


# Thèse de Doctorat

pour l'obtention du grade de docteur délivré par

## Télécom ParisTech

Spécialité : **Informatique et Réseaux**

présentée par

**Fabien CADORET**

# Génération stratégique de code pour la maîtrise des performances de systèmes temps-réel embarqués

Soutenue le X mai 2014 devant le jury composé de :

| | |
|---|---|
| M. Fabrice KORDON | Président |
| M. Xavier BLANC | Rapporteur |
| M. Sébastien GÉRARD | Rapporteur |
| M. Frank SINGHOFF | Examinateur |
| M. Jérome HUGUES | Examinateur |
| M. Laurent PAUTET | Directeur de thèse |
| M. Etienne BORDE | Co-encadrant de thèse |

# Table des matières

# Listings

# Table des figures

# Chapitre 1

# Introduction générale

Les systèmes embarqués sont présents partout dans notre vie quotidienne. Beaucoup d'objets d'utilisation courante sont contrôlés par des systèmes embarqués, plus ou moins dissimulés dans ces derniers : par exemple, le régulateur de vitesse de notre voiture.

Certains de ces systèmes sont particulièrement critiques, comme le système de guidage d'un avion, et nécessitent des procédés de conception et de validation plus poussés. Le logiciel et le matériel doivent passer par des phases de certification qui seront une garantie sur leur bon fonctionnement. Pour réaliser cette certification, différentes méthodologies et standards sont proposés, notamment : *DO-178B*, *ARINC653*. DO-178B traite des problématiques de criticité et de sûreté de systèmes avioniques en fixant des contraintes de développement. Le standard avionique ARINC653 concerne la problématique de sûreté en isolant les composants logiciels les uns des autres lors de leur exécution (isolation spatiale et temporelle).

La généralisation des systèmes embarqués a pour conséquence une complexité accrue dans leur conception et leur validation. A cela s'ajoute des contraintes commerciales qui demandent une production dans des délais de plus en plus courts. Les concepteurs ont alors recours à des processus de génération de code qui facilitent la conception, la validation et l'implantation du système. Le concepteur fournit une abstraction du système à partir de laquelle il valide les exigences et traduit automatiquement cette abstraction en code exécutable. L'utilisation d'un tel procédé a donc pour objectif de faciliter les différentes étapes qui interviennent dans la production du système.

Cependant, cela suppose une certaine fiabilité du processus de génération. En effet, pour garantir une validation cohérente, le code exécutable généré doit être équivalent à l'abstraction du système en terme de propriétés et de performances : en particulier, il faut s'assurer que le temps d'exécution et l'occupation mémoire réels correspondent à ceux spécifiés. La difficulté initiale de mettre en place un système valide repose donc en grande partie sur la capacité à assurer la fiabilité du processus de génération. L'objectif initial d'un tel processus est de faciliter le travail de l'ingénieur en lui évitant de spécifier les détails d'implémentation. Ce processus a un impact sur la capacité à préserver les propriétés initialement indiquées dans la spécification : plus le modèle est de haut-niveau d'abstraction, plus sa spécification est simple, mais plus le travail du générateur de code

est important. Ainsi, faciliter la conception et obtenir une validation fiable sont des objectifs qui semblent incompatibles.

Les approches de l'Ingénierie Dirigée par les Modèles (IDM) proposent des techniques intéressantes qui peuvent répondre en partie à cette problématique. La notion de transformation de modèle-à-modèle définit un processus automatisé traduisant un modèle initial en un second modèle. Dans le contexte de la génération de code, l'utilisation d'une transformation aurait pour objectif de transformer un modèle de haut-niveau d'abstraction en un second modèle de plus bas-niveau d'abstraction (proche du code exécuté). De cette manière, on facilite la conception en épargnant à l'utilisateur les détails d'implémentation, en traduisant automatiquement ce modèle en un modèle plus précis afin d'aboutir à une validation plus fiable. Cependant, dans le cadre des systèmes embarqués, ce type d'approche n'a été expérimenté que dans des contextes bien spécifiques (*e.g.* systèmes synchrones). Aucune méthodologie n'est proposée pour adapter le générateur à différentes plates-formes d'exécution. De plus, les phases d'analyse et de génération s'enchaînent le plus souvent manuellement. Par conséquent, le modèle validé peut éventuellement aboutir à un code final n'assurant plus les propriétés vérifiées par le modèle. L'utilisateur doit alors avoir un niveau d'expertise élevé sur l'ensemble du processus pour assurer la validité du code généré.

## 1.1 Objectifs

Dans ce manuscrit, nous proposons une démarche d'ingénierie dans le cadre des systèmes embarqués ayant pour objectif de faciliter à la fois la spécification du système et la conservation des propriétés tout au long du processus. Notre premier objectif est de proposer une méthodologie pour organiser l'architecture du générateur de code en nous appuyant sur l'IDM pour maîtriser l'impact du code généré. On souhaite obtenir un générateur articulé autour d'une succession de raffinements, chacun rapprochant progressivement le modèle vers le code final, et dont l'impact peut être évalué par une analyse spécifique. A chaque étape de raffinement, le générateur doit pouvoir décider s'il est nécessaire de changer sa stratégie pour assurer le respect des contraintes du système. La mise en place d'un tel processus pouvant se révéler coûteux, notamment si différentes stratégies de génération doivent être définies, notre second objectif est alors de proposer une méthode pour factoriser leur développement. Enfin, le troisième objectif est d'adapter les composants logiciels pour maîtriser les performances du code exécuté. Pour prendre en compte l'impact de ces composants sur les performances, ceux-ci doivent être intégré lors du raffinement du modèle.

## 1.2 Contexte

Cette section aborde le contexte de notre travail, les systèmes embarqués temps-réel critiques. Nous commençons par définir ces systèmes et à traiter de leur processus de développement. Ensuite, nous abordons les différentes contraintes liées à la mise en place d'un tel processus.

### 1.2.1 Systèmes Embarqués Temps-Réel Critiques (SETRC)

Les systèmes embarqués temps-réel critiques (SETRC) correspondent à une famille de systèmes qui regroupe plusieurs catégories : le critique, l'embarqué et le temps-réel. Chacune implique un processus d'ingénierie bien spécifique :

- Un système est considéré **critique** si une défaillance particulière est capable d'engendrer des conséquences grâves comme des dégâts sur l'homme (cas du domaine des transports ou du médical), économiques, ou environnementales (cas des centrales nucléaires). Ces systèmes doivent ainsi être rigoureusement conçus et validés avant d'être implantés et mis en œuvre.
- Un système **embarqué** est défini comme un système autonome piloté par un logiciel intégré. Ces systèmes sont constitués d'un ensemble de capteurs et d'actionneurs qui détectent des phénomènes de l'environnement pour les uns (*e.g.* température, lumière, contact), et produisent des phénomènes physiques pour les autres (*e.g.* mouvement, son, lumière). Ils disposent de ressources limitées par rapport aux systèmes non-embarqués : mémoire, énergie... Par conséquent, ils doivent être conçus en tenant compte de ces différentes contraintes (*e.g.* choix des algorithmes, représentation en mémoire des informations).
- Un système **temps-réel** réalise un ensemble d'activités qui doivent respecter des délais imposés. Pour garantir le respect des contraintes temporelles, chaque activité doit s'exécuter dans un temps borné. De plus, l'entrelacement des exécutions des différentes activités doit aussi garantir le respect de ces délais. L'ordonnançabilité doit être assurée à la conception.

Un système temps-réel embarqué critique (SETRC) intègre ces trois aspects et présente ainsi des sources de défaillance telles que le sous-dimensionnement mémoire et le dépassement des échéances temporelles.

**Évaluation de l'empreinte mémoire** Le caractère embarqué du système implique des ressources limitées comme la mémoire. Le système déployé sur un OS embarqué doit tenir compte de la capacité de stockage de ce dernier. La mémoire doit être suffisante pour stocker d'une part, les données spécifiées à la conception, les données propres à l'OS mais également celles introduites lors de la génération de code. La difficulté est alors d'estimer précisément l'empreinte mémoire liée à la génération de code.

**Évaluation de l'ordonnançabilité** Cette contrainte est liée au caractère temps-réel du système. Celui-ci est modélisé par un ensemble de tâches ayant leurs propres contraintes temporelles. Chaque tâche réalise l'une des activités du système dans un délai borné et selon un schéma d'activation spécifique. Par exemple, une tâche va s'activer périodiquement toutes les 10 ms pour traiter les données reçues par un capteur dans un délai borné à 3 ms. Pour assurer le bon fonctionnement du système, il faut garantir que la tâche ait le temps de traiter les données avant sa prochaine activation en tenant compte du délai dû à l'exécution des autres tâches.

### 1.2.2 Cycles de développement

Dans le cadre des systèmes embarqués, il est nécessaire de disposer d'une méthode de développement qui prenne en compte le mauvais respect des exigences lors de la phase de conception. Plusieurs méthodes de développement existent. La figure 1.1 en donne un exemple. La phase

initiale est l'expression et le recueil des exigences. Dans un second temps, une architecture candidate est proposée pour répondre à ces exigences. Elle est ainsi ensuite analysée pour déterminer si celle-ci ne remet pas en question les exigences. Si ce n'est pas le cas, elle doit être modifiée en conséquence. Enfin, l'architecture validée est traduite en code exécutable. Ce cycle peut de nouveau inclure une phase de test du code généré. Dans ce cas, le code est exécuté pendant une durée fixée afin d'obtenir des traces d'exécution. Cependant, l'analyse à l'exécution est souvent moins fiable car elle illustre rarement le scénario du pire cas.

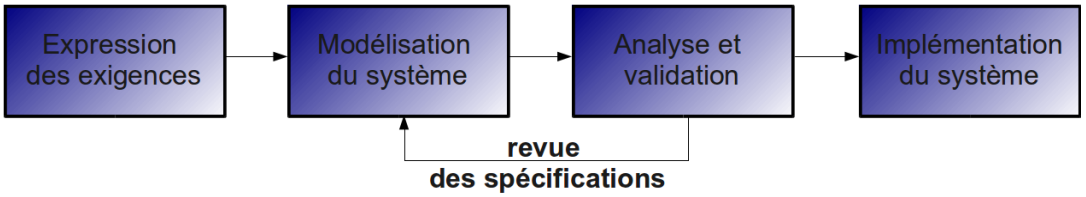



FIGURE 1.1 – Exemple de cycle de développement d'un SETRC

Dans la suite de ce manuscrit nous utiliserons le terme de *générateur de code* pour parler du processus global de développement allant du modèle initial jusqu'au code exécuté, en passant par les étapes d'analyses. Nous utiliserons le terme *génération* pour indiquer la phase finale produisant le code exécutable.

## 1.3 Plan

Ce manuscrit est organisé comme suit. Le chapitre 2 présente l'état de l'art autour de la génération de code pour les systèmes considérés. Ensuite, le chapitre 3 traite les différentes problématiques soulevées. Au chapitre 4 nous introduisons notre approche et nous la détaillons plus précisément au chapitre 5. Puis, le chapitre 6 présente la mise en œuvre de notre contribution que nous illustrons au chapitre 7. Enfin, le chapitre 8 conclut sur notre travail.

# Chapitre 2

# Etat de l'art

Ce chapitre passe en revue l'état de l'art concernant différents aspects liés au développement de générateurs de code pour SETRC. Face aux problématiques soulevées par les SETRC, ces processus de génération doivent assurer une maîtrise du code généré. En effet, la compilation du code généré ne doit avoir lieu que pour un système préalablement validé. Nous nous intéressons ainsi aux processus de génération qui sont conditionnées par des analyses préalables réalisées à partir d'une modélisation du système.

En section 2.1 nous présentons les principaux générateurs de code pour SETRC qui sont définis dans cette optique. La section 2.2 aborde l'impact des éléments d'implémentation sur les propriétés du système et comment les prendre en compte en amont du processus. Notamment, la section 2.3 présente les principaux langages de modélisation architecturale pour SETRC et leur capacité à changer de niveau d'abstraction. Ensuite, en section 2.4 nous nous intéressons aux différentes techniques pour diminuer le niveau d'abstraction d'un modèle architectural. Grâce à ces techniques, l'objectif est d'assurer la conformité du code généré en analysant le modèle bas-niveau. Enfin, la section 2.5 passe en revue différents outils qui permettent de maîtriser le coût du code généré en orchestrant ces transformations.

## 2.1 Frameworks de génération de code pour SETRC

Le premier aspect que nous abordons est la génération de code pour SETRC. Nous nous focalisons sur les processus qui assurent les trois phases suivantes : la modélisation du SETRC, sa validation et sa traduction en code exécutable. La traduction automatique du modèle en code exécutable permet de s'abstraire de la complexité de mise en œuvre du système. Dans cette section, nous présentons différents générateurs de code s'appuyant sur cette démarche. Pour chacun, nous introduisons le formalisme dans lequel est décrit le système. Nous traitons en particulier des objectifs fixés par le générateur en terme de validation et nous soulignons ses limitations éventuelles.

### 2.1.1 OCARINA

Ocarina [49] est une suite d'outils développée à Télécom ParisTech pour la vérification et la génération de SETRC modélisés en langage AADL [83]. Ocarina assure l'analyse syntaxique et sémantique des modèles AADL et fournit des passerelles vers des outils d'analyse spécifiques :
– CPN-AMI [41] : réseaux de Pétri (simulation, model checking)
– Cheddar [85] : analyse d'ordonnancement (tests de faisabilité et simulation)
– Bound-T [37] : analyse statique du code généré (calcul du WCET)
– REAL [38] : analyse de contraintes (conformité à des patrons architecturaux)

```
1  thread receiver
2  features
3    datain: in event data port Base_Types::Integer_16 { Queue_Size => 5; };
4  properties
5    Compute_Entrypoint => classifier (receiver_job);
6    Compute_Execution_Time => 2 ms .. 3 ms;
7    Dispatch_Protocol => Periodic;
8    Period => 10 ms;
9    Deadline => 10 ms;
10 end sender;
11
12 processor cpu1
13 properties
14   Scheduling_Protocol => RATE_MONOTONIC_PROTOCOL;
15 end cpu1;
16
17 process implementation p.impl
18 subcomponents
19   sender: thread sender;
20   receiver: thread receiver;
21 connections
22   port sender.dataout -> receiver.datain;
23 end p.impl;
```

Listing 2.1 – Modélisation d'un système en AADL

Le listing 2.1 donne un exemple de modèle AADL dans sa notation textuelle. Le modèle AADL est constitué d'un ensemble de composants de différents types. Ceux-ci peuvent être connectés à travers leurs *features* pour modéliser des échanges entre ces composants. Ils sont également annotés avec des propriétés qui précisent leur sémantique d'exécution et leur déploiement. Par exemple, la propriété *Period* d'un composant *thread* indique une activation périodique d'une certaine fréquence. Le composant *processor* modélise les éléments matériels et logiciels responsables de l'exécution des *thread*. Il est annoté pour préciser l'ordonnancement des *threads*.

**Démarche d'analyse** L'objectif d'Ocarina est d'assurer d'une part que la génération de code n'ait lieu que pour un système pour lequel les contraintes de dimensionnement et d'ordonnançabilité ont été vérifiées au préalable : en réalisant des traductions du modèle AADL vers des formats compatibles avec les outils d'analyse. D'autre part, le second objectif est de minimiser l'écart entre le modèle et le code exécutable en optimisant l'empreinte mémoire et le surcoût temporel du binaire final (support d'exécution + applicatif généré). Pour cela, il s'appuie sur des supports d'exécution (intergiciels/micro-noyaux) conçus dans cette optique : PolyORB-HI-C, PolyORB-HI-Ada [48] et POK [27]. Plus précisément, Ocarina détermine à partir du modèle AADL l'ensem-

ble des fonctionnalités du support d'exécution qui sont requises pour mettre en œuvre le système modélisé : les fonctionnalités non requises ne sont alors pas incluses dans le binaire. Par exemple, si le modèle AADL spécifie l'utilisation de communications à travers un réseau, alors Ocarina va générer une directive appropriée pour le compilateur de manière à inclure les pilotes réseau dans le binaire. Dans le cas contraire, le binaire ne contiendra pas ces pilotes. Ocarina détermine ainsi les ressources requises et configure le support d'exécution en conséquence. Cela renforce également le déterminisme du code généré en dimensionnant statiquement les ressources nécessaires (évitant l'utilisation d'allocation dynamique). Par exemple, le nombre de tâches et de files de messages sont déterminés statiquement. Les constantes correspondantes sont générées afin de dimensionner les structures contenant l'ensemble des tâches et des files de message.

**Limitations** La démarche d'analyse d'Ocarina intervient donc essentiellement lors de la phase de conception. Le niveau d'abstraction assez élevé du modèle AADL facilite la spécification mais rend difficile l'évaluation du surcoût de la génération de code. Le surcoût du code généré, même minimisé par le support d'exécution, n'est pas pris en compte lors de la validation du système. En effet, la génération introduit nécessairement des fonctionnalités (*e.g.* composants logiciels) du support d'exécution qu'il faut prendre en compte. Par conséquent, Ocarina limite l'écart entre modèle et code mais n'assure pas complètement la validité du code généré.

