous mettons l'accent sur QVT et en particulier l'outil ATL avec lequel nous avons réalisé l'ensemble de nos transformations. Le choix d'ATL s'explique en grande partie par la maturité du langage (syntaxe textuelle précise...) et de ses outils associés et par les contraintes de réalisation (manipulation de méta-modèle Ecore, dévelopement de plug-ins ECLIPSE/RCP) que nous devons satisfaire pour l'implantation d'un prototype évolutif, maintenable et son intégration au canevas de développement OSATE2.

2.6.1 Approches de transformation de modèles

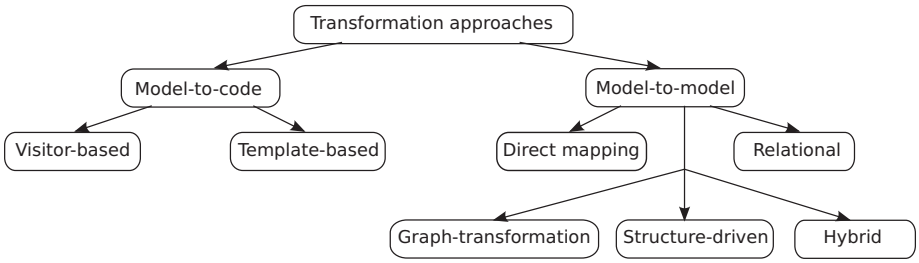

FIGURE 2.6 – Principales approches de transformation de modèles

La figure 2.6 présente les principales approches de transformation de modèles selon les critères de classification de Krzysztof Czarnecki et Simon Helsen [Czarnecki and Helsen, 2003; Czarnecki and Helsen, 2006]. Nous retrouvons les deux catégories présentées dans les sections précédentes : *modèle-à-code* (présenté en section 2.5.2) et *modèle-à-modèle*. Nous nous concentrons ici sur les transformations *modèle-à-modèle*. On distingue cinq principales approches.

1. L'*approche basée sur la manipulation directe de modèles* est généralement implantée sous forme de framework orienté objet (par exemple JMI [Dirckze, 2002]), fournit une représentation interne des modèles et une API pour la manipulation de celle-ci. Les règles de transformation sont définies à l'aide d'un langage de programmation (par exemple Java) par l'utilisateur.
2. L'*approche relationnelle* vise à spécifier, sous forme de contrainte, une relation entre éléments des modèles source et cible. La relation est déclarative et la spécification n'est pas exécutable.
3. L'*approche basée sur les transformations de graphes* repose sur la théorie des graphes. Elle vise la représentation graphique des modèles sous forme de graphes étiquetés et contraints par des règles de cohérence. Les techniques de réécriture de graphes et

de transformation de graphes sont ensuite appliquer. Une description de transformation est un ensemble de règles de réécriture et les règles de transformation peuvent être exprimées de manière déclarative comme dans les approches relationnelles. Nous citons à titre illustratif pour cette catégorie les outils ATOM [Lara and Vangheluwe, 2002], UMLX [Eclipse,], VIATRA [Eclipse Community,] et GREAT [Vanderbilt University, 2012].

4. L'*approche dirigée par la structure* réalise une transformation en deux étapes. La première étape crée la hiérarchie de la structure de la cible et la deuxième étape définit les valeurs des attributs et des références dans la cible. Les règles de transformation sont spécifiées par l'utilisateur. Les stratégies d'ordonnancement et d'application des règles sont définies par l'approche.
5. L'*approche hybride* est une combinaison des approches précédentes. Le langage de transformation de règles combine les approches déclarative et impérative. La spécification et l'implantation d'une transformation sont distinctes et deux types de règles sont définis. Les règles de correspondance déclarent les relations entre éléments du modèle source et du modèle cible. Les règles opérationnelles définissent les règles exécutables (création, modification, suppression, etc.). KERMETA [Fleurey, 2006] et ATL [ATL, 2012] sont des exemples d'approche hybride que nous détaillons dans les sections suivantes.

2.6.2 Outils de transformation de modèles

Intéressons-nous maintenant, brièvement, aux différentes catégories d'outils dédiés à la transformation de modèles. Nous pouvons distinguer cinq catégories dominantes.