### 2.1.2 SynDEx

Pour assister les concepteurs de systèmes temps-réel, le centre de recherche de l'INRIA Paris-Rocquencout propose la méthodologie appelée AAA [39, 40] (*Algorithm Architecture Adequation*) et son logiciel de conception assistée par ordinateur (CAO) SynDEx [39]. SynDEx se focalise sur les systèmes synchrones multi-processeurs et son objectif est de minimiser les ressources mises en œuvre pour l'exécution du programme sur la machine cible. La spécification fonctionnelle de ces systèmes est écrite en langage SIGNAL [80] et formalisée sous forme de graphe de flot d'exécution : chaque noeud représente une activité de calcul pouvant être détaillée dans un sous-graphe. La figure 2.1 donne un exemple de flot d'exécution modélisé avec SynDEx.

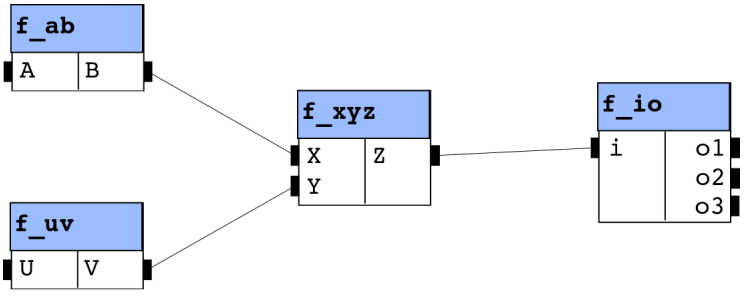



FIGURE 2.1 – Représentation du flot d'exécution avec SynDEx

Le flot d'exécution est constitué de quatre blocs fonctionnels. Les paramètres d'entrée de chaque bloc sont énumérés dans leur colonne de gauche tandis que les paramètres de sortie sont énumérés dans leur colonne de droite. Le lien entre deux blocs modélise une connexion entre un paramètre de sortie du premier au paramètre d'entrée du second.

**Démarche d'analyse** L'objectif de SynDEx est de déterminer un déploiement approprié d'un modèle exclusivement fonctionnel selon l'impact du déploiement sur les performances du système final. En effet, SynDEx prend en compte les ressources disponibles (nombre de processeurs, mémoire, moyens de communication) pour répartir le programme (*i.e.* les blocs fonctionnels) sur les processeurs. Le graphe de flot d'exécution est alors décomposé en sous-graphes répartis sur chaque processeur. L'impact de la répartition choisie sur les performances de l'application est mesuré par simulation [28]. Les différentes répartitions peuvent être explorées manuellement par l'utilisateur ou automatiquement à l'aide d'heuristiques [92]. Une fois la combinaison optimale identifiée, le code fonctionnel est automatiquement réécrit pour tenir compte de la répartition choisie. La durée d'exécution des activités de calcul est alors réevaluée en prenant en compte le coût des mécanismes de communication introduits lors de la réécriture. La spécification fonctionnelle est ensuite traduite en code exécutable pour des plates-formes d'exécution dédiées.

**Limitations** La démarche d'analyse de SynDEx est donc réalisée sur la modélisation en SIGNAL du système. Contrairement à Ocarina, le système est partiellement modélisé puisqu'il se limite à une description fonctionnelle proche du code généré. La répartition des blocs fonctionnels et le déploiement (*e.g.* canaux de communication) ne sont ainsi pas initialement modélisés mais sont introduits lors de l'analyse en restructurant le modèle SIGNAL. Par rapport à Ocarina, SynDEx intervient donc à un niveau d'abstraction différent, le déploiement des blocs fonctionnels n'étant pas connue initialement. Par conséquent, SynDEx n'est pas contraint par un choix d'organisation imposé par l'utilisateur. La démarche d'analyse de SynDEx semble donc plus flexible que la précédente. Cependant, SynDEx ne concerne pas les mêmes types de système que Ocarina. En effet, l'un concerne les systèmes synchrones et l'autre les systèmes asynchrones.

### 2.1.3 OASIS

OASIS [9] est un modèle d'architecture Time-Triggered développé par le CEA dans le cadre des SETRC pour centrales nucléaires [25]. Afin de répondre à la difficulté d'assurer le respect de contraintes de sûreté tels que les contraintes temporelles, OASIS définit un modèle de tâches et d'interactions qui garantit un déterminisme temporel. Ces modèles sont implémentés dans le noyau d'OASIS. En effet, les tâches sont écrites dans un langage spécifique qui introduit des contraintes temporelles : $\psi C$ et $\psi Ada$ [90], dérivés du C et de l'Ada, introduisent l'instruction *ADV* qui spécifie une contrainte de temps exprimant *"l'opération courante doit se terminer au plus tard à l'instant logique t, et l'opération suivante ne peut démarrer avant cet instant"*. Ainsi, chaque tâche dispose d'une horloge qui caractérise les instants auxquels ses entrées/sorties sont observées. Dans le cas d'une tâche classique, cette horloge représente ses dates d'activation. Entre deux instants consécutifs, le temps est considéré figé et la tâche ne peut observer la réception de nouveaux messages. Ces derniers ne seront visibles qu'au prochain instant. Pour une tâche classique, cela correspond à dépiler les messages en début d'activation. Il est cependant possible de définir des dates d'échéances intermédiaires pour chaque opération réalisée dans le job de la tâche. Dans ce cas, la date d'échéance d'une opération caractérise sa date de terminaison au plus tard mais aussi la date d'exécution au plus tôt de l'opération suivante.

**Démarche d'analyse** La démarche d'analyse d'OASIS consiste donc à définir un modèle de tâche pour lequel les interactions sont déterministes et reposent sur une approche *Time-Triggered*. OASIS fournit deux types d'interactions : les variables temporelles et les messages. Les mécan-

ismes mis en œuvre n'occasionnent aucun temps de blocage. Cela est rendu possible par une mise à jour différée des variables partagées (voir [20]) et par des échanges de messages entre producteur et consommateur à la charge du système. Ces interactions sont spécifiées via des instructions spécifiques : l'instruction $SEND(i,t)$ indique un envoi à la tâche $i$ d'un message qui sera visible $t$ instants après l'instant (logique) courant. Ces modèles d'exécution et ces instructions sont ainsi implantés dans le noyau d'OASIS et assurent les contraintes de temps. Cela repose sur la définition d'une horloge globale régulière $H_\Omega$. Cette horloge couvre l'ensemble des instants définis par les instructions $ADV$ de toutes les tâches. Ces instants n'étant pas espacés dans le temps de manière régulière, des instants intermédiaires sont définis afin d'obtenir un tick régulier. L'obtention de l'horloge régulière permet ensuite de transformer chaque tâche en un diagramme d'états-transitions dont les états sont les contraintes temporelles exprimées en nombre de ticks de $H_\Omega$. Les transitions sont annotées des instructions réalisées, comme le montre la figure 2.2.

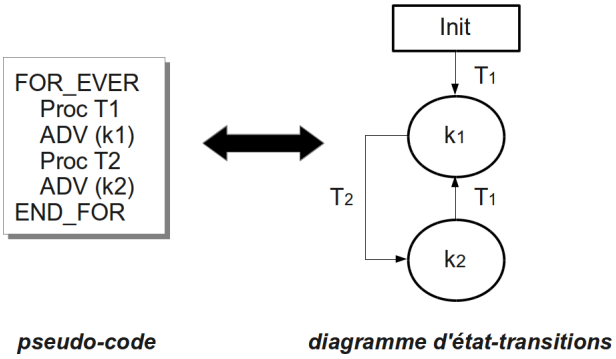



FIGURE 2.2 – Représentation d'une tâche définie avec OASIS

A chaque itération, la tâche exécute une fonction $T_1$. Ensuite, l'instruction $ADV(k_1)$ indique que $T_1$ doit se terminer au plus tard $k_1$ instants logiques plus tard que l'instant courant. L'instruction $ADV(k_1)$ indique également que $T_2$ ne commencera son exécution qu'à partir de la date $k_1$, indépendamment de la date de terminaison de $T_1$. Ensuite, l'instruction $ADV(k_2)$ indique que $T_2$ doit se terminer $k_2$ instants après l'instant courant $k_1$ et que la prochaine exécution de $T_1$ ne débutera qu'à partir de cette date.

**Limitations** A l'instar de SynDEx, OASIS intervient à un niveau d'abstraction assez proche du code exécutable. L'écart entre le modèle et le code est donc relativement limité. Cependant, cette approche repose entièrement sur l'utilisation du noyau OASIS. Elle est donc fortement liée à l'implantation d'OASIS et à ses mécanismes. La principale limitation de ce processus de génération est donc qu'il est entièrement dédié à une unique plate-forme d'exécution, contrairement à Ocarina et SynDEx, et par conséquent il ne propose pas de solution générique.

### 2.1.4 Giotto

Giotto [46] est un modèle de programmation abstrait pour SETRC qui définit des systèmes de contrôle composés de tâches périodiques, de capteurs et d'actionneurs. Il a été expérimenté sur différents projets industriels, notamment [53]. En Giotto, un système de contrôle est modélisé par un ensemble de composants connectés par des ports comme le montre la figure 2.3 qui définit une

tâche *T*. Cette modélisation spécifie les dates logiques d'activation des tâches, de lecture des capteurs et de mise à jour des actionneurs indépendamment de la plate-forme cible et du déploiement (*e.g.* nombre de noeuds, répartition des tâches).

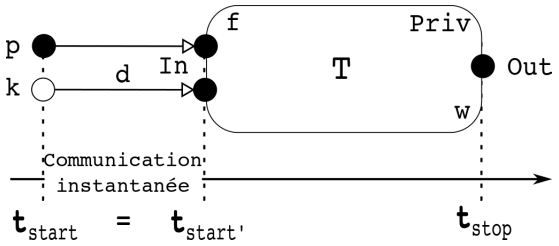



FIGURE 2.3 – Représentation d'une tâche définie avec Giotto

Sur cet exemple, la tâche *T* dispose d'un ensemble *In* de ports d'entrée et un ensemble *Out* de ports de sortie. Chaque port contient une donnée et la conserve jusqu'à ce qu'il soit mis à jour. Sur cette figure, le premier port est mis à jour par un autre port *p*, tandis que le second est initialisé avec une constante *k* convertie au format de la donnée du port par le *driver d*. La mise à jour des ports d'entrée est considérée instantanée, c-a-d dans un temps logique nul, et ne peut être interrompue. A chaque invocation de fréquence *w*, la tâche *T* exécute sa fonction *f* prenant en paramètres les valeurs des ports *In* et le mode courant *Priv* de la tâche. L'invocation démarre à une date $t_{start}$ et se termine à la date $t_{end}$. La taille de l'intervalle entre $t_{start}$ et $t_{stop}$ est déterminé par la fréquence *w* de la tâche. Cette fréquence peut changer selon le mode courant de la tâche. La sémantique du modèle garantit qu'à la date $t_{stop}$, les ports de sortie *Out* de la tâche *T* sont mis à jour avec le résultat de la fonction *f*, sans préciser à quel moment la fonction *f* est exécutée.

**Démarche d'analyse** Giotto ne précise ni comment ni quand les tâches sont ordonnancées. C'est lors de la compilation sur une plate-forme donnée que le compilateur résout le problème d'ordonnancement. Par exemple, un même modèle peut être compilé sur une plate-forme composée d'un ou plusieurs nœuds. Le compilateur assure alors que les contraintes du modèle sont préservées (*e.g.* contraintes de temps). Pour cela, il s'appuie sur une machine virtuelle temps-réel supervisant les aspects temporels [44]. Il est possible d'annoter le compilateur pour spécifier des contraintes liées au déploiement et à l'ordonnancement des tâches et des communications : affectation d'une tâche à un noeud, affectation d'une priorité à une tâche, traitement d'un événement à une date précise.

**Limitations** L'approche de Giotto est similaire à OASIS en reposant sur un support d'exécution spécifique. Son utilisation requiert une implémentation spécifique de la machine virtuelle pour chaque plate-forme d'exécution visée. Une nouvelle implémentation doit ainsi être fournie pour chaque plate-forme. De plus, les mécanismes de communication semblent être également imposés. Giotto ne semble pas conçu dans l'optique de s'intégrer dans une architecture existante. Comme pour SynDEx, Giotto repose sur une spécification partielle du système et sur un déploiement déterminé automatiquement lors de la compilation. De plus, il se focalise sur les systèmes synchrones et ne semble pas adapté aux systèmes asynchrones.

Nous avons présenté différents générateurs de code pour SETRC. Le domaine de l'embarqué temps-réel nécessite d'assurer le respect de certaines contraintes telles que les contraintes de temps et de dimensionnement. Ainsi, ces générateurs s'appuient sur une validation préliminaire du modèle afin d'assurer que la génération ne s'applique que pour des spécifications valides. L'impact de la génération de code sur les contraintes préalablement validées est également pris en compte. Ocarina s'appuie sur des supports d'exécution hautement configurables dans le but de minimiser l'empreinte mémoire et le surcoût temporel. Cependant, le coût d'utilisation des composants intergiciels n'est pas pris en compte. Ocarina nécessite une modélisation complète du système, notamment la répartition des tâches ou encore le nombre de nœuds du réseau. A l'opposé, SynDEx, OASIS et Giotto se focalisent sur la modélisation fonctionnelle et font abstraction des aspects liés au déploiement. Par conséquent, leurs processus de génération disposent de plus de flexibilité pour sélectionner des configurations appropriées aux contraintes. Cependant ces trois processus ciblent des systèmes synchrones contrairement à Ocarina. Giotto et OASIS contraignent la plate-forme d'exécution par l'utilisation d'une surcouche particulière supervisant les aspects temporels. Ces processus de génération imposent donc un ensemble de restrictions pour assurer la conformité du code généré vis à vis des contraintes. Dans la section suivante, nous abordons comment l'évaluation de l'écart entre le modèle et le code généré peut être renforcée.

## 2.2 Évaluation de l'écart entre modèle et code exécutable

La génération de code pour SETRC semble ainsi peu flexible en reposant sur un support d'exécution dédié optimisé pour assurer le respect des contraintes du système. Ces processus de génération ne fournissent pas une réponse générique applicable dans un cadre plus général. Dans cette section, nous abordons la prise en compte des éléments d'implémentation en amont de la génération de code. En section 2.2.1, nous mettons en avant ces éléments impactant les contraintes du système. Ensuite, en section 2.2.2 nous décrivons comment ceux-ci peuvent être pris en compte au sein du modèle afin d'obtenir une spécification plus précise.

### 2.2.1 Modèle d'exécution et choix de mise en œuvre

La difficulté d'analyser le comportement d'un SETRC est liée aux mécanismes complexes mis en œuvre. Ces mécanismes dépendent du support d'exécution qui propose notamment des moyens de communication en local ou à travers un réseau : files de messages, variables partagées... Le support d'exécution impose donc des mécanismes spécifiques. Tous ces mécanismes influencent le comportement du système et le prédire avec précision et fiabilité est souvent difficile : quel impact sur le temps de réponse du système ? quel sera le contenu de telle file de message à l'instant t ? comment dimensionner les files de messages de façon optimale ? Par conséquent, on restreint le modèle d'exécution du système afin de simplifier son comportement et améliorer son analyse. Nous traitons les deux familles (synchrone/asynchrone) de SETRC dans les paragraphes suivants qui proposent des mises en œuvre différentes pour répondre à la cette problématique.

**Systèmes asynchrones** Les systèmes asynchrones consistent à séparer les différentes activités du système dans des fils d'exécution distincts. Le système est alors constitué d'un ensemble de tâches qui s'exécutent en parallèle. Cette approche facilite la spécification des activités, chacune

étant implémentée indépendamment des autres. Cependant, l'approche asynchrone soulève certains problèmes d'ordonnancement :

– Dans le cas de communications à travers des files de messages, c'est-à-dire basées sur le modèle *producteur/consommateur*, la problématique est d'assurer qu'au moment d'exécuter le consommateur, sa file de messages ne doit pas être vide. Une solution est d'ajouter une contrainte de précédence [14, 21] afin d'autoriser l'exécution du consommateur seulement si le producteur a terminé son exécution (et a déposé un message dans la file). Néanmoins, cette solution impose un producteur et un consommateur périodiques de même période.
– Concernant l'utilisation de variable à état partagé, l'exclusion mutuelle entre les producteur et consommateur doit être assurée puisque la donnée est accédée potentiellement à n'importe quel moment au cours de l'exécution. Le phénomène d'inversion de priorité impose l'utilisation de protocole d'exclusion mutuelle comme PCP [84] ou SRP [10]. Ces protocoles ont pour objectif d'assurer l'accès exclusif à la variable partagée à l'une ou l'autre des tâches demandeuses jusqu'à ce qu'elle libère la ressource. Par conséquent, les tâches en attente sont bloquées jusqu'à ce que la donnée devienne disponible. Cela a donc un impact non négligeable sur le temps de réponse de ces tâches et peut les conduire à manquer leurs échéances. Des patrons de conception sont cependant proposés pour optimiser l'utilisation de ces protocoles [86].