1. Les *outils associés aux langages de programmation* sont disponibles sous forme d'APIs dans un langage de programmation classique comme C++ ou Java. Ainsi, Java fournit l'API JMI (*Java Metadata Interface* [Dirckze, 2002]) qui permet l'implantation d'une infrastructure pour gérer la création, l'enregistrement, l'accès, la recherche et l'échange de métadonnées MOF. Si l'utilisateur n'a pas besoin d'apprendre un nouveau langage pour définir les transformations de modèles, les outils de cette catégorie s'avèrent limités dans le cadre de la méta-modélisation et de l'implantation de transformations complexes.
2. Les *outils génériques* tels que XSLT [W3C, 1999] et XQUERY [W3C, 2010] ont profité de la popularité et des nombreux travaux autour d'XML pour atteindre un bon niveau de maturité. S'ils se sont avérés simples et efficaces pour la manipulation des éléments de la syntaxe abstraite, de nombreux travaux ont montré leurs limites en terme de validation et de maintenance et au niveau sémantique des modèles manipulés.
3. Les *outils intégrés aux ateliers de génie logiciel* tel OBJECTEERING [Software, 2012], OPTIMALJ ou FUJABA [Fujaba Core Development Group, 2012] ont pour avantage leur maturité et la qualité de leur intégration dans des ateliers de génie logiciel souvent propriétaires. Cependant, ces outils souffrent aussi de ce dernier avantage car ils sont souvent développés en second plan dans leurs ateliers respectifs et présentent des limites dans le développement (structuration, modularité, réutilisation, maintenance, etc.).
4. Les *outils spécifiques* tel ANDROMDA [AndroMDA, 2012], ATL [ATL, 2012], SMART-QVT [Alizon *et al.*, 2007] ont pour avantage leur simplicité de développement, leur forte expressivité, la maintenance des transformations, l'interopérabilité et la composition des règles de transformation. Bon nombre de ces outils universitaires et industriels de cette catégorie reposent ou s'inspirent du standard QVT de l'OMG que nous présentons dans la section suivante.

5. Les *ateliers (ou outils) de méta-modélisation* reposent sur la programmation par objets pour la construction des modèles de transformations et sur l'IDM pour l'exécution de ces modèles. Une transformation de modèles est ainsi un méta-programme exécutable. Ces outils sont moins nombreux, nous pouvons citer les plus connus KERMETA [Fleurey, 2006], EMF [Steinberg *et al.*, 2009] ou METAEDIT [MetaCase, 2012].

Dans les sous-sections suivantes, nous mettons l'accent sur les ateliers de méta-modélisation EMF et KERMETA, sur le standard QVT et sur l'outil ATL. Une synthèse termine cette section et conclut sur notre choix d'utiliser le couple EMF et ATL pour l'implantation de nos transformations de modèles.

ECLIPSE MODELING FRAMEWORK (EMF/Ecore)

Description. EMF (*Eclipse Modeling Framework* [Steinberg *et al.*, 2009]) est à la fois une plate-forme de modélisation et de génération de code. Son objectif vise à faciliter la construction et le développement rapide d'outils basés sur des modèles structurés et leur intégration au sein de la plate-forme ECLIPSE. De ce fait, l'architecture d'EMF s'articule autour d'un certain nombre de plug-ins. Parmi les plus essentiels, nous pouvons citer : le méta-modèle Ecore, canevas de classes pour la description de modèles EMF et la manipulation des référentiels de modèles ; le canevas de classe *EMF.Edit* pour le développement d'outils d'édition de modèles ; le modèle de génération *GenModel* qui assure la configuration et la personnalisation de la génération de code en Java à partir des éléments décrits au sein d'un modèle EMF (voir figure 2.7). La persistance des instances de modèles est assurée à l'aide de XMI (XML).