L'analyse du comportement des systèmes asynchrones est donc complexe en raison du fait qu'il est difficile de prédire quelle tâche s'exécute à tel moment et quand accède t-elle aux ressources partagées.

**Systèmes synchrones** Dans le cas des systèmes synchrones, les changements d'état des ressources partagées sont visibles à des instants prédéterminés indépendamment du flot d'exécution. En particulier, une première approche s'est d'abord intéressée à l'ordonnancement hors-ligne, c'est-à-dire à définir statiquement quelle action doit être réalisée à chaque instant [56]. Le flot d'exécution étant connu statiquement, on peut alors prédire très précisément le comportement du système. Cependant, face au manque de flexibilité de cette approche, d'autres solutions ont été proposées :

– Dans [20] est mis en avant l'idée d'exprimer des contraintes temporelles uniquement sur les tâches complexifie la conception et donc l'analyse. L'approche proposée est de définir des modèles orientés données et non orientés tâche. Ce modèle, le Time-Sensitive Object (TSO), modélise l'évolution des données au cours du temps à travers un historique. Chaque valeur contenue dans l'historique possède un intervalle de validité qui décrit à quel moment et pendant combien de temps la valeur observée est considérée valide et fiable.
– Cette approche a été mise en œuvre dans OASIS [9] et a été étendue aux files de messages. Par rapport aux approches précédentes, celle-ci évite des scénarios de blocage à cause de files de messages vides ou de variables partagées non disponibles. En effet, la conservation d'un historique de valeurs permet d'une part d'extrapoler une nouvelle valeur d'une donnée si une file est vide et d'autre part d'éviter des temps de blocage sur une variable partagée en utilisant des "versions" différentes de la donnée entre l'écrivain et les lecteurs.

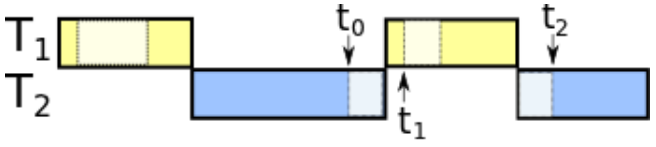



FIGURE 2.4 – Time-Sensitive Object : absence de blocage entre lecteur et écrivain

La figure 2.4 illustre ce modèle à travers une tâche lecteur *T1* et une tâche écrivain *T2*. L'écrivain *T2* modifie la donnée partagée entre les dates $t_0$ et $t_2$. Entre temps, le lecteur *T1* accède à la donnée à la date $t_1$ sans être bloqué car il consulte une version de la donnée antérieure à celle actuellement modifiée par l'écrivain.

Les systèmes synchrones semblent ainsi plus simples à analyser et ne nécessitent pas de mécanismes complexes pour assurer la cohérence du flot de données. Cependant, leur spécification est plus contraignante car il faut définir le flot d'exécution global et déterminer comment les activités doivent être entrelacées. A l'opposé les systèmes asynchrones sont plus simples à spécifier mais requièrent des mécanismes complexes qui rendent difficile leur analyse. La mise en place d'un SETRC nécessite cependant de favoriser à la fois la conception et l'analyse du système. Ces deux familles de systèmes apportent donc des réponses complémentaires : les systèmes asynchrones facilitent la spécification tandis que les systèmes synchrones facilitent l'analyse.

La spécification d'un SETRC est ainsi traduite en un ensemble d'éléments de mise en œuvre qui vont avoir un impact particulier sur les performances et sur les contraintes du système. Nous l'avons illustré dans le cas des communications. Ces éléments sont néanmoins en partie masqué lors de la spécification et par conséquent il est difficile de les prendre en compte lors d'une analyse préliminaire du système (lors de la conception). Face à cette problématique, les processus de génération de code présentés dans la section précédente proposent plusieurs solutions. Les approches synchrones s'orientent plutôt vers une spécification fonctionnelle proche du code. L'approche asynchrone d'Ocarina utilise au contraire une modélisation architecturale et repose sur des plates-formes d'exécution hautement configurables pour optimiser l'impact sur l'empreinte mémoire et le temps d'exécution. Pour à la fois faciliter la spécification mais aussi l'analyse, une troisième solution est de combiner les techniques du synchrone et de l'asynchrone. Cela consiste alors à mettre en place un processus qui, à partir d'une spécification de haut-niveau d'abstraction, intègre automatiquement les éléments de mise en œuvre, notamment les solutions proposés dans le cas du synchrone. L'objectif est ainsi de traduire la spécification initiale en une spécification détaillée proche du code exécuté et analysable. Ce type de processus existe dans un contexte plus général que les SETRC et est présenté dans la section suivante.

### 2.2.2 Modélisation du code généré

Pour répondre à la difficulté croissante de concevoir et analyser des systèmes de plus en plus complexes et à une palette de plus en plus large de technologies, l'Object Management Group (OMG) a introduit l'Architecture Dirigée par les Modèles [68] (MDA). Celle-ci place le modèle au centre du processus de production à travers des techniques de manipulation et de transformation. La MDA définit une approche qui facilite la conception en séparant les préoccupations que sont la spécification fonctionnelle (comportement du système modélisé) et l'implémentation (langage de programmation, composants logiciels). De cette manière, la validation fonctionnelle est réalisée indépendamment des choix d'implémentation. Une même spécification fonctionnelle peut ainsi être implantée sur différentes plates-formes d'exécution en étant traduite en différentes spécifications d'implémentations. On distingue ainsi les modèles indépendants de la plate-forme d'exécution (*PIM : Platform Independant Model*) et les modèles spécifiques à la plate-forme d'exécution (*PSM : Platform Specific Model*). La finalité de la MDA est de partir d'un PIM défini par l'utilisateur et validé par un ensemble d'outils, pour obtenir sa transformation automatique en PSM, à son

tour traduit en code exécutable.

Dans [68], on distingue différents types de transformation : PIM vers PSM (déjà abordé au paragraphe précédent), PIM vers PIM (raffinement fonctionnel), PSM vers PSM (raffinement de modèles d'implémentation) et PSM vers PIM (rétro-ingénierie). Ceux-ci répondent donc à des besoins différents. Les différents types de transformation sont illustrés par la figure 2.5. De plus, on distingue le PIM du PSM sur leur langage qui n'est pas nécessairement le même. Généralement, le raffinement (PIM vers PIM, PSM vers PSM) est réalisé sur des modèles définis dans le même langage. On qualifie alors la transformation d'endogène (même langage) ou d'exogène (langages différents).

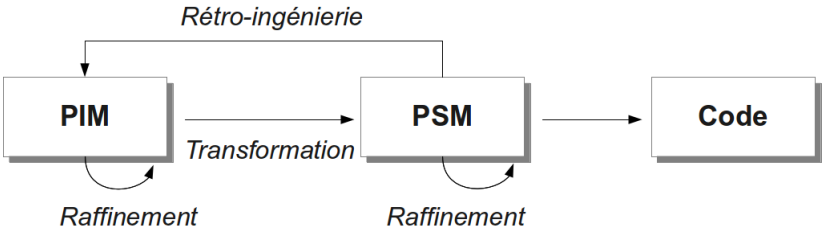



FIGURE 2.5 – Représentation des différents types de transformation

Le raffinement fonctionnel (PIM vers PIM) consiste à remplacer des notations implicites, facilitant la spécification par l'utilisateur, par des notations explicites tout en faisant abstraction des choix technologiques. Par exemple, lors de la spécification d'un modèle UML [73], l'utilisation d'attributs publiques sous-entend généralement que les attributs sont en réalité privés mais rendus accessibles par des méthodes automatiquement générées. Ce raffinement est illustré par la figure 2.6. Un attribut publique $x$ est traduit en attribut privé $x$ accompagné de deux méthodes publiques *getX()* et *setX()*. Cette traduction assure le contrôle des accès et des modifications externes de l'attribut et évite à l'utilisateur d'expliciter ces notations qui peuvent nuire à la lisibilité du modèle.

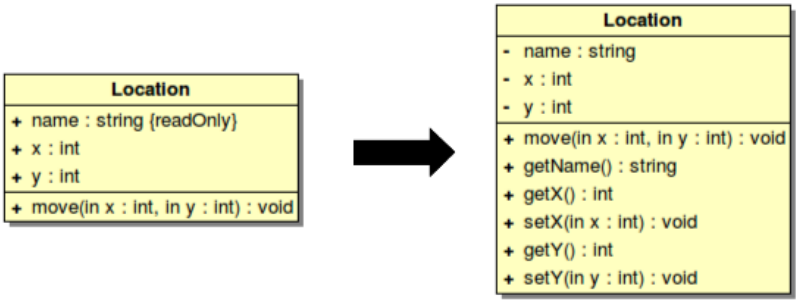



FIGURE 2.6 – Exemple de raffinement d'une classe UML : traduction des attributs publiques en privés

La transformation (PIM vers PSM) consiste à préciser les choix technologiques et d'implantation. Le PSM est alors une spécialisation du PIM vers une plate-forme particulière ou vers un domaine métier particulier. Par exemple, le langage de modélisation UML introduit la notion de

profile [70] permettant d'annoter le modèle pour préciser les composants intergiciels mettant en œuvre chaque élément du modèle. Le modèle UML initial est alors transformé en modèle UML "profilé" pour une plate-forme technologique particulière (*e.g.* CORBA [74], EJB [72]). Il s'agit dans cet exemple d'une transformation endogène, puisque le modèle initial et le modèle transformé sont tous les deux exprimés dans le même langage (UML). Grâce à l'utilisation du même langage de modélisation, la transformation endogène favorise la compatibilité des outils d'analyses à la fois sur le modèle initial et sur le modèle transformé. Cela offre la possibilité d'évaluer les conséquences du changement de niveau d'abstraction sur certaines propriétés. A l'opposé, l'utilisation de langages distincts permet d'obtenir un PSM dans un langage dédié au domaine métier qui exprime plus simplement des concepts spécifiques ou qui bénéficie d'outils d'analyses dédiés. Pour reprendre l'exemple du langage UML, le profile Marte [77] est défini pour les systèmes temps-réel. Pour analyser les propriétés temporelles de ces modèles, il existe des traductions en réseaux de Pétri temporisés [35]. Dans [63], une démarche propose d'utiliser le sous-langage CCSL [64], afin d'introduire les sémantiques d'exécution d'AADL au sein de MARTE, notamment à des fins de simulation.

Afin de mettre place ces mécanismes de transformation, l'OMG introduit la notion de méta-modèle. Un méta-modèle est un modèle définissant un langage : un exemple de méta-modèle représentant le langage UML est donné en figure 2.7. Celui-ci définit les concepts UML (*e.g.* Classe, Attribut, Opération) et leurs relations.

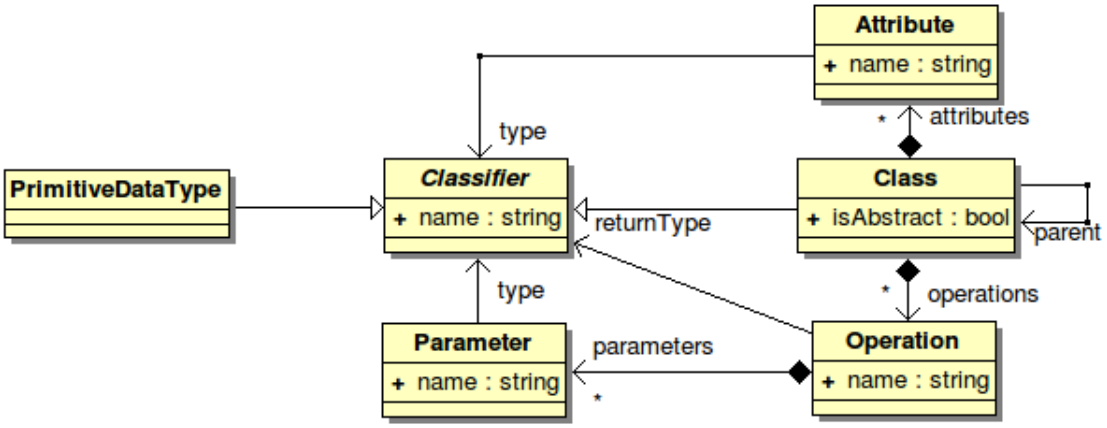



FIGURE 2.7 – Extrait d'un méta-modèle définissant le langage UML

Les concepts communs à l'ensemble des méta-modèles (*i.e.* concepts d'entité et de relation) sont définis au sein du méta-méta-modèle MOF [71] (Meta-Object Facility). Celui-ci est introduit par l'OMG pour faciliter la manipulation et la transformation de modèles. Tout méta-modèle conforme au MOF peut être enregistré et échangé au format XMI [76]. Il s'agit d'un langage dérivé de XML adapté au format objet. D'autres langages de méta-modélisation ont été proposés, notamment autour de l'environnement de développement Eclipse [2, 17, 19].

L'utilisation du MDA apporte ainsi une réponse aux difficultés de conception que sont la séparation des préoccupations et l'évaluation de l'écart entre différents niveaux d'abstraction. D'une

part, la séparation des préoccupations est assurée par un ensemble de raffinements et de spécialisations automatisés. D'autre part, la transformation vers des langages pivots comme UML permet de bénéficier d'outils d'analyse dédiés.

Concernant les difficultés liées à la conception et l'analyse de SETRC, le MDA semble donc être une solution intéressante. En effet, une transformation endogène permettrait d'automatiser la traduction d'une spécification initiale d'un SETRC en une spécification détaillée, dans le même formalisme, et qui intègre les éléments de mise en œuvre. De plus, l'utilisation du même formalisme favoriserait la réutilisation des outils d'analyse sur la spécification détaillée, pour ainsi tenir compte des nouveaux éléments introduits. Cela nécessite ainsi un formalisme de modélisation supportant différents niveaux d'abstractions. Les principaux langages de modélisation de SETRC sont présentés dans la section suivante.

## 2.3 Modélisation architecturale de SETRC

Dans la section précédente nous avons souligné le besoin d'intégrer le code généré au sein du modèle afin de mesurer son impact sur les contraintes du système. Cela peut être réalisé via des transformations endogènes. Cette section présente les principaux langages de modélisation de SETRC et nous traitons de leur capacité à répondre à ce besoin.

Pour chaque langage, nous illustrons la modélisation d'un composant, la spécification de contraintes et nous traitons des éléments permettant de détailler la mise en œuvre, notamment sur les aspects comportementaux.

### 2.3.1 AADL

Le langage AADL [83] est un standard international dédié à la modélisation de SETRC. Il est conçu pour les systèmes critiques de différents domaines (*e.g.* automobile, avionique, spatial). Il peut ainsi spécifier des contraintes de type temporel, ordonnancement, sûreté, sécurité, tolérance aux fautes et déploiement.

**Définition d'un composant** Les types de composants sont prédéfinis. Ils englobent les composants logiciels (*e.g. thread, subprogram, data*) et matériels (*e.g. memory, processor, bus*). Le composant *system* modélise le système global, ou un sous-système, et regroupe des composants logiciels et matériels. Un composant est ainsi défini par sa catégorie comme le montre le listing 2.2 : le modèle définit un composant *th_Sender1* de type *thread*.

Un composant est ainsi défini par une interface qui spécifie un ensemble d'éléments fournies en entrée et en sortie du composant. Dans l'exemple précédent, la tâche *th_Sender1* définit deux éléments de sortie. L'élément *data port* modélise le changement d'état d'une donnée (*e.g.* lue par un capteur). Dans notre exemple, la donnée mise à jour est un flottant codé sur 32 bits (ligne 3). L'élément *event data port* modélise des données de type entier mises en file d'attente. L'exemple précise que la file est limitée à 10 éléments.

```
1  thread th_Sender1
2  features
3    dataout1: out data port Base_Types::Float_32;
4    dataout2: out event data port Base_Types::Integer_32 { Queue_Size => 10; };
5  end th_Sender1;
```

Listing 2.2 – Définition d'une tâche en AADL : spécification de son interface

Il est également possible de préciser la structure interne du composant. Pour une tâche, on va par exemple préciser les fonctions métier exécutées qui vont traiter les données arrivant par les ports d'entrée du composant et produire les données des ports de sortie. La structure interne est spécifiée via le mot-clé *implementation* illustré par le listing 2.3.

```
1  thread th_Sender
2  ...
3  end th_Sender;
4
5  thread implementation th_Sender1.impl
6  calls
7    seq1 : { call1: subprogram compute1;
8             call2: subprogram compute2;
9             call3: subprogram compute3; }
10 connections
11   parameter call1.valueout -> call2.valuein;
12   parameter call2.valueout -> call3.valuein;
13   parameter call3.valueout1 -> dataout1;
14   parameter call3.valueout2 -> dataout2;
15 end th_Sender1.impl;
```

Listing 2.3 – Définition d'une tâche en AADL : spécification du comportement

**Spécification de contraintes** Le langage AADL fournit des propriétés standard pour préciser le comportement attendu d'un composant. Notamment, pour une tâche, on souhaite préciser son type d'activation (*e.g.* périodique, sporadique) ainsi que les contraintes temporelles associées (*e.g.* période, échéance, temps d'exécution). Ces contraintes sont spécifiées via des propriétés prédéfinies comme illustré par le listing 2.4 qui complète l'exemple précédent (lignes 5 à 9).

```
1  thread th_Sender1
2  features
3    dataout1: out data port Base_Types::Float_32;
4    dataout2: out event data port Base_Types::Integer_32 { Queue_Size => 10; };
5  properties
6    Dispatch_Protocol => Periodic;
7    Period => 10 ms;
8    Deadline => 10 ms;
9    Compute_Execution_Time => 1 ms .. 2 ms;
10 end th_Sender1;
```

Listing 2.4 – Définition d'une tâche en AADL : ajout de contraintes temporelles

**Modélisation bas-niveau** Ainsi, les aspects comportementaux sont modélisés à haut-niveau d'abstraction via l'interface du composant et de ses propriétés. Dans l'objectif de rapprocher le

modèle de son implémentation finale, plusieurs éléments permettent d'expliciter les éléments de mise en œuvre. D'une part, il est possible de spécifier à chaque composant un automate comportemental qui explicite les contraintes spécifiées via les propriétés. Pour une tâche, cela va modéliser son comportement selon son type d'activation. D'autre part, les abstractions telles que les ports peuvent être modélisées à plus bas niveau comme des données partagées. Ces dernières sont annotées de propriétés qui doivent indiquer le type de donnée (*e.g.* tableau, struct). Le listing 2.5 illustre comment un *port* peut-être modélisé à un plus bas niveau d'abstraction. Le *port dataout2* de l'exemple précédent est modélisé cette fois-ci en donnée (lignes 1 à 6). Cette modélisation permet de déduire l'occupation mémoire du port selon son implémentation concrète (*e.g.* tableau, liste chainée). De plus, ce raffinement peut s'accompagner de fonctions supplémentaires pouvant être ajoutées dans la liste de fonctions métiers afin de préciser les algorithmes d'insertion/suppression mis en œuvre ainsi que leur temps d'exécution.

```
1  data dataout2_port
2  properties
3    Data_Representation => Array;
4    Base_Type => (classifier (Base_Types::Integer_32));
5    Dimension => (10);
6  end dataout2_port;
7
8  thread th_Sender1
9  features
10 ...
11   dataout2: requires data access dataout2_port;
12 ...
13 end th_Sender1;
```

Listing 2.5 – Modélisation à bas-niveau d'abstraction d'un port en AADL

AADL fournit ainsi les éléments nécessaires pour modéliser un système à différents niveaux de détails. Ce langage est donc tout à fait approprié pour une transformation endogène détaillant la mise en œuvre concrète du système modélisé.

### 2.3.2 UML MARTE

Le profil UML MARTE [77] est un standard de l'OMG pour la modélisation de SETRC. Il succède au profil UML-SPT [75] en ajoutant un certain nombre d'améliorations, notamment la modélisation des plates-formes logicielles et matérielles. MARTE introduit notamment le langage VSL pour modéliser les propriétés non-fonctionnelles des SETRC : latence de bout en bout, taux d'utilisation processeur, consommation d'énergie...

**Définition d'un composant** MARTE définit un ensemble de stéréotypes UML pour modéliser les différents composants d'un SETRC à l'aide de classes. Par exemple, les stéréotypes *SwSchedulableResource*, *MemoryPartition* et *StorageResource* modélisent respectivement les tâches, les espaces d'adressage et les différents types de mémoire. La figure 2.8 donne un exemple de tâche définie avec MARTE : celle-ci spécifie deux ports de sortie *dataout1* et *dataout2*.

L'utilisation de commentaire précise la sémantique de chaque élément. Le commentaire associé à *dataout1* indique le stéréotype *flowPort* qui correspond à la sémantique du *data port* en AADL.

De la même façon, le port *dataout2* est associé à un commentaire pour préciser qu'il s'agit d'une file de message (stéréotype *ClientServerPort*) de taille 10 (propriété *MessageQueueCapacityElements*).

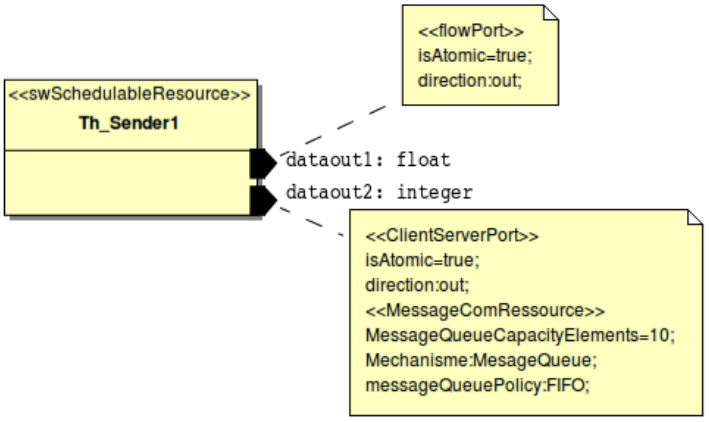



FIGURE 2.8 – Définition d'une tâche en MARTE : spécification de son interface

Le flot d'exécution d'une tâche est modélisable à l'aide d'un diagramme de séquence UML rattaché à la classe. Celui-ci précise la séquence de fonctions appelées par la tâche à chaque activation. Les acteurs du diagramme sont la tâche et les librairies contenant les fonctions appelées. La figure 2.9 donne le diagramme de séquence de la tâche précédente. Le diagramme précise les fonctions appelées par la tâche *th_Sender*. Dans cet exemple, les fonctions sont définies dans la librairie *library*.

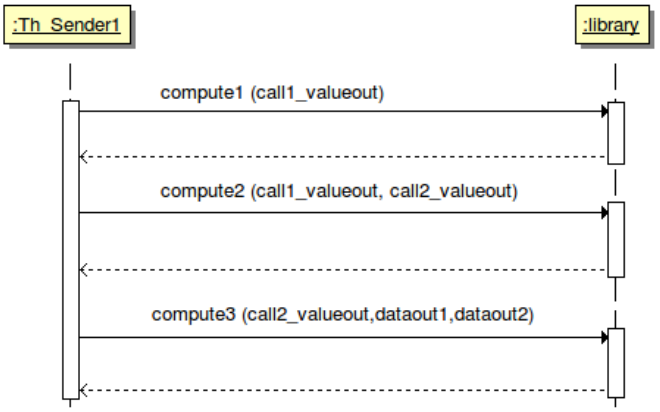



FIGURE 2.9 – Définition d'une tâche en MARTE : spécification du comportement

**Spécification de contraintes** La modélisation de contraintes, notamment temporelles, se fait via des propriétés standard ou définies par l'utilisateur. Ces propriétés sont initialisées via le diagramme d'instance UML. Les instances des classes modélisant les tâches sont spécifiées sur ce diagramme et leurs propriétés sont initialisées pour définir les différentes contraintes. Sur l'exemple de la figure 2.10, la classe *th_Sender1* est instanciée et la propriété *arrival* est initialisée pour préciser le type d'activation périodique et la période associée.

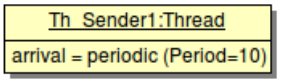



FIGURE 2.10 – Définition d'une tâche en MARTE : ajout de contraintes temporelles

**Modélisation bas-niveau** UML-MARTE fournit le stéréotype *DataType* pour spécifier des types de données à l'aide de classes. Les structures concrètes mises en œuvre sont ainsi modélisables en UML-MARTE. Les données protégées en exclusion mutuelle sont modélisées par le stéréotype *MutualExclusionResource*. La figure 2.11 donne quelques exemples de types de données. De droite à gauche : un tableau, une structure contenant deux champs *x* et *y* et la même structure protégée en exclusion mutuelle. Un *port* peut-être donc également modélisé par un type de donnée, éventuellement protégé en exclusion mutuelle. Pour tenir compte de l'impact sur le temps d'exécution des mécanismes sous-jacents, le diagramme de séquence de la tâche peut également être raffiné pour introduire des opérations spécifiques liées à l'insertion/la suppression de messages dans cette donnée modélisant le *port*. Le WCET des opérations est annoté via une propriété spécifique. Cependant, cela ne semble pas adapté au cas des WCET paramétriques (dépendant des paramètres d'entrée de l'opération). Le WCET doit alors être de nouveau spécifié pour chaque opération pour chaque nouveau jeu de tâches.

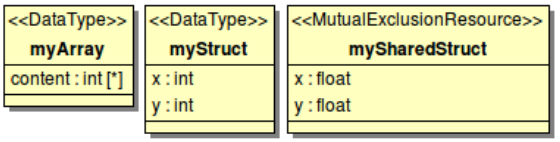



FIGURE 2.11 – Modélisation des types de données en MARTE

UML-MARTE semble fournir les éléments nécessaires pour modéliser à différents niveaux d'abstractions, et semble également approprié pour pour une transformation endogène introduisant les éléments de mise en œuvre. Néanmoins, la modélisation utilisée (*e.g.* annotations, diagrammes de séquence) est assez complexe du fait que MARTE soit une surcouche à un langage généraliste comme UML qui utilise différentes vues pour modéliser un même système (diagramme de classe, de séquence, d'objets, d'activité...).

### 2.3.3 SysML

SysML [33] est une extension du langage UML normalisée par l'OMG. C'est l'un des principaux langages de modélisation de systèmes embarqués avec AADL et MARTE. Contrairement aux deux autres, celui-ci se focalise essentiellement sur l'architecture logicielle et matérielle et n'aborde pas les aspects liés à l'exécution ni aux aspects temporels. Il regroupe les informations permettant de modéliser un système et de simuler son comportement. Les résultats de simulation sont comparés aux exigences afin de valider ou non l'architecture proposée. SysML introduit notamment le *diagramme d'exigences* pour définir les exigences de manière plus ou moins formelle et de les rattacher aux éléments du modèle.

**Définition d'un composant** SysML modélise les aspects logiciels mais aussi matériels du système à l'aide d'un *diagramme de blocs*. Celui-ci représente l'organisation interne du système par un ensemble de composants (logiciels, matériels, ou abstraits) définis par des classes stéréotypées *block*. Les blocs sont connectés entre-eux via des ports qui modélisent notamment les flux de données. Contrairement aux langages précédents, celui-ci modélise le système à un haut-niveau d'abstraction. Il ne précise pas par exemple la répartition des blocs fonctionnels sur un ensemble de tâches. La figure 2.13 illustre la définition d'un bloc SysML. Celui-ci est constitué d'une opération *Compute* ainsi que de trois données *DataIn*, *DataOut* et *InUse*.

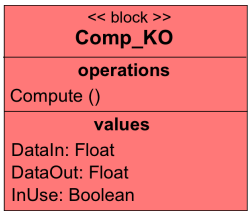



FIGURE 2.12 – Définition d'un bloc en SysML

**Spécification de contraintes** Les contraintes sont formalisées de différentes manières. D'une part, le *diagramme paramétrique* exprime notamment les lois mathématiques reliant les entrées aux sorties des composants. Ce type de modélisation débouche ensuite sur de la simulation qui vérifie le respect de ces contraintes. D'autre part, une classe stéréotypée *block* peut définir, en plus de ses attributs et opérations, un ensemble de contraintes. La figure 2.13 donne un exemple de contrainte fonctionnelle en SysML.

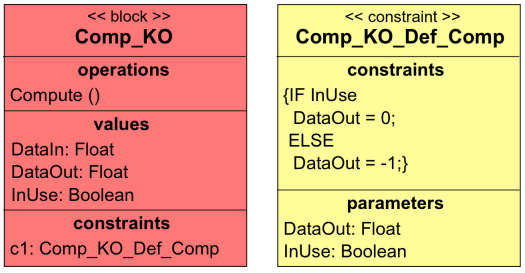



FIGURE 2.13 – Définition d'un bloc en SysML : ajout de contraintes

**Modélisation bas-niveau** Tout comme UML-MARTE, SysML utilise les classes UML pour définir de concepts plus ou moins abstraits de l'architecture modélisée. Une architecture peut être raffinée notamment en détaillant les diagrammes de blocs : une classe stéréotypée *dataType* peut ainsi modéliser les types de données mis en œuvre. De plus, un bloc peut être décomposé en un ensemble de compartiments, notamment via une propriété de type *Part* qui permet de référencer un sous-bloc faisant partie intégrante du bloc principal. Les diagrammes d'activité, modélisant les aspects comportementaux, peuvent eux aussi être raffiné pour référencer les éléments spécifiques introduits précédemment.

Malgré la capacité d'éclater l'organisation du système en un ensemble de blocs et de sous-blocs, SysML reste à un haut-niveau d'abstraction par rapport aux deux langages précédents. D'une part,

celui-ci ne précise pas comment les blocs fonctionnels sont répartis ni exécutés (pas de spécification des tâches ni des contraintes temporelles). Son utilisation reste ainsi assez limitée pour modéliser précisément l'implémentation d'un SETRC.

Nous avons présenté les principaux langages de modélisation architecturale pour SETRC. Ces langages sont illustrés dans [31] qui dresse un état de l'art sur le développement des systèmes embarqués. La fiabilité du futur système dépend de la précision des informations fournies par le modèle. Un processus de génération de code pour SETRC doit s'appuyer sur un langage capable de modéliser le système aussi bien à un niveau conceptuel qu'à un niveau d'implémentation afin d'évaluer l'impact de la génération de code sur les performances du système. AADL et MARTE semblent s'adapter à différents niveaux d'abstractions tandis que SysML reste à un niveau d'abstraction assez élevé. Cependant, contrairement à AADL, MARTE ne dispose pas d'une spécification textuelle et nécessite l'utilisation de plusieurs vues.

Nous avons souligné que l'utilisation d'un langage de modélisation architecturale semble une solution pour prendre en compte l'impact de la génération de code en intégrant les détails d'implémentation au sein du modèle. Cependant, il est nécessaire d'intégrer ces éléments de manière automatique en traduisant le modèle en un second modèle. La section suivante décrit les moyens pour automatiser cette traduction.

## 2.4 Techniques de transformation de modèle

Pour tenir compte de l'impact des éléments de mise en œuvre sur les propriétés du système modélisé, le MDA est une solution pour automatiser l'intégration de ces éléments au sein du modèle. Différentes techniques existent pour réaliser des transformations de modèle à modèle. Un grand nombre d'études ont déjà été réalisées sur ce domaine, notamment [24, 67, 89]. Ces études comparent les différentes techniques sur des critères tels que l'adaptabilité, la modularité, le nombre de modèles sources/cibles (*i.e* N-to-1, 1-to-N, N-to-N), les opérations utilisées (*e.g.* création, mise à jour, suppression). Nous passons en revue les principales techniques existantes dans les sections suivantes : transformations impératives, orientées graphes, relationnelles et hybrides.

### 2.4.1 Impérative

Les approches impératives consistent à coder "soi-même" la transformation à l'aide le plus souvent d'un langage de programmation généraliste et en s'appuyant sur un framework particulier. Ce dernier fournit des types de base (modélisant les règles de transformation) qui doivent être étendus pour spécialiser le framework. La logique de parcours des éléments et de transformation sont laissées à la charge de l'utilisateur.

Plusieurs frameworks de transformation ont été implémentés en langage Java comme SiTra [7] ou Jamda [3]. SiTra définit une API minimaliste illustrée par le listing 2.6. Elle est constituée de deux interfaces : *Rule* et *Transformer*. La première est implémentée pour chaque règle de transformation que l'on souhaite définir. Chaque classe implémentant l'interface *Rule* doit préciser les

types génériques *S* et *T* qui correspondent respectivement au type des éléments d'entrée et au type des éléments de sortie. Chaque règle réalise donc une transformation de type 1-vers-1. Le type *Rule* est constitué de plusieurs méthodes : *check* indique si l'élément d'entrée de type *S* remplit les conditions pour être transformé en élément de type *T*. Si c'est le cas, la méthode *build* est appelée pour réaliser la transformation et est suivie d'un appel à la méthode *setProperties* qui finalise l'initialisation de l'élément *T* produit.

```
1  interface Rule<S,T> {
2      boolean check(S source);
3      T build(S source, Transformer t);
4      void setProperties(T target, S source, Transformer t);
5  }
6  interface Transformer<S,T> {
7      Object transform(Object source);
8      List<Object> transformAll(List<Object> sourceObjects);
9      <S,T> T transform(Class<Rule<S,T>> ruleType, S source);
10     <S,T> List<T> transformAll(Class<Rule<S,T>> ruleType,List<S> source);
11 }
```

Listing 2.6 – Framework de transformation SiTra

L'interface *Transformer* gère la logique d'application des règles précédemment définies. En particulier, la méthode *transform* applique la ou les règles correspondantes au type de l'objet fourni en paramètre. Un objet *Transformer* est donné en paramètre des méthodes de l'interface *Rule* pour pouvoir appeler récursivement plusieurs règles de transformation sur les éléments agrégés à l'élément source.