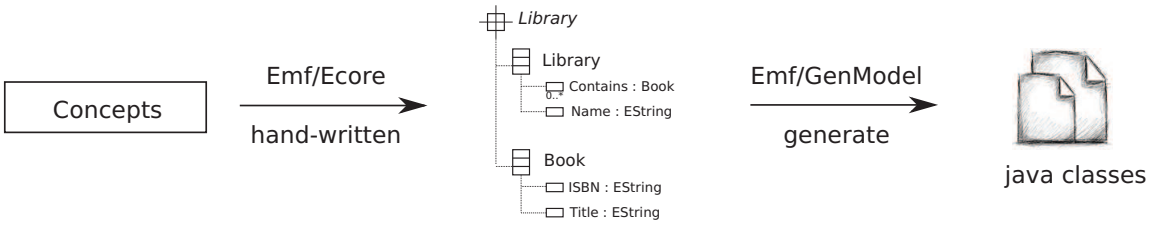

FIGURE 2.7 – Interactions des plug-ins EMF

Enfin, un certain nombre de plug-ins additionnels, développés dans le contexte du *Eclipse Modeling Project* ou par la communauté ECLIPSE en général, s'exécutent au dessus d'EMF et permettent l'interrogation (*Model Query*), la validation (*Validation Framework*) ou la transformation de modèles (EMT), etc.

Discussion. EMF est un atelier de méta-modélisation orienté vers la réalisation de méta-modèles et d'outils dédiés à IDM (représentation, édition, persistance...) à partir de techniques génératives. L'avantage principal de cette technologie est l'importante communauté qui la soutient et les nombreux projets d'IDM venant se greffer au dessus, assurant la maintenance, l'évolution et l'intégration de nombreux standards et outils dédiés à l'IDM. Cependant, son utilisation seule pour effectuer des transformations de modèles reste limitée, nécessitant l'utilisation d'une autre technologie (par exemple ATL que nous détaillons plus bas) pour assurer efficacement les objectifs de transformation de modèles.

KERMETA

Description. KERMETA [Fleurey, 2006; Jézéquel *et al.*, 2011], développé par l'IRISA-INRIA de Rennes [Triskell, 2012], vise à définir un environnement de méta-modélisation qui permet à la fois de spécifier la structure et la sémantique des méta-modèles, d'instancier ces méta-modèles et de réaliser des transformations de modèles. C'est à la fois un langage de méta-modélisation et de programmation impérative. Il offre des supports pour la spécification de contraintes, la spécification comportementale à l'aide d'un langage d'actions et la vérification de modèles.

Une description en KERMETA est comme un programme issu de la fusion d'un ensemble de méta-données (EMOF, Ecore...) et du méta-modèle d'action AS (*Action Semantics* intégré à UML 2.0). KERMETA intervient ainsi à deux niveaux dans l'architecture de méta-modélisation définie par l'OMG (voir figure 2.5). C'est à la fois un langage de niveau *M3* (tous les méta-modèles lui sont conformes) mais également une bibliothèque de base pour la construction de méta-modèles de niveau *M2*.

Un méta-modèle peut être spécifier en EMOF, Ecore ou dans le langage KERMETA. Un support est fourni pour la traduction vers des modèles EMF permettant ainsi d'utiliser l'outillage EMF. Les constructions du langage d'actions permettent de définir des expressions, de naviguer dans les modèles, de créer/modifier des modèles, de parcourir l'arbre syntaxique des méthodes et de modifier la sémantique opérationnelle, etc. Les modèles et les méta-modèles sont enregistrés dans des référentiels. Le moteur de transformation associé au langage lit le modèle source à partir du référentiel, exécute la transformation et écrit le modèle cible dans un référentiel conformément aux spécifications de la transformation.

Enfin, KERMETA supporte la gestion des erreurs et la programmation par aspect mais ne fournit aucun support pour la traçabilité, la multi-directionnalité et la transformation de modèles incrémentale.

Discussions. L'idée principale de KERMETA est de fournir un noyau "commun" pour la spécification de la structure et de la sémantique des méta-modèles afin de palier aux limitations de cohérence et à l'interopérabilité liées à l'hétérogénéité des technologies utilisées (MOF, EMOF ou Ecore pour l'écriture des méta-modèles, OCL ou AS *Action Semantics* pour l'expression de contraintes, Java ou QVT pour l'écriture de transformations, etc.). Si cette idée présente un réel avantage du fait que l'utilisateur n'a pas besoin d'apprendre de multiples technologies, il s'avère que, bien souvent, pour des fonctionnalités manquantes ou des besoins d'interopérabilité, l'utilisateur soit amené à exporter ces modèles vers une autre technologie tel par exemple EMF.