**Démarche** L'utilisation d'une approche impérative comme SiTra ne nécessite pas l'apprentissage d'un nouveau langage et est simple à prendre en main. D'autres approches impératives utilisent des langages impératifs spécialisés qui facilitent l'écriture des transformations. C'est le cas notamment des langages comme Xion [69], MTL [93] ou encore Kermeta [50] qui est issu de ces deux langages. Certains de ces langages intègrent en particulier des expressions OCL pour faciliter le filtrage des éléments. Le listing 2.7 illustre le langage Xion sur la classe principale *Transformation*. Celle-ci transforme l'ensemble des classes persistantes du langage source sous forme de tables. On remarque l'utilisation d'expressions OCL (lignes 2 à 5) pour sélectionner les classes persistantes pour lesquelles la fonction *transformPersistentClass* est appliquée.

```
1  public Void Transformation::Run (Class class, Table table) {
2     MM::Class.allInstances()
3        ->select(parent == null)
4        ->select(c : c.is_persistent || c.hasPersistentChildren())
5        ->collect(c : transformPersistentClass(c));
6
7     this.classTransformation->collect(t : t.transform());
8  }
9  private Void Transformation::transformPersistentClass (Class class) {
10    MM::ClassTransformation t =
11       new MM::ClassTransformation(class, new MM::Table(class.name));
12
13    this.addclassTransformation(t);
14    this.addremaining(t);
15 }
```

Listing 2.7 – Langage de transformation Xion : écriture d'une règle

L'approche impérative semble ainsi assez simple à prendre en main pour un programmeur qui souhaite s'initier aux transformations de modèle. En effet, certaines de ces approches utilisent des frameworks qui s'intègrent dans un langage de programmation généraliste. D'autres fournissent des DSL qui facilitent l'écriture tout en restant dans un contexte impératif.

**Limitations** Ce type d'approche à l'aide d'un langage généraliste nécessite cependant de filtrer les éléments de manière explicite à l'aide de boucles, rendant l'écriture assez fastidieuse. Malgré l'intégration des expressions OCL dans certains langages, l'écriture impérative nuit à la lisibilité. Par ailleurs, la logique d'application des règles de transformation est souvent modélisée dans un format différent des règles et inclut des mécanismes de sélection non-déterministes.

### 2.4.2 Orientée Graphes

Cette catégorie de transformation se base sur la théorie des graphes et la représentation graphique des règles de transformation. Chaque règle est définie par un *LHS* (Left Hand Side) et un *RHS* (Right Hand Side). Le *LHS* et le *RHS* représentent respectivement sous forme de graphe les éléments sources et les éléments cibles de la transformation. Par exemple, pour une règle s'appliquant sur un élément source *A* associé à un élément *B*, le *LHS* va être modélisé comme un graphe constitué d'un sommet *A* relié à un sommet *B*. Chaque sous-graphe (du modèle source) correspondant au *LHS* est alors remplacé par un sous-graphe modélisé par le *RHS*. Le *LHS* est parfois annoté d'une condition pour restreindre le champ d'application de la règle. L'initialisation des éléments cibles est modélisée via des expressions logiques mettant en relation les attributs des éléments sources et les attributs des éléments cibles.

**Démarche** Cette catégorie inclut des langages tels que GReAT [11], VIATRA [23] et UMLX [95]. Ces langages sont assez intuitifs par l'utilisation d'une représentation graphique. Par exemple, en langage GReAT une règle est constituée du *LHS*, du *RHS*, de ports d'entrée et de sortie, une garde, un ensemble d'actions. L'utilisateur spécifie les actions à réaliser (*e.g.* CreateNew, Bind, Delete) en reliant les sommets du *LHS* aux sommets du *RHS*. Les éléments (*i.e.* sommets) sources ou cibles peuvent être passés en entrée d'une seconde règle de transformation via les ports de sortie de la première. Dans ce cas, ces éléments sont reliés au port correspondant. La figure 2.14 montre un exemple de règle de transformation modélisée en GReAT. Celle-ci dispose de deux ports d'entrée *IR* et *IP*. Le *LHS* est constitué des éléments *Parent* et *Child*. L'objet *Parent* est fourni en entrée de la transformation par le port *IP*. Cet objet a donc été potentiellement produit lors d'une règle précédente. L'objet *Child* indique que le *LHS* se réfère à un objet existant de type *Child* rattaché à l'objet *Parent*. L'objet *Child* est transformé en objet *Actor*. Ce dernier est associé à un objet *Root* qui fait partie du *RHS* mais qui a été crée lors d'une règle précédente (obtenu par le port *IR*).

**Limitations** La définition d'une règle peut rapidement devenir assez lourde et peu lisible si elle fait intervenir un grand nombre d'éléments. De plus, pour ce type d'approche, la logique de sélection et d'application des règles doit être le plus souvent spécifiée par l'utilisateur à l'aide d'un langage particulier. Comme pour l'approche impérative, la difficulté est alors de déterminer à quel moment chaque règle doit être exécutée.

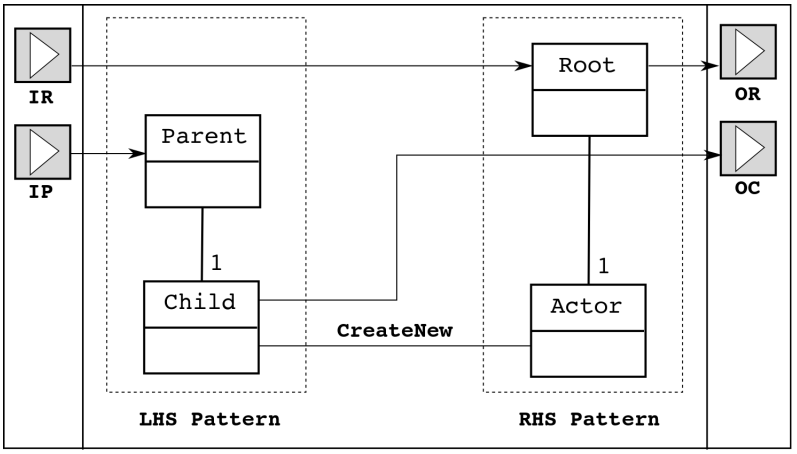



FIGURE 2.14 – Langage de transformation GReAT : modélisation d'une règle

### 2.4.3 Relationnelle

L'approche relationnelle est similaire à la précédente par la définition déclarative des règles de transformation. Contrairement à l'approche précédente, la logique de sélection des règles de transformation est le plus souvent implicite. Cette approche s'appuie principalement sur les relations mathématiques. Chaque règle est ainsi définie par la déclaration des éléments sources et des éléments cibles ainsi que d'éventuelles contraintes. Comme pour l'approche précédente et contrairement à l'approche impérative, les éléments cibles sont crées implicitement. Les langages rentrant dans cette catégorie se distinguent notamment par l'utilisation de relations bi-directionnelles ou unidirectionnelles. Dans le premier cas, la règle de transformation peut être interprétée dans les deux sens.

**Démarche** Des langages comme Tefkat [61] ou QVT [78] rentrent dans cette catégorie. En langage Tefkat, les éléments sources et cibles d'une règle sont spécifiés respectivement dans les clauses *FORALL* et *MAKE*. Cela est illustré par le listing 2.8. La règle *ClassAndTable* transforme chaque classe persistante (ligne 2) en une table (ligne 6). La table créée est initialisée avec le nom de la classe en définissant une variable *N* qui met en correspondance les attributs *name* des deux éléments. Les accolades permettent de définir des conditions de sélection et de mettre en correspondance les attributs des éléments.

```
1  RULE ClassAndTable(C, T)
2    FORALL Class C {
3       is_persistent: true;
4       name: N;
5    } MAKE Table T {
6       name: N;
7    } LINKING ClsToTbl WITH class = C, table = T;
8  ...
9  CLASS ClsToTbl {
10   Class class;
11   Table table;
12 };
```

Listing 2.8 – Langage de transformation Tefkat : écriture d'une règle

Pour améliorer la lisibilité des règles, les conditions de sélection complexes peuvent être définies dans des expressions annexes appelées *patterns* comme le montre le listing 2.9. Sur cet exemple, le *pattern* sélectionne l'ensemble des éléments de type *Class* pour lesquels l'attribut *attrs* contient un élément correspondant aux critères demandés.

```
1 PATTERN ClassHasSimpleAttr(Class, Attr, Name, IsKey)
2   FORALL Class Class {
3     attrs: Attribute Attr {
4       type: PrimitiveDataType _PT;
5       name: Name;
6       is_primary: IsKey;
7     };
8   };
```

Listing 2.9 – Langage de transformation Tefkat : définition de patterns

Par rapport aux approches impératives et orientées graphes, les approches relationnelles offrent ainsi plus de lisibilité en rendant implicite la logique d'exécution des transformations. La transformation est ainsi plus concise en se focalisant sur les relations entre les éléments et en faisant abstraction de la logique de parcours du modèle.

**Limitations** La principale limitation de ce type d'approche est le manque de souplesse dans l'écriture de la règle de transformation : l'absence de code impératif pouvant conduire à écrire des expressions logiques complexes.

### 2.4.4 Hybride

Les approches hybrides regroupent les langages qui combinent des techniques de plusieurs paradigmes de transformation. Dans [36] est souligné que la mise à jour d'un modèle cible nécessitant parfois des aspects procéduraux, les approches déclaratives (*i.e.* relationnelles), plus pratiques par le parcours implicite du modèle, n'offrent cependant pas ces aspects. Les approches hybrides sont ainsi proposées pour répondre à ce besoin. Elles offrent plus de flexibilité et sont souvent utilisées dans la pratique.

**Démarche** Les langages hybrides formalisent la transformation en combinant les notations des paradigmes hérités. Par exemple, le langage Tefkat, précédemment cité dans la catégorie des approches relationnelles, peut également définir des règles à l'aide de constructions impératives. Le langage ATL [51] fait également partie de cette catégorie. En effet, d'une part on distingue les règles ATL par l'utilisation ou non de code impératif dans une clause particulière *do* qui succède aux clauses *from* et *to* correspondant aux clauses *FORALL* et *MAKE* en langage Tefkat. D'autre part, les règles sont différenciées selon si elles sont implicitement exécutées dès lors qu'un élément d'entrée est compatible avec celles-ci, ou si elles doivent être explicitement appelées dans du code impératif par l'utilisateur. Le listing 2.10 illustre ces différents types de règle ATL : relationnelle, impérative et hybride. La règle impérative se caractérise par la clause *do*. L'utilisation de paramètres ($x$ et $y$) implique que la règle est appelée explicitement au sein d'une autre règle (ligne 17). La troisième règle est une combinaison des deux précédentes.

```
1  rule FullyRelational {
2    from
3      x : MM1!X (x.value > 0)
4    to
5      y : MM2!Y (value <- x.value)
6  }
7  rule FullyImperative (x : MM1!X, y : MM2!Y) {
8    do { -- initialize (previously created) variable y
9      if (x.value > 0) {
10       y.value <- x.value
11     }
12   }
13 }
14 rule HybridRule {
15   from   x : MM1!X (x.value > 0)
16   to     y : MM2!Y
17   do { y.value <- thisModule.FullyImperative(x,y); }
18 }
```

Listing 2.10 – Langage de transformation ATL : différents types de règle

En combinant les différents paradigmes de transformation, les approches hybrides bénéficient par conséquent des avantages de chacune. Les relations entre les éléments peuvent ainsi être exprimées de différentes façon. L'utilisateur peut choisir la manière la plus appropriée pour exprimer chaque relation.

**Limitations** Les approches hybrides héritent des restrictions des paradigmes utilisés. Cependant, la finalité de ces langages étant d'offrir plus de flexibilité en combinant différentes approches, ils sont généralement peu limités.

Nous avons passé en revue les principales techniques de transformation de modèle qui sont décrites dans la littérature. Ces techniques permettent de réaliser aussi bien des transformations endogènes qu'exogènes. Dans l'objectif d'obtenir un modèle transformé proche de l'implémentation réelle, ce dernier doit être analysé par des outils spécifiques au domaine pour déterminer si la transformation réalisée assure le respect des contraintes fixées en amont. Pour cela, les transformations doivent être décidées au sein d'un processus incrémental qui évalue le modèle transformé et décide en conséquence de faire progresser le raffinement dans une direction particulière. La prochaine section aborde les différentes solutions techniques pour mettre en place ce type de processus.

## 2.5 Orchestration de transformations

Nous avons introduit, dans la section précédente, les principaux paradigmes de transformation utilisés pour réaliser le raffinement de modèles. Dans le contexte des SETRC, l'objectif du raffinement est d'obtenir un modèle d'implémentation décrivant la mise en œuvre concrète de l'architecture modélisée. Le raffinement doit cependant être évalué pour déterminer si celui-ci est adapté au modèle source considéré. En effet, on souhaite appliquer des chaînes de raffinements différentes

pour des modèles ayant des contraintes différentes. Par conséquent, il est nécessaire d'utiliser un outil permettant de sélectionner et d'appliquer des raffinements adaptés aux contraintes du modèle source. Ces différents outils sont présentés dans les sous-sections suivantes.

### 2.5.1 Wires*

Wires* [82] est un langage graphique pour l'orchestration de transformations ATL. La spécification modulaire favorise la réutilisation de blocs de transformation. Ainsi, on distingue deux types d'étapes de transformation : atomique et composite. Une étape atomique représente une transformation ATL basique tandis qu'une étape composite est une chaîne de transformation réutilisable comme boite noire dans une autre chaîne de transformation. Wires* permet de choisir les transformations à réaliser selon les propriétés du modèle. Une étape de transformation peut ainsi être constituée de deux transformations alternatives et d'une expression OCL qui détermine quelle alternative doit être réalisée.

**Démarche** La figure 2.15 est un exemple de processus modélisé en langage Wires*. Le processus transforme le modèle d'entrée *m1* en un modèle *m2*. La transformation est soit réalisée par la transformation *t1* soit par la transformation *t2* selon la valeur booléenne retournée par l'expression OCL encapsulée dans l'objet *q*. La phase décisionnelle est donc modélisée par trois connexions depuis le modèle source : les deux transformations alternatives ainsi que l'expression OCL qui va déterminer quelle transformation sera réalisée. Dans cet exemple, la transformation *t1* (resp. *t2*) est exécutée si l'expression *q1>3* renvoie *true* (resp. *false*).

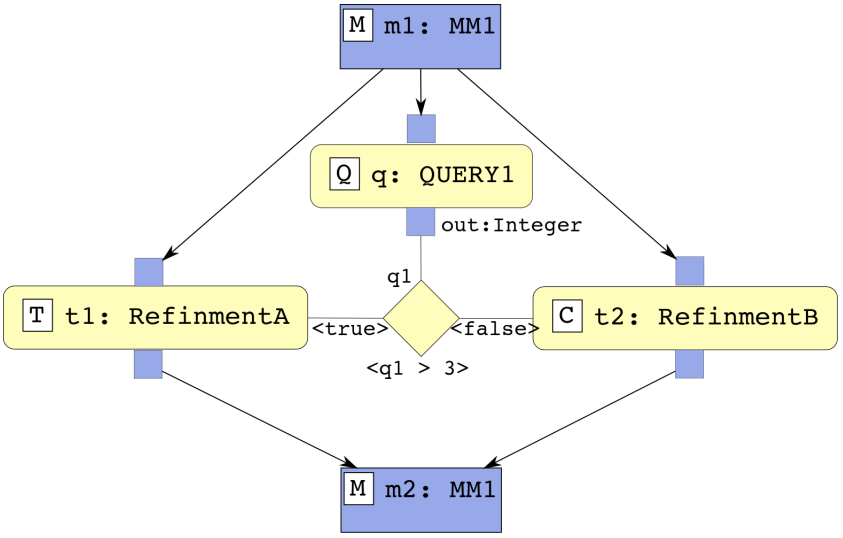



FIGURE 2.15 – Langage Wires* : modélisation d'un processus de transformation

La transformation *t1* est atomique et est par conséquent implémentée en langage ATL. Au contraire, la transformation *t2* est composite, elle est donc décrite dans un sous-processus illustré par la figure 2.16. Celui-ci est constitué de deux étapes successives *T1* et *T2*. Les étapes composites favorisent ainsi la réutilisation de chaînes de transformation pour définir des processus de transformation qui partagent une partie de leur raffinement. Le langage Wires* permet également de

modéliser la parallélisation de raffinements indépendants, les boucles ainsi que la conservation de modèles intermédiaires. Les boucles consistent à appliquer une même transformation endogène sur le modèle courant jusqu'à ce que l'expression OCL soit valide.

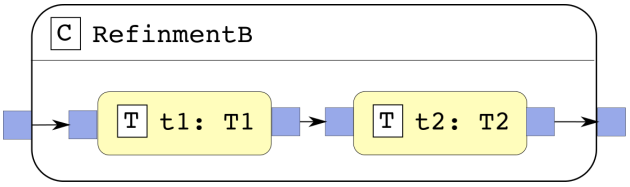



FIGURE 2.16 – Langage Wires* : modélisation d'une étape composite

Ainsi, Wires* définit des processus de transformation qui supportent une certaine adaptation selon les propriétés du modèle. Il favorise également la réutilisation de transformations à l'aide des étapes *composite*. Cet outil semble donc répondre en partie aux besoins des SETRC en terme de facilité de mise en œuvre et d'adaptation du code généré.

**Limitations** Wires* montre une limitation importante : les phases décisionnelles sont uniquement implémentées en OCL. Pour un SETRC, il est souvent nécessaire de faire appel à un outil dédié, par exemple pour faire une simulation d'ordonnancement, et OCL n'est pas adapté pour ce type d'analyse.

### 2.5.2 UniTI

UniTI [91] est une technologie qui souhaite répondre au manque d'abstraction des langages de transformation connus. Il s'inspire notamment des principes de la programmation orientée composant. Notamment, il met en avant le principe de la boite noire séparant le comportement "publique" d'une transformation de son implémentation concrète. En effet, le découplage de la logique de transformation de son implémentation concrète dans tel ou tel langage facilite la maintenance et la compréhension du processus global. Par conséquent, la spécification externe de la transformation (modèles d'entrées/sorties) est modélisée indépendamment du langage dans lequel elle est implémentée.

**Démarche** Ce type de modélisation peut ainsi modéliser un processus de transformation constituée d'étapes de transformations écrites dans différents langages : ATL, Java, MTF... UniTI introduit alors le méta-modèle UTR (*Unified Transformation Representation*) visant à modéliser ce type de processus. Un extrait du méta-modèle UTR est donné en figure 2.17.

A l'instar de Wires*, UniTI distingue les transformations atomiques et composites. Une étape de transformation est modélisée par le type *TFSpecification* qui se décline en *AtomicTFSpecification* et *CompositeTFSpecification*. UniTI favorise ainsi la réutilisation de chaînes de transformation au sein d'étapes composite. UniTI intègre également les contraintes OCL pour vérifier la conformité architecturale du modèle vis-à- vis d'une étape transformation.

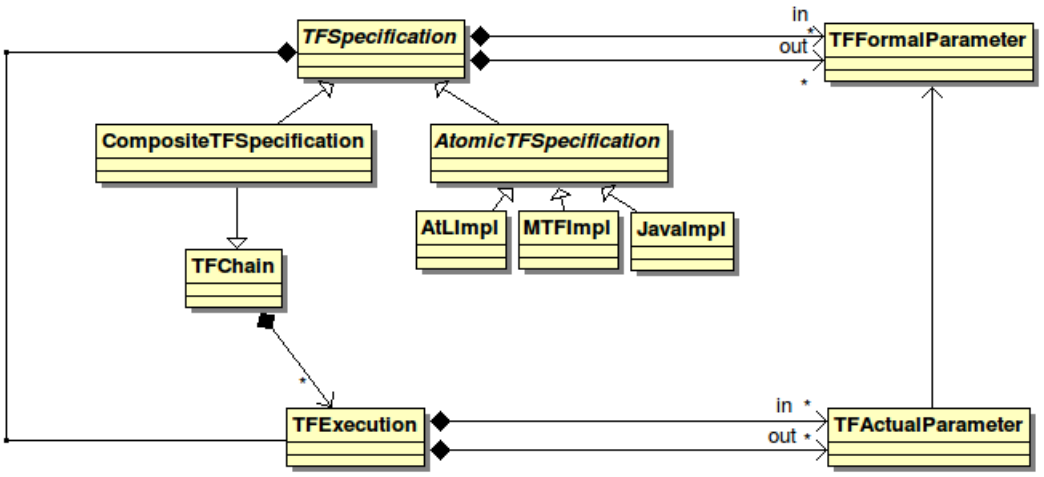



FIGURE 2.17 – Langage UniTI : extrait du méta-modèle UTR

**Limitations** Les analyses à l'aide d'OCL ne semblent pas permettre de spécifier des transformations alternatives. Les processus de transformation modélisés avec UniTI sont donc assez restrictifs et peu adaptés aux SETRC : notamment concernant l'adaptation de la transformation selon les propriétés du modèle et l'impossibilité d'utiliser des outils d'analyse dédiés. D'autres approches similaires ont été proposées [6, 65] pour chaîner les transformations mais limitent également le processus à une séquence de transformation exécutées sans condition.

### 2.5.3 MT-Flow

MT-Flow [58] est un environnement interactif pour la conception de lignes de produit à l'aide du langage de spécification Executable UML [66] et de son Object Action Language [66]. Il utilise un modèle de *workflow* guidant l'utilisateur à travers des étapes de configuration du système modélisé. Ces phases de configuration consistent à sélectionner les éléments que l'utilisateur souhaite faire figurer au sein du système. A chaque étape de configuration, le modèle est transformé et son changement d'état est directement visible au sein de l'environnement graphique de MT-Flow.

**Démarche** Le méta-modèle de *workflow* utilisé dans MT-Flow est partiellement donné en figure 2.18. Un *workflow* est constitué d'un ensemble d'étapes de types *ConfStep*. Chacune est implémentée dans un objet *Template* qui décrit la transformation d'un système modélisé en XMI. Elle s'appuie également sur des objets de type *Function* qui sont des entités permettant de récupérer le contenu du modèle. Ces objets sont écrits dans un langage propre à MT-Flow. A chaque étape peut succéder un ensemble de sous-étapes par l'intermédiaire d'objets de type *Transition*. Chaque transition est associée à un ensemble de fonctions d'évaluation par l'intermédiaire d'objets *Control-Connector*. Ainsi, une transition n'est franchie seulement si l'ensemble des fonctions d'évaluation sont évaluées à *true*.

**Limitations** Cette approche fournit des étapes d'analyses qui conditionnent partiellement l'exécution du processus de transformation. D'une part, celles-ci sont écrites dans un langage dédié et

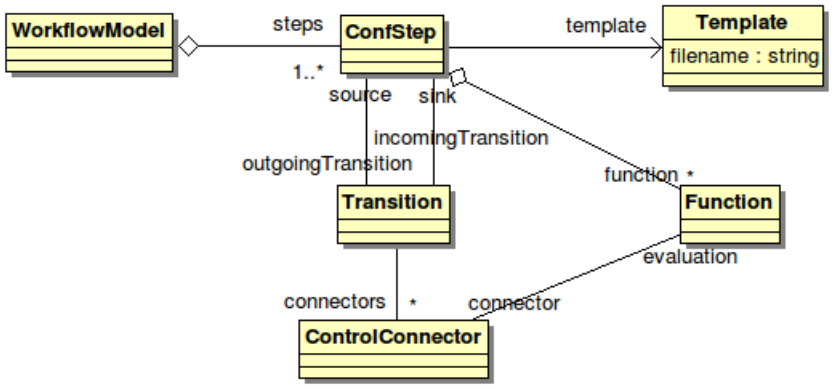



FIGURE 2.18 – Langage MT-Flow : extrait du méta-modèle

ne semblent pas pouvoir appeler des outils externes. D'autre part, ces étapes d'analyses sont utilisées pour limiter la progression d'une étape de transformation mais ne sont pas exploitées pour proposer l'utilisation de transformation alternatives.

Nous avons présenté différents outils permettant de mettre en place des transformations incrémentales. Ceux-ci ne sont pas spécifiques au domaine des SETRC et ne répondent pas réellement au besoin de maîtriser le coût du code généré. En effet, le changement de stratégie de génération n'est pas ou peu supporté. Par conséquent, ceux-ci ne semblent pas adaptés à la génération de code pour SETRC.

## 2.6 Conclusion

Ce chapitre a abordé l'état de l'art sur différents aspects liés à la maitrise de la génération de code pour SETRC et au coût du code généré. Nous avons introduits les différents frameworks de génération de code spécifiques. Ceux-ci ne se limitent pas à l'activité de générer du code mais font intervenir des analyses afin de valider l'architecture modélisée. Les frameworks s'appuyant sur des modèles de haut-niveau d'abstraction facilitent la spécification mais soulèvent des difficultés à maîtriser l'impact de la génération de code sur les propriétés validées en amont. Nous nous sommes alors intéressés aux éléments de mise en œuvre impactant ces propriétés et à la manière de les prendre en compte par l'utilisation de la MDA. Une telle approche nécessitant un langage de modélisation supportant le raffinement d'un modèle conceptuel en modèle d'implémentation, nous avons étudié les principaux langages de modélisation de SETRC. Ensuite, nous avons traité les différents paradigmes de transformation afin d'automatiser le raffinement du modèle. En particulier, les approches hybrides sont assez flexibles en combinant différentes techniques, et semblent appropriées pour réaliser ce type de processus. Nous avons alors passé en revue les différents frameworks de génération généralistes basés sur des étapes de transformation. Ceux-ci ne sont pas adaptés pour modifier la stratégie de génération en fonction du coût engendré par le code généré. Le chapitre suivant soulève la problématique de la maîtrise du code généré.

# Chapitre 3

# Problématique

Pour garantir les exigences d'un système embarqué temps-réel critique (SETRC), le code exécutable doit être *vérifié* et *adapté* aux contraintes du système. La vérification repose sur la capacité à évaluer les performances (*e.g.* temps de réponse) du SETRC et à déterminer si les contraintes du système sont respectées ou non (*e.g.* dates d'échéances). L'écart entre le système tel qu'il a été initialement modélisé et vérifié, et celui tel qu'il sera effectivement exécuté peut conduire à une mauvaise estimation des performances : cela s'explique par un passage à un niveau de détail plus important et à l'apparition d'éléments non modélisés. En effet, l'implémentation d'une abstraction induit un coût supplémentaire qu'il faut maîtriser. Le code exécutable produit manuellement ou automatiquement doit être *adapté* aux contraintes du système en reposant sur des choix d'implémentation qui garantissent le respect de celles-ci.

Afin d'obtenir un code exécutable *vérifié* et *adapté*, celui-ci doit être produit de façon automatique et contrôlée. D'une part, l'utilisation d'un générateur de code assure la maîtrise et la cohérence de l'implémentation. En complément, le code doit être généré de façon contrôlée : le générateur doit tenir compte du coût de l'implémentation choisie et de son impact sur les performances du SETRC. Pour cela, il doit fournir des implémentations alternatives dont il doit être en mesure d'évaluer le coût afin de garantir le respect des contraintes du SETRC. Par ailleurs, cela nécessite de conserver les propriétés analysables tout au long du processus.

Il ne faut pas négliger le coût que représente la mise en place d'un tel générateur, notamment en terme d'évolution, et des difficultés de certification. En particulier, si plusieurs plates-formes d'exécution sont visées par un même générateur, la réduction du coût par une mise en commun des stratégies de génération ne doit pas remettre en cause les propriétés du code exécutable. Par exemple, l'utilisation de composants intergiciels génériques réutilisables facilite la génération de code mais induit un surcoût supplémentaire par rapport à une implémentation dédiée : un algorithme générique prenant en compte différents cas d'utilisation sera moins efficace qu'un algorithme dédié à un unique cas. Le choix d'une implémentation générique n'est alors pas adapté.

Nous abordons ces problématiques dans les sections suivantes.

## 3.1 Rupture d'analyse entre système modélisé et système généré

Un processus classique de développement de SETRC est constitué de trois grandes étapes : la modélisation du système, sa validation par un ensemble d'analyses effectuées sur le modèle, puis la génération de code si la validation réussie.

La validation à la conception facilite la détection de problèmes tôt dans le processus de génération et évite ainsi des modifications du code exécutable, une fois généré [87]. En supposant que l'ensemble des analyses nécessaires à la validation sont réalisées lors de la conception, le code généré peut remettre en cause ces analyses [12] : en particulier, l'introduction de composants intergiciels lors de la génération de code impacte les propriétés du système modélisé telles que le temps de réponse. Le modèle initial n'incluant pas ces composants, la validation ne peut alors tenir compte de leur impact sur les performances. Par conséquent, la validation en amont ne garantit pas le respect des contraintes pour le système final issu de la génération de code.

Il est alors nécessaire de conserver les propriétés analysables tout au long du processus : du modèle initial au code exécutable. De cette manière l'analyse s'effectue tout au long des différentes étapes de génération de code. Cependant, étant donné qu'une même analyse peut intervenir à plusieurs étapes du processus, elle peut soit s'effectuer sur le modèle initial ou bien sur le code final (code exécutable en langage de programmation). Ces entités hétérogènes que sont le modèle et le code exécutable requièrent alors des outils distincts pour effectuer une même analyse. Par exemple, un modèle d'architecture décrit en langage *UML Marte* [77] va être validé par des outils compatibles tels que *MAST* [42] tandis que le code généré sera évalué par des outils d'analyse de code exécutable (analyse statique du code, analyse à l'exécution). Cela soulève la problématique de la cohérence entre les analyses. Puisque plusieurs outils sont nécessaires pour réaliser les mêmes analyses, il faut s'assurer que leurs méthodes d'évaluation sont identiques ou équivalentes pour garantir une démarche de validation cohérente.

Pour obtenir des méthodes d'évaluation équivalentes, le modèle et le code généré doivent donc être homogènes pour être évalués avec les mêmes outils. Dans [13] est proposée une démarche d'analyse indépendante du méta-modèle en proposant de représenter le modèle par une séquence d'opérations de création et d'initialisation. Cependant ce type d'approche se limite à l'analyse structurelle du modèle et à l'analyse du séquencement des opérations. On peut également s'intéresser aux techniques de raffinements de modèle. Notamment, dans le domaine du MDA [79], les transformations de modèle sont des techniques consistant à transformer automatiquement un modèle initial (*PIM : Platform Independant Model*) pour obtenir un modèle raffiné spécifique à la plate-forme d'exécution visée (*PSM : Platform-Specific Model*). Parmi ces techniques, les transformations endogènes [67] produisent un PSM dans le même formalisme que le PIM. Cela donne ainsi la possibilité d'utiliser un même outil d'analyse en début et en fin du processus. La capacité à analyser le code généré et la précision des analyses dépendent alors de l'écart entre le PSM et le code généré. Par exemple, dans [60], on réalise une grande partie de la génération de code exécutable au sein d'une transformation endogène. La phase finale étant l'appel à un *pretty-printer* traduisant le modèle raffiné en code exécutable sans l'ajout de ressources supplémentaires. L'évaluation du PSM donne ainsi la possibilité de déterminer si la stratégie de génération est adaptée aux contraintes du système. Cependant, les processus de transformation permettant d'entrelacer analyses et transformations [82, 58, 43] requièrent une spécification complète des méthodes d'analyse

plutôt que d'utiliser des outils dédiés. Leur utilisation n'est donc pas adaptée au domaine des SETRC. Une alternative est d'accompagner le PSM d'un modèle de prédiction de performances. Dans [87], le processus de transformation est dérivé en une seconde transformation, appelée *transformation couplée*, produisant un modèle d'analyse correspondant. Cette approche nécessite un niveau d'expertise élevé pour déterminer et évaluer les aspects qualitatifs induits par la transformation, la transformation couplée étant écrite manuellement par l'utilisateur. Enfin, l'analyse du PSM nécessite de conserver les propriétés tout au long du processus de transformation : par exemple, en accompagnant la transformation de traces qui indiquent pour chaque élément source les éléments cibles produits[52].

**Conclusion** Plusieurs techniques existent autour du MDA pour obtenir un modèle analysable plus proche de l'implémentation réelle. La transformation endogène favorise la réutilisation d'outils de validation dédiés au domaine (*e.g.* temps-réel embarqué) aux différentes étapes de génération. Cependant, les approches alternant transformations et analyses ne sont pas adaptées au domaine du SETRC. En particulier, elles requièrent une capacité de l'utilisateur à formaliser les méthodes d'analyse. De la même façon, la production d'un modèle d'analyse complémentaire au modèle raffiné proposée dans [87], requiert également un haut niveau d'expertise sur la prédiction des performances du modèle raffiné. Finalement, aucun processus n'est proposé pour maîtriser l'impact du code généré sur les performances du système : notamment, l'absence de *workflow* adapté au domaine et basé sur des raffinements dont le coût est évalué.

## 3.2 Adaptabilité du processus de génération

La problématique précédente souligne le besoin d'analyser le modèle au cours de la génération afin de déterminer si l'implémentation choisie est adaptée aux propriétés du SETRC. En effet, la génération de code doit pouvoir être adaptée en fonction de ces propriétés. Premièrement, le choix de la plate-forme d'exécution visée implique l'utilisation de composants logiciels dédiés : notamment lorsqu'elle introduit des concepts spécifiques qui ne sont pas partagés avec d'autres plates-formes d'exécution. La stratégie de génération va alors dépendre de la plate-forme d'exécution ciblée. Deuxièmement, on peut également fournir des implémentations alternatives pour une même plate-forme. Le coût des mécanismes offerts par la plate-forme, ou les caractéristiques du système (*e*.g. nombre de tâches, jigue) peut influencer le choix de la stratégie. En effet, une implémentation sera appropriée pour certaines configurations et pas pour d'autres.

Développer un tel processus proposant plusieurs stratégies de génération peut être une tâche difficile, en particulier en terme de maintenance. On peut alors s'intéresser à la portabilité d'un générateur de code existant afin de l'adapter à une autre plate-forme d'exécution ou à un autre choix d'implémentation.