Avec KERMETA, les modèles et les méta-modèles ont besoin d'être explicitement chargés en mémoire et sauvegardés dans un référentiel. Les éléments cibles ont aussi besoin d'être explicitement instanciés et ajoutés au modèle cible. Le contrôle de la stratégie d'application et d'exécution des règles de transformation doit être explicitement spécifié. A la différence de certains outils de transformation de modèles (ATL, SmartQVT, etc.), ces points nécessitent un effort et l'écriture de code supplémentaire dans le langage KERMETA par l'utilisateur.

Query/View/Transformation (QVT)

Description. QVT (*Query/View/Transformation*) est un langage hybride de transformation de modèles standardisé par l'OMG [OMG, 2009]. Il s'aligne sur l'approche MDA [Blanc, 2005] et utilise le langage MOF (*Meta Object Facility* [OMG, 2006c]) pour décrire la syntaxe abstraite

de son méta-modèle. Il offre un langage de requêtes s'appuyant sur le langage OCL [OMG, 2010]. QVT définit une syntaxe textuelle concrète et un méta-modèle basé sur XMI pour créer des représentations de modèles de transformation (voir définition 2.5.1). Il autorise l'intégration et l'invocation d'implantation (du code) "externe" à l'intérieur d'une transformation à l'aide d'un mécanisme de *Black Box*. La traçabilité est un objectif du standard QVT et est supportée de manière automatique.

Langages et structure. La dernière version du standard QVT [OMG, 2009] définit trois langages (et méta-modèles) pour des transformations modèle-à-modèle. Les langages QVT Relational et QVT Core sont des langages déclaratifs et ont des niveaux d'abstraction différents. Le langage QVT Operational est un langage impératif.

- QVT Relational permet de spécifier des relations entre des modèles du MOF et repose sur l'utilisation de patrons d'objets. Une transformation est définie à l'aide d'un ensemble de patrons qui sont instanciés pour la spécification de nouveaux éléments, mis en relation avec des éléments et/ou utilisés pour effectuer des modifications dans le modèle. Une syntaxe textuelle et une syntaxe graphique simple sont fournies. Le langage offre des mécanismes pour la création et la suppression automatiques d'objets, l'expression des relations entre modèles, l'identification des éléments cibles et la gestion automatique implicite des informations de traçabilité entre éléments des différents modèles d'une transformation. L'utilisateur n'a donc pas ces derniers points à gérer. Le langage supporte les transformations bidirectionnelles et le sens de la transformation doit être spécifié à l'exécution.
  Ce langage de relations permet de vérifier et de renforcer la cohérence lors de la modification du modèle cible, de synchroniser les deux modèles et autorise les transformations *in-place*. Le langage OCL permet d'exprimer des requêtes pour l'élaboration de patrons complexes. Enfin, la sémantique est définie par un *mapping* avec le langage QVT Core.
- QVT Core est un langage de transformation de modèles technique de bas niveau. C'est une extension minimale de EMOF et OCL qui sert de fondation pour le langage de relations. Il est défini par une syntaxe textuelle.
- QVT Operational définit la partie impérative de QVT et étend les deux langages déclaratifs précédents en ajoutant des constructions impératives et des constuctions OCL. Il définit une syntaxe concrète plus familière aux utilisateurs des langages de programmation impératifs et des mécanismes de gestion automatique de traces.

Remarque. Un langage d'implantation opaque d'opérations MOF est proposé et définit une *Black Box* pour étendre et invoquer des fonctionnalités de transformation implantées dans un langage supportant le MOF, d'implanter une partie des transformations de manière opaque ou d'utiliser des librairies spécifiques à un domaine (dans le cas de contraintes non exprimables en OCL). Ainsi, une transformation peut être spécifiée dans l'un des langages déclaratifs et implantée de manière impérative dans le langage QVT Operational ou par le mécanisme de *Black Box*.