Pour illustrer le problème de la portabilité, nous représentons les SETRC comme un empilement de quatre couches, en partant des composants applicatifs à la plate-forme d'exécution :

1. **Composants applicatifs** : modèles fournis par l'utilisateur. Il représentent le système : topologie, interactions entre composants, propriétés temporelles et de dimensionnement...

Les composants s'accompagnent de code implémentant les fonctions métier de l'application à déployer.

2. **Conteneurs pour composants applicatifs** : code généré à partir des modèles de l'utilisateur (couche 1) pour faire le lien avec le code fonctionnel de l'intergiciel (couche 3).
3. **Intergiciel** : composants intergiciels servant de façade à des plates-formes d'exécution spécifiques. Ces composants ont leur propre sémantique d'exécution qui n'est pas nécessairement identique à celle de la plate-forme d'exécution (couche 4). L'intergiciel permet ainsi de s'abstraire des différences architecturales et comportementales.
4. **Plate-forme d'exécution** : plate-forme matérielle et noyau.

Lorsque l'on souhaite fournir plusieurs stratégies de génération, deux solutions peuvent être envisagées pour adapter le générateur existant : la première consiste à assurer la portabilité par un intergiciel générique (couche 3). Celui-ci adapte le code généré (couche 2) à la plate-forme d'exécution (couche 4). Une API générique est alors définie de façon à pouvoir interagir de façon identique avec l'ensemble des plate-formes supportées. Ainsi, lorsque l'on souhaite générer du code vers une nouvelle plate-forme d'exécution, aucune modification du générateur existant n'est nécessaire. Par contre, l'API doit être réécrite pour prendre en compte les spécificités de la nouvelle plate-forme. L'objectif est alors de fournir un intergiciel qui tienne compte des différences architecturales et comportementales des plates-formes visées. Pour conclure, cette solution présente certaines limitations. D'une part, les composants intergiciels doivent alors être suffisamment génériques pour pouvoir être utilisés sur l'ensemble des plates-formes d'exécution visées. C'est une tâche complexe tant les différences architecturales et comportementales sont importantes. D'autre part, cela implique des coûts supplémentaires de développement et un impact négatif sur les performances[49] dû à l'adaptation du code.

La seconde consiste à assurer la portabilité par le générateur de code (couche 2). Dans cette situation, ce dernier fournit plusieurs stratégies de génération. Cela revient à développer plusieurs générateurs. Cependant, le développement et la maintenance de plusieurs générateurs indépendants représente un coût important. Pour réduire ce coût, plusieurs techniques sont possibles :

– Décomposer la génération en étapes successives : la transformation du PIM en PSM se fait progressivement par un ensemble de raffinements produisant des PSM intermédiaires. Chaque générateur est alors défini par un ensemble de raffinements qui sont sélectionnés et peuvent être partagés. La transformation est alors formalisée dans un *workflow* [58, 82] qui décrit l'enchaînement des étapes de raffinement. Cependant, il n'y a pas de méthodologie pour identifier et factoriser les raffinements réutilisables.
– Structurer les générateurs en couches : la transformation du PIM et PSM est définie de façon modulaire. Chaque couche définit ou redéfinit une partie de la transformation. Ce type d'approche [57, 7] peut être facilement implémentée dans les transformations écrites en langages orientés-objet [7] qui bénéficient des techniques d'héritage et de composition. Cependant, dans le cas des langages de transformation relationnelle ou hybride, cette technique n'a pas été mise en avant.
– Dériver le générateur par un processus automatique de transformation : le générateur est vu comme un modèle que l'on peut transformer/raffiner par une transformation d'ordre supérieur (HOT)[79]. On définit alors un module de transformation qui définit des règles pour adapter/dériver le générateur existant. Il faut alors exécuter la transformation pour obtenir le nouveau générateur que l'on peut à son tour exécuter. L'utilisation de HOT dans le cadre des SETRC n'est pas clairement définie : en particulier les relations entre les différentes techniques d'adaptation.

**Conclusion** Les techniques d'adaptation de générateur manquent de méthodologie dans le cadre des SETRC, notamment concernant les transformations relationnelles et hybrides. Les transformations impératives, décrites en langage de programmation, bénéficient cependant des techniques de l'orienté-objet pour être facilement adaptables. Finalement, assurer la portabilité d'un générateur de code présente des difficultés. D'une part, l'utilisation d'un intergiciel générique a un impact négatif sur les performances. De plus, l'écriture de composants génériques est une tâche complexe. D'autre part, les techniques d'adaptation du générateur ne sont pas clairement définies dans le cadre des SETRC.

## 3.3 Adaptabilité du code généré

La génération de code doit ainsi proposer des stratégies afin d'assurer que le code généré soit adapté aux propriétés du système. Cependant, la plate-forme d'exécution contraint la génération de code en imposant des choix d'implémentation. Cela impacte alors les performances du système sur des critères donnés sur lesquels on ne peut intervenir. Un exemple classique est le choix d'une plate-forme d'exécution d'allouer de la mémoire pour stocker des données pré-calculées ou bien de les calculer à l'exécution. Le temps d'exécution et l'empreinte mémoire vont alors être impactées en conséquence.

Le choix d'imposer une implémentation est aussi motivé par les besoins d'analyser le comportement du système. En effet, l'étude du système requiert une connaissance globale de son comportement. La génération de code pouvant compliquer l'analyse du système global, voire rendre inopérant les outils d'analyse précédemment utilisés, certaines fonctionnalités sont alors définies statiquement au sein de la plate-forme d'exécution alors qu'elles pourraient être générées et optimisées. Dans le cas des SETRC, un exemple typique est la gestion des communications. Les communications asynchrones peuvent être implémenté de plusieurs façons, notamment sur la gestion des accès concurrents.

L'utilisation de verrou assure à la tâche courante un accès exclusif en bloquant les autres tâches. En terme de validation, il faut déterminer le temps de réponse des tâches en tenant compte de leur temps de blocage et aussi du délai due à la prise/relâche du verrou (hors blocage). Ces délais sont d'autant plus important lorsque le passage en mode privilégié implique des mécanismes coûteux de la plate-forme d'exécution : dans le cas d'une architecture ARINC653[29], le système doit garantir l'isolation temporelle et spatiale de chaque partition. Cela se traduit par des coûts supplémentaires. Dans le cas d'un ordonnancement RMS préemptif, le calcul du temps de réponse d'une tâche peut être formulé ainsi[54] :

$$r_i = C_i + B_i + \sum_{j \in hp(i)} \lceil \frac{r_i}{P_j} \rceil C_j$$

Dans cette formule, le temps de réponse $r_i$ est dépendant du pire temps de blocage $B_i$. La valeur de $B_i$ est calculé en fonction du protocole d'accès à la ressource[18] et est souvent difficile à estimer finement. On fixe alors cette valeur à une borne que l'on sait inaccessible. Cela garantit que si le temps de réponse calculé est en deça de l'échéance de la tâche, la tâche ne manquera aucune échéance. Au contraire, si le test échoue, cela signifie que la tâche manquera une échéance

dans un pire cas théorique, mais pas nécessairement dans le pire cas réel. Cet exemple montre que l'utilisation de verrou complexifie l'évaluation des performances réelles. Il existe cependant des méthodes pour réduire $B_i$, mais ce dernier n'est jamais nul. Il est donc toujours nécessaire de fournir une borne supérieure.

Une implémentation non-bloquante est une alternative à l'utilisation de verrou. On distingue les techniques *lock-free* [8, 45], qui consistent à tenter d'accéder à l'objet partagé jusqu'à ce qu'il devienne disponible, et les techniques *wait-free* [22, 47, 45] qui dupliquent les objets partagés. Ces techniques introduisent cependant des surcoûts non-négligeables, en temps d'exécution (techniques *lock-free*) ou en empreinte mémoire (techniques *wait-free*), les rendant souvent inappropriées pour les SETRC. Cependant, ces solutions sont préférables si leur coût est inférieur à l'utilisation de verrou et que la sémantique des communications permet de s'en abstraire.

L'adaptabilité de l'implémentation est également une problématique dans le domaine de la programmation orientée-aspects. Cette approche part du principe que l'entrelacement de code métier et de code technique nuit au développement et à l'évolution d'une application. Cette approche de programmation orientée composants consiste à séparer le code métier du composant de son code technique lié à l'environnement d'exécution par la définition d'*aspects*. Différents frameworks mettent en oeuvre ce type d'approche, notamment *Fractal* [16], *Kilim* [30] et *JAC* [81]. Ceux-ci reposent sur l'utilisation du langage Java pour la définition des composants et des aspects. Ils ne sont donc pas applicables dans un contexte non orienté-objet. De plus, ils ne semblent pas adresser les problématiques des SETRC que sont le déterminisme du code exécuté et le respect des contraintes temps-réel.

**Conclusion** La capacité à produire un code adapté aux contraintes du système est fortement limitée par les choix d'implémentation de la plate-forme d'exécution et des composants logiciels fournis. Cela est justifié par les difficultés d'analyser un système dont le comportement n'est pas défini statiquement et qui dépend du code généré. Ainsi, les fonctionnalités sont définies statiquement et ne peuvent pas être optimisées en fonction du système modélisé. C'est notamment le cas pour la gestion des communications asynchrones dont l'implémentation est identique à chaque système déployé sur la plate-forme d'exécution. Pouvoir choisir entre l'utilisation de mécanismes d'exclusion mutuelle avec verrou ou sans verrou permettrait d'optimiser le coût induit par ces mécanismes selon la configuration du système modélisé. Cela faciliterait l'analyse du système, en particulier l'analyse d'ordonnancement, mais nécessiterait un processus de génération favorisant la réutilisation des outils d'analyse pour tenir compte du coût des mécanismes choisis.

## 3.4 Conclusion

La conception de systèmes temps-réel critiques soulève des problématiques liées à la génération de code.

En particulier, le code généré étant plus complexe que le modèle défini par l'utilisateur, la question de l'écart entre les performances évaluées et les performances réelles se pose. Le MDA propose des techniques pour obtenir un modèle plus proche de l'implémentation réelle. Parmi ces techniques, la transformation endogène produit un modèle détaillé dans le même formalisme que le modèle initial, facilitant ainsi les analyses. L'impact de la génération de code peut alors être évalué de la même façon que pour le modèle initial. La maîtrise du coût du raffinement nécessite d'alterner transformations et analyses. Les démarches proposant ce type de processus de génération ne sont cependant pas adapté aux SETRC car requièrent un niveau d'expertise important sur les méthodes de validation.

Cette capacité à évaluer l'impact du code généré sur les performances détermine si la stratégie de génération est adaptée aux contraintes du système. Dans le cas contraire, il est nécessaire de fournir une stratégie alternative. Le coût important que représente le développement de générateurs de code indépendants pousse à vouloir étudier la portabilité d'un générateur existant afin de réduire le coût de développement du nouveau générateur. La mise en place d'une couche intergiciel générique a un impact négatif sur les performances. L'alternative est d'adapter le générateur existant : les transformations écrites en langages orienté-objet sont facilement adaptables grâce aux mécanismes d'héritage et de composition. Cependant, les techniques d'adaptation de transformations relationnelles ou hybrides ne sont pas clairement définies dans le cadre des SETRC et manquent de méthodologie.

De plus, malgré la volonté d'adapter la stratégie de génération pour réduire le coût du code généré, la plate-forme d'exécution limite la capacité à produire un code adapté. Cela se justifie par la difficulté d'analyser un système dont le comportement n'est pas défini statiquement et dépend du code généré. Certaines fonctionnalités sont ainsi implémentées statiquement au sein de la plate-forme d'exécution et ne peuvent donc être optimisées en fonction des propriétés du système. Cela nécessiterait de générer ces fonctionnalités et compliquerait l'analyse du système. Par exemple, les communications asynchrones requièrent des mécanismes qui assurent l'exclusion mutuelle aux files de messages : il existe des solutions avec verrou ou sans verrou, chacune ayant un coût qui varie selon la plate-forme d'exécution. On peut alors vouloir utiliser l'une ou l'autre selon le coût engendré. Cependant, la plate-forme d'exécution contraint ce choix.

Nous répondons à ces trois problématiques dans le chapitre suivant.

# Chapitre 4

# Approche

Le développement de générateur de code pour SETRC soulève trois problématiques. Pour y répondre, nous proposons un processus global de génération.

Afin d'évaluer et de maîtriser l'écart de performances entre modèle et code généré (problématique de *la rupture entre système modélisé et système généré*), nous nous appuyons sur un **processus de raffinement incrémental** (section 4.1). Ce processus a pour objectif d'assurer que le code généré ne remet pas en cause les contraintes du système. Celui-ci s'appuie sur la définition de stratégies de raffinement alternatives.

La mise en place de stratégies de génération alternatives étant coûteux, une méthodologie est nécessaire pour faciliter leur développement (problématique de *l'adaptabilité du processus de génération*). Face à ce manque, nous proposons une méthodologie qui favorise la réutilisation d'une transformation existante pour obtenir une nouvelle transformation. Le principe est de dériver la transformation existante en y ajoutant une surcouche. Nous définissons des patrons (section 4.2), que l'on nommera **patrons de transformation**, qui décrivent des techniques d'adaptation répondant à des objectifs différents.

La plate-forme d'exécution limitant la capacité à maîtriser le coût du code exécutable, en contraignant l'utilisation d'éléments d'implémentation qui ne sont pas toujours adaptés (problématique de *l'adaptabilité du code généré*), nous proposons une démarche de **génération de composants intergiciel adaptés** qui introduit les composants intergiciel lors de la génération de code tout en favorisant leur prise en compte lors des phases d'analyse. Nous abordons ce point dans la section 4.3.

Ces trois aspects sont traités dans les sections suivantes.

## 4.1 Processus de raffinement incrémental

Les SETRC requièrent des processus d'ingénierie qui assurent leur bon fonctionnement. La modélisation, la validation et la génération de code font partie intégrante de ces processus. Dans la problématique de *la rupture d'analyse entre système modélisé et système généré* (section 3.1), nous avons souligné que la génération de code pouvait remettre en cause les contraintes du SETRC. Les propriétés doivent donc rester analysables du modèle initial au code final. Dans cette optique, les transformations endogènes semblent être une bonne solution car favorisent la réutilisation des mêmes outils de validation sur le modèle initial et le modèle raffiné final proche du code généré. Cela assure ainsi une démarche de validation cohérente. Cette capacité à évaluer le coût de la génération de code détermine si cette dernière est adaptée aux contraintes du système. Dans le cas contraire, il faut s'orienter vers d'autres choix d'implémentation. Il faut alors intégrer des phases d'analyses au sein du processus afin d'évaluer la stratégie choisie et de la modifier si nécessaire. Les processus de génération ayant la capacité d'intégrer des phases décisionnelles nécessitent cependant un haut niveau d'expertise sur la formalisation des étapes d'analyse : les rendant difficilement utilisables dans le contexte des SETRC qui nécessitent des outils d'analyse dédiés.

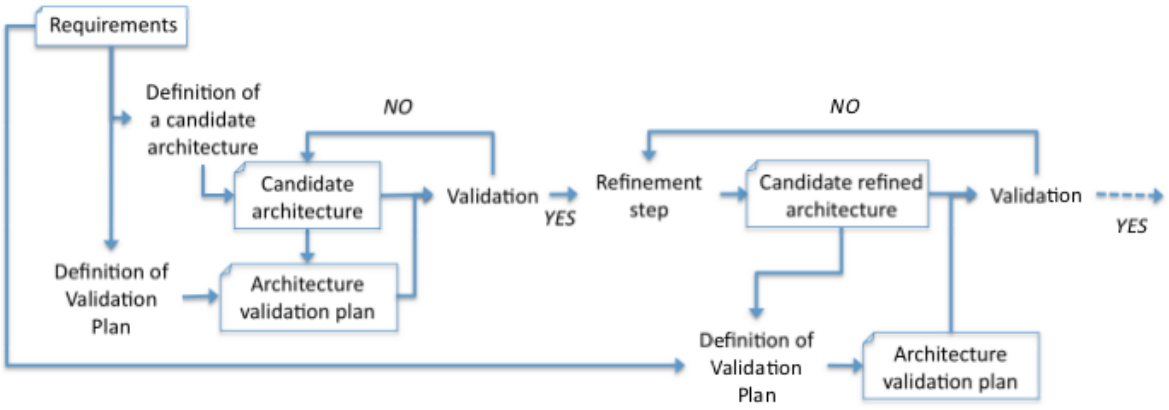



FIGURE 4.1 – Processus de raffinement incrémental

Notre première contribution est alors la définition d'un processus de raffinement incrémental qui évalue le coût de sa stratégie de génération pour l'adapter si elle remet en cause les contraintes du système. Il s'appuie alors sur des transformations endogènes (raffinements de modèle) afin d'obtenir un modèle d'implémentation. Le raffinement donne la possibilité de valider le modèle d'implémentation avec les outils de l'utilisateur utilisés sur le modèle initial. Cela assure ainsi une démarche de validation cohérente.