Outillages. Le standard QVT de l'OMG est une référence pour les outils spécifiques à la transformation de modèles. Ainsi, il existe de nombreuses implantations des langages du standard QVT (ou s'en inspirant fortement tel ATL) commerciales ou libres. Nous présentons brièvement les outils SMARTQVT et MODELMORF.

SMARTQVT [Alizon *et al.*, 2007] est une implantation du standard QVT Operational basée sur EMF. La description de transformation est compilée en code Java. L'outil supporte le mécanisme de *Black Box*, la traçabilité, le contrôle de la stratégie et des paramètres des règles. La transformation incrémentale et la multi-directionnalité ne sont pas supportées.

MODELMORF [Services, 2012] est une implantation du langage standard QVT Relational. Il supporte la multi-directionnalité et une même règle de transformation peut être utilisée pour *mapper* des éléments dans les deux directions. Les transformations *in-place* sont supportées de même que la transformation incrémentale du modèle cible ce qui implique que, lors d'un changement du modèle source, seule la partie changée du modèle est transformée. De plus, il offre un support et des fonctionnalités pour créer ses propres traces. Enfin, la réflexion et la transformation incrémentale du modèle source ne sont pas supportées.

Discussion. QVT a été proposé par l'OMG dans l'objectif de normaliser la définition de transformations de modèles et d'assurer l'interopérabilité des outils de transformation de modèles. Si l'initiative est très intéressante, celle-ci se heurte aujourd'hui à une définition très large de la notion de conformité d'un outil au standard QVT. Ainsi, de nombreux outils tel MODELMORF ou SMARTQVT n'implantent qu'une partie de la norme et ne répondent pas forcément aux attentes en termes de réutilisation et de portabilité des transformations. De plus, le caractère non standard des *Black Box* complexifie fortement la réalisation de ces derniers points. Enfin, la spécification QVT concerne uniquement la définition de transformations de modèle-à-modèle et la normalisation des transformations modèle-à-code est en cours d'étude.

ATLAS TRANSFORMATION LANGUAGE (ATL)

Description. ATL (*ATLAS Transformation Language* [ATL, 2012]) est un langage de transformation modèle-à-modèle hybride - *i.e.* autorisant des constructions déclaratives et impératives - développé par OBEO et supporté par la communauté ECLIPSE. Il s'inspire en grande partie du standard QVT [Jouault *et al.*, 2006]. La structure déclarative est la plus utilisée, elle permet de définir une implantation de transformation plus simple et plus claire. Les constructions impératives visent des descriptions de transformations plus complexes.

ATL supporte les transformations endogènes et exogènes et autorise les descriptions de transformation prenant en entrée (resp. en sortie) un ou plusieurs modèles sources (resp. cibles). Une description de transformation est compilée (en ASM) puis exécutée par le moteur de transformation d'ATL [Jouault *et al.*, 2008]. Dans le mode initial, les transformations sont unidirectionnelles (sources vers cibles), l'ordre d'exécution des règles est déterminé automatiquement et le contrôle de la stratégie d'application des règles n'est pas explicitement permis. Néanmoins, celui-ci peut être influencé par filtrage des patrons du modèle source. La transformation incrémentale de modèles n'est pas supportée - *i.e.* un modèle source complet est lu et un modèle source complet est créé - mais certains travaux autour d'ATL ont montré la possibilité d'obtenir un résultat similaire [Jouault and Tisi, 2010]. ATL supporte la traçabilité. Le *refining mode* d'ATL autorise des transformations *in-place* en limitant l'utilisation et la combinaison des constructions et autorise la suppression des éléments dans le modèle cible.

Enfin, des travaux ont montré la possibilité d'utiliser ATL à des fins de vérification de validation de modèles par l'expression de contraintes sous la forme de requêtes OCL [Bézivin and Jouault, 2006] .

Structure d'une description de transformation. L'exemple de code ATL 2.1 illustre une description de transformation ATL. Celle-ci est composée de règles décrivant comment créer et initialiser les éléments du modèle cible. ATL définit un modèle MOF pour sa syntaxe abstraite et possède une syntaxe concrète textuelle. Un sous-ensemble du langage OCL est supporté et permet d'exprimer, sous forme d'expressions, des requêtes (*helper*) pour accéder aux éléments d'un modèle, naviguer entre les éléments du modèle, appeler des opérations, extraire des informations et exprimer des contraintes ou des gardes sur les patrons.

Une règle déclarative (*Matched Rule)* est constituée d'un nom, d'un ensemble de patrons sources (*InPattern*) désignant des éléments des modèles sources et d'un ensemble de patrons cibles (*OutPattern*) représentant les éléments créés des modèles cibles. Deux constructions impératives sont utilisables : des règles impératives (*Called Rule*), exprimées dans la même syntaxe que les règles déclaratives et appelées explicitement ; un bloc d'instructions impératives (*ActionBlock)* attaché aux deux types de règles dont la syntaxe peut-être du code (par exemple Java).

Exemple 2.1 – Exemple de description de transformation ATL

```
-- @path Families=/Families2Persons/Families.ecore
-- @path Persons=/Families2Persons/Persons.ecore

module Families2Persons;
create OUT : Persons from IN : Families;

helper context Families!Member def: isFemale() : Boolean =
  if not self.familyMother.oclIsUndefined() then
    true
  else
    if not self.familyDaughter.oclIsUndefined() then
      true
  else
      false
  endif
endif;

rule Member2Male {
  from    s : Families!Member (not s.isFemale())
  to      t : Persons!Male (fullName <- s.firstName + ' ' + s.familyName)
}

rule Member2Female {
  from    s : Families!Member (s.isFemale())
  to      t : Persons!Female (fullName <- s.firstName + ' ' + s.familyName)
}
```

Outillages. Les outils de transformation liés à ATL sont intégrés sous forme de plug-in ADT (ATL *Development Tool*) pour la plate-forme de développement ECLIPSE et peuvent gérer les modèles basés sur EMF. Les modèles basés sur des profils UML construits à partir d'EMF sont aussi supportés.

Exécution. ATL supporte deux modes d'exécution. Dans le mode standard, les éléments sont uniquement créés lorsque les patrons sources définis dans les règles déclaratives sont reconnus, puis le système instancie les éléments des patrons cibles. Une fois l'instanciation réalisée, un lien de traçabilité est créé associant chaque élément du modèle source (reconnu) à un élément du modèle cible. Ces liens de traçabilité sont ensuite évalués et permettent de déterminer les propriétés des éléments instanciés.

Dans le mode par raffinement (*refining mode*), tous les éléments du modèle source non reconnus par un patron source sont recopiés automatiquement dans le modèle cible, réduisant les efforts et favorisant ainsi les transformations d'une toute petite partie du modèle.

Discussion. L'atout principal d'ATL réside dans l'expressivité de ces transformations (syntaxe précise et facilement compréhensible), sa compatibilité avec EMF (méta-modèles Ecore) et dans sa facilité d'utilisation (que nous verrons en détail dans le chapitre 8 de ce manuscrit). Son intégration comme projet au sein de la plate-forme ECLIPSE, le support de la communauté ECLIPSE et celui de la société OBEO (qui offre un support industriel) sont autant d'atouts considérables qui ont permis de structurer, de développer et de maintenir le langage et son outil. Le support de la réflexion permet d'utiliser des descriptions de transformation ATL comme modèle source et/ou cible d'une transformation et offre des facilités pour la mise en œuvre des mécanismes d'annotation, de traçabilité, d'évolution et de maintenance efficaces.

Néanmoins, le mode par raffinement proposé supporte un ensemble très faible des constructions du langage et limite fortement l'utilisation de règles impératives. La structure des descriptions de transformation (exemple 2.1) contraint le chainage transformations de modèles et notament, la *superimposition* des descriptions de transformation (permettant la surchage des règles de transformation) a une configuration statique des entrées/sorties des transformations. Enfin, les transformations unidirectionnelles limitent les possiblités de synchronisation entre les modèles sources et cibles.

2.6.3 Conclusion et justification du choix du langage de transformation

Dans cette seconde partie de notre état de l'art, nous avons présenté le principe, les concepts généraux et les problèmes adressés par la transformation de modèles. A travers une "brève" étude des différents approches et outils de transformation de modèles, nous avons présenté les principales caractéristiques attendues et celles supportées par des outils pertinents ayant un bon niveau de maturité.

Notre objectif est de définir des transformations de modèles qui permettent de décrire et d'automatiser un processus de raffinement incrémental d'un système critique modélisé en AADL. L'automatisation de l'ensemble du processus de raffinement nécessite la définition de transformations qui agissent uniquement sur la structure des composant AADL et non sur l'arbre syntaxique de ses méthodes - *i.e* sa représentation technologique interne. De plus, nous visons l'intégration de notre prototype de raffinement par transformation de modèles au sein de l'outil OSATE2, canevas de développement et plate-forme fédératrice pour l'intégration des projets autour du langage AADL, reposant sur ECLIPSE. Cet outil fournit notamment le méta-modèle du langage AADL (en Ecore) réalisé à l'aide des technologies UML 2.0 et EMF. En outre, par contraintes de coûts, temporelle et humaine, nous ne souhaitons pas redévelopper un langage ou un outil de transformation mais réutiliser une technologie libre offrant des perspectives d'interopérabilité avec l'outil OSATE2, de maintenance aisée et d'un support s'inscrivant sur le long terme.

Tous ces éléments nous amènent à éviter la complexité du développement d'un prototype à l'aide des ateliers de méta-modélisation tel KERMETA. Les caractéristiques et les fonctionnalités d'ATL répondent à nos critères, nous permettent la réutilisation du méta-modèle AADL et simplifient l'intégration de notre prototype au sein de l'outil OSATE2 (ces deux outils sont construits au dessus de la plate-forme ECLIPSE). Enfin, le langage ATL nous permet aussi d'assurer notre dernier objectif de production automatisée du code source du système.

2.7 Synthèse

Ce chapitre, divisé en deux parties, nous a permis de traiter de l'état de l'art de deux domaines différents mais en relation, à savoir : l'ingénierie des systèmes critiques et la transformation de modèles.

Dans la première partie, nous avons présenté les différentes *briques* requises (intergiciel critique, approches par composants, validation, déploiement et configuration...) et leur rôle pour la conception et l'analyse de systèmes TR^{2}E critiques. Nous avons vu, lors de l'étude de différentes méthodes intégrées pour la conception de systèmes critiques, le rôle central du "modèle" et son implication au sein du processus de conception. Ce dernier est supporté par un ensemble de méthodes, de standards et technologies définis dans le spectre large de l'*Ingénierie Dirigée par les Modèles* (IDM).

Si l'apport de l'IDM a été conséquent à tous niveaux du cycle de développement logiciel (spécifications et traçabilité des exigences, automatisation de tâches manuelles critiques...), les différentes abstractions pertinentes du système introduisent un biais important entre les analyses des exigences fonctionnelles, non-fonctionnelles et du comportement réalisées *à-priori* sur le système et vis-à-vis de son implantation physique. Ceci se traduit par une différence sémantique voir comportementale plus ou moins importante dûe à la différence de niveau d'abstraction défini pour la modélisation des composants architecturaux et comportementaux utilisés lors des analyses de faisabilité et de performance et celui requis pour leur traduction dans un langage de programmation traditionnel (comme Ada ou C).

Ainsi, nous nous sommes intéressés à différentes approches basées sur la modélisation des systèmes critiques afin de déterminer les limites introduisant ce biais. Ces limitations sont rappelées et détaillées dans le chapitre 3 de ce manuscrit.

La transformation de modèles est une composante fondamentale de l'IDM permettant la manipulation de modèles. A travers un bref état de l'art, nous avons présenté les concepts généraux et les problématiques adressées par ce domaine. Ainsi, nous avons retenu des éléments pertinents à intégrer au sein de notre approche en réponse aux objectifs et aux limitations que nous avons présentés respectivement en introduction et dans la première partie de ce chapitre. Dans le chapitre suivant, nous détaillons le rôle et l'intervention de ces éléments dans notre solution.