Le processus dispose alors d'un ensemble de stratégies de raffinement pour chaque système. Les stratégies sont testées les unes après les autres jusqu'à obtenir un raffinement qui garantisse le respect des contraintes. Ce processus est illustré par la figure 4.1. Un ensemble d'exigences est fourni par l'utilisateur. Le processus de transformation doit alors proposer une architecture qui garantisse le respect de ces exigences. Le processus dispose alors d'une liste d'architectures candidates. La première architecture candidate est évaluée. Celle-ci détaille la mise en œuvre des abstractions du modèle utilisateur. Un plan de validation détermine ensuite si cette architecture

garantit le respect des exigences. Le plan de validation peut avoir une portée globale au système ou locale à un composant. Un exemple d'analyse globale est l'analyse d'ordonnancement qui nécessite d'étudier le comportement de l'ensemble des tâches. Au contraire, une analyse de WCET va s'appliquer à une unique tâche pour évaluer son pire temps d'exécution. Si la validation échoue, l'architecture candidate suivante est évaluée. Dans le cas contraire, le modèle de l'architecture est raffiné afin d'obtenir un modèle plus détaillé. Ce dernier fait alors l'objet d'un nouveau plan de validation qui va tenir compte des nouvelles caractéristiques du modèle raffiné. Ce processus est répété jusqu'à obtenir un modèle valide et suffisamment proche d'une implémentation réelle : au sens où un raffinement supplémentaire n'introduirait plus de nouvelles ressources. Enfin, le modèle raffiné est traduit en code source (*e.g.* C, Ada) et peut être compilé.

Nous formalisons ce processus à l'aide d'une *chaîne de raffinement* qui décrit l'enchaînement des stratégies et des étapes d'analyses. Chaque stratégie de raffinement est implémentée à travers des étapes de transformation endogènes écrites dans un langage hybride comme ATL. L'utilisation d'un tel langage est motivé par le fait qu'il combine les approches relationnelles et impératives, et offre ainsi plus de flexibilité [24]. Avant la première transformation, une analyse globale est réalisée sur le modèle d'entrée pour déterminer si la spécification respecte les contraintes du système, avant même de tenir compte du raffinement. Cette première analyse est nécessaire pour déterminer si une future analyse échoue à cause du surcoût induit par un raffinement ou si cela est dû au modèle initial, et dans ce cas, aucun raffinement n'est possible. Ensuite, chaque transformation produit un modèle raffiné qu'il est possible d'analyser. Dans ce cas, à la transformation peut succéder une ou plusieurs analyses réalisées par les mêmes outils que ceux utilisés sur le modèle d'entrée. Pour évaluer l'écart entre le modèle précédent et le modèle raffiné, on réévalue le WCET de chaque tâche du modèle raffiné pour prendre en compte le coût des composants intergiciel introduits dans leur spécification. Les annotations du modèle sont mises à jour, notamment la propriété WCET. On peut alors réaliser de nouveau les analyses précédentes qui vont pouvoir prendre en compte ces coûts supplémentaires et ainsi déterminer si l'implémentation choisie assure le respect des contraintes du système. Dans le cas contraire, un raffinement alternatif doit être choisi. Pour se faire, le modèle raffiné actuel n'est pas conservé et la transformation alternative s'effectue alors sur un modèle antérieur. Les alternatives sont testées les unes après les autres dans un ordre prédéfini : le coût d'une alternative n'étant connu qu'une fois appliquée. Ce processus s'assure donc qu'une fois le code généré, l'écart de performances avec le modèle initial ne remet pas en cause la validité du système.

La spécification des exigences et des plans de validation d'un tel processus peut être formalisée à l'aide d'un langage de spécification tel que RDAL[15]. Notre contribution se focalise sur les phases de raffinement d'architecture. La spécification du processus de raffinement de l'architecture peut être formalisée en langage XML comme illustré par le listing 4.1. Un identifiant est donné au modèle initial en spécifiant l'attribut *IN* de la balise *process* (ici l'identifiant donné est $inputModel$). Une analyse initiale identifiée par l'attribut *startAnalysisID* est effectuée sur le modèle d'entrée et stoppe le processus si celle-ci échoue. Sinon, le processus teste une première transformation endogène $strategie1$ dont l'implémentation est fournie par les fichiers listés dans l'attribut *modules*. Le modèle d'entrée à transformer est identifié par l'attribut *IN* tandis que l'attribut $OUT$ affecte un identifiant ($refined1$) au modèle raffiné. Le coût du raffinement est ensuite évalué par une analysis $WCET$ réalisé sur le modèle raffiné $refined1$. Cette analyse réévalue et met à jour les propriétés temporelles du modèle. Ensuite, le modèle raffiné est analysé de nouveau. Si l'analyse réussie, alors le surcoût du raffinement ne remet pas en cause les contraintes du sys-

tème et le code exécutable peut alors être généré (balise *yes*). Dans le cas contraire, une seconde transformation est testée sur le modèle initial. De la même façon, les mêmes analyses sont réalisées pour déterminer l'impact de la seconde stratégie de raffinement. Le code exécutable est alors généré si l'analyse réussie. Le formalisme XML est donc tout à fait adapté à la mise en œuvre des phases de raffinement d'architecture du processus décrit précédemment par la figure 4.1.

```
1  <process IN="inputModel" startAnalysisID="ResponseTime">
2    <trans ID="strategie1" modules="..." IN="inputModel" OUT="refined1">
3      <analysis ID="WCET" IN="refined1"/>
4      <analysis ID="ResponseTime" IN="refined1">
5          <yes><generation IN="refined1"/></yes>
6          <no>
7               <trans ID="strategie2" IN="inputModel" OUT="refined2" modules="...">
8                  <analysis ID="WCET" IN="refined2"/>
9                  <analysis ID="ResponseTime" IN="refined2">
10                     <yes><generation IN="refined2"/></yes>
11                     <no>...</no>
12              </trans>
13         ...
```

Listing 4.1 – Exemple de formalisation en XML d'un processus de raffinement incrémental

Le respect des contraintes du SETRC peut ainsi être assuré par ce type de processus, en fournissant des stratégies alternatives automatiquement évaluées. La prise en compte de l'impact de la génération de code sur les performances du système n'est alors pas dépendante du niveau d'expertise de l'utilisateur. Cependant, le développement de multiples alternatives est relativement coûteux et rend difficile la maintenance d'un tel processus. Nous abordons ce second point dans la section suivante.

## 4.2 Adaptation de transformation

Définir un générateur de code proposant plusieurs stratégies (section 4.1) dont le coût de chacune est évalué répond ainsi à la problématique de *la rupture d'analyse entre système modélisé et système généré*. Le développement de stratégies de génération indépendantes est néanmoins relativement coûteux, notamment en terme de maintenance. On s'intéresse alors à répondre à la problématique de *l'adaptabilité du processus de génération* (section 3.2) afin de faciliter la mise en place d'un tel processus. Notre deuxième contribution est alors la définition d'une démarche pour réutiliser une transformation existante afin de la dériver en une nouvelle transformation, adaptée aux nouveaux besoins. Pour réaliser cette adaptation, nous ajoutons à la transformation des couches supplémentaires qui la redéfinissent partiellement. La nouvelle transformation est donc obtenue par assemblage de la transformation existante et des couches ajoutées. Pour cela, nous nous appuyons sur un mécanisme existant de composition qui sera abordé en section 5.1.3. Nous proposons alors des patrons de transformation pour faciliter l'adaptation de règles de transformation.

La mise en œuvre de ces patrons nécessite de réécrire une règle existante afin de l'adapter selon les besoins. Les parties de la règle que l'on souhaite adapter sont déléguées à des fonctions annexes. De cette manière, l'adaptation de la règle ne requiert qu'une modification des fonctions

annexes et conserve la définition globale de celle-ci. Pour réaliser l'adaptation, on ajoute à la transformation une surcouche (*i.e.* une seconde transformation) qui redéfinit ces fonctions annexes. La transformation dérivée est alors obtenue en assemblant et en compilant ensemble la transformation initiale et les couches ajoutées. Ainsi, selon les objectifs, on sélectionne les couches à ajouter à la transformation. On conserve ainsi la définition de la transformation initiale, que l'on peut utiliser séparément. Cette approche facilite le développement de stratégies de génération alternatives en favorisant la réutilisation. Ces patrons de transformation sont inspirés des patrons de conception de l'orienté-objet [34]. Ils répondent à des objectifs similaires :

**Stratégie** La mise en place d'un processus de raffinement incrémental, comme celui proposé en section 4.1 repose sur des stratégies de transformation alternatives. La définition de variantes nécessite de spécialiser la logique de transformation. Le patron *Stratégie*, de l'orienté-objet, a pour objectif d'altérer le comportement d'un objet en fonction du contexte en déléguant une partie du comportement à un objet tierce que l'on peut remplacer. Dans le cas d'un processus de transformation, l'objectif est de pouvoir adapter la transformation au contexte. Ainsi, la définition d'un tel patron consisterait alors à fournir des variantes d'une transformation en remplaçant les objets annexes à la transformation. Les étapes de transformation et d'analyse sont alternées de manière à déterminer si la transformation est adaptée aux exigences ou si son comportement doit être altéré en remplaçant les objets annexes à la transformation. Soit $M$ le module contenant la définition initiale de la règle $R$ que l'on souhaite modifier par la suite. La partie à altérer de la règle $R$ est externalisée dans une fonction annexe $H$. Un second module $M'$ modifie la définition de la fonction $H$ de manière à altérer le comportement de la règle $R$. Dans le cas où l'on souhaite utiliser la règle initiale, seul le module $M$ est chargé. Sinon, on charge $M$ puis $M'$ afin de redéfinir $H$.

**Adaptateur** La définition d'une logique de transformation est dépendante de l'interface des éléments manipulés. Plus les interfaces sont hétérogènes plus le nombre de règles de transformation est important, et plus le processus est difficile à maintenir. La complexité d'un processus de transformation dépend donc en partie de l'homogénéité de ces éléments. Le patron *Adaptateur* de l'orienté-objet répond justement à ce type de problématique : l'objectif est d'intégrer dans une architecture des types externes qui n'ont pas été conçus pour. Le principe est d'alors de définir de nouveaux types, compatibles avec l'architecture, qui encapsulent les types externes. Du point de vue d'un langage de transformation, il s'agit de réutiliser une règle de transformation sur un type d'élément d'entrée qui n'est initialement pas compatible avec l'interface des éléments d'entrée de la transformation. Cela permet de bénéficier d'une règle existante plutôt que définir une nouvelle règle au comportement similaire.

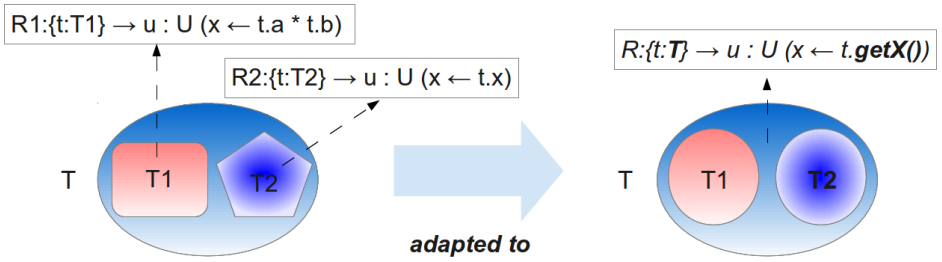



FIGURE 4.2 – Utilisation du patron Adaptateur pour factoriser deux règles définies sur des types hétérogènes

**Substitution partielle** Ce patron n'a pas d'équivalent dans l'orienté-objet. L'objectif est d'assurer la compatibilité d'une transformation sur des plates-formes d'exécution distinctes. En effet, certains concepts sont spécifiques à une plate-forme. Pour tout autre type de plate-forme, ces concepts n'existent pas et sont assimilés à d'autres concepts plus généraux. En conséquence, ils sont traduits de la même façon. Si l'on souhaite adapter une transformation définie pour une première plate-forme d'exécution, afin de la porter vers une seconde, il faut s'assurer que l'interprétation des concepts de la première soit compatible avec l'interprétation qui en faite dans la seconde. Si ce n'est pas le cas, les concepts assimilés dans la première, doivent être distingués dans la seconde. On souhaite alors redéfinir une règle de la première plate-forme pour qu'elle réduise son champ d'application aux concepts assimilables. La réalisation de ce patron consiste alors à externaliser l'expression définissant l'ensemble d'entrée de la transformation afin de pouvoir le réduire ultérieurement en redéfinissant cette expression. Ce principe est illustré par la figure 4.3. Une règle $R$ est initialement appliquée sur tous les éléments de type $T$. La prise en compte d'une nouvelle plate-forme d'exécution nécessite de distinguer deux sous-ensembles d'éléments. Les éléments de type $T$ qui valident la condition $a \leq 0$ ne doivent pas être traités par la règle $R$. L'application de *Substitution partielle* favorise la réutilisation de la règle $R$ en réduisant son champ d'application dans le cas de la seconde plate-forme d'exécution.

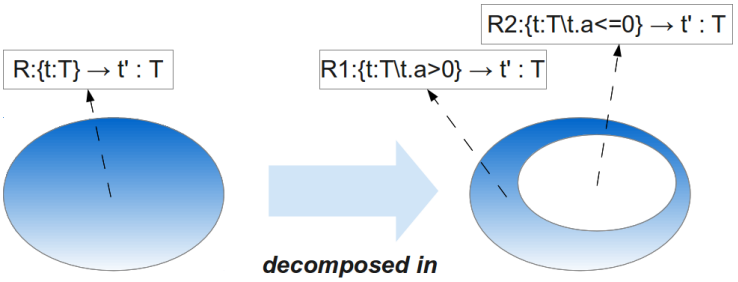



FIGURE 4.3 – Principe du patron Substitution Partielle

**Memento** Ce patron s'applique dans le cadre de la certification du processus de génération. La certification requiert d'assurer une cohérence globale du processus. On souhaite par exemple déterminer l'origine du code généré (*quelle exigence a conduit à la production de cette partie du code ?*) ou au contraire vérifier que les exigences exprimées en amont (lors de la conception) se retrouvent dans le code généré. On parle alors de traçabilité des exigences. Pour cela, on produit un ensemble de traces lors de la génération de code afin de faire le lien entre les propriétés définies par l'utilisateur et le code généré à partir de celles-ci. Concrètement, il s'agit de retrouver pour chaque élément cible crée, l'ensemble des éléments source qui ont conduit à sa création. La conservation des traces de transformation donne la possibilité de mettre en correspondance des éléments du modèle source et des éléments de code et de vérifier la cohérence du processus. La figure 4.4 décrit un exemple de métamodèle définissant les structures nécessaires à la mise en oeuvre du patron.

Pour conclure, nous avons présenté des patrons de transformation facilitant la mise en œuvre du processus de génération défini en section 4.1. Nous illustrerons l'utilisation des patrons en section 5.2. Le patron *Stratégie* aide à implémenter des stratégies alternatives par réutilisation d'une transformation existante. Le patron *Adaptateur* réduit la complexité de la logique de transformation en homogénéisant les interfaces des objets. L'intégration de nouvelles plates-formes d'exécu-

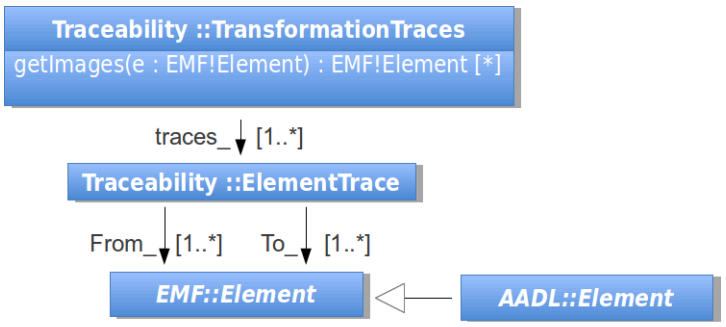



FIGURE 4.4 – Métamodèle de traçabilité pour la mise en oeuvre du patron Memento

tion est facilitée par le patron *Substitution partielle* qui adapte la portée d'une transformation aux spécificités de chaque plate-forme. Enfin, la certification du processus est rendue possible grâce à l'application du patron *Memento*. Ces patrons facilitent donc l'adaptation du processus notamment dans l'objectif de maîtriser le coût du code généré. Cependant ce dernier est en partie limité par les choix d'implémentation imposés par la plate-forme d'exécution. Nous répondons à cette problématique dans la section suivante.

## 4.3 Génération de composants intergiciel adaptés

Le développement de générateur de code pour SETRC requiert une maîtrise du coût de développement et des performances du système généré. Face à cette difficulté, nous avons proposé un processus de génération offrant des stratégies alternatives (section 4.1) qui implique une variabilité sur les composants intergiciel utilisés. Pour faciliter le développement d'un tel générateur, nous avons défini des patrons de transformation (section 4.2) qui favorisent la réutilisation et l'adaptation de ces composants. Cependant, le choix des composants induit un coût sur les performances du système. L'utilisation systématique des mêmes services imposés par l'intergiciel rend difficile l'optimisation des composants en exploitant les connaissances de l'architecture du système (problématique de *l'adaptabilité du code généré*). La possibilité d'adapter ces composants en les générant compliquerait l'évaluation des performances réelles du système. De plus, les outils de validation ayant des hypothèses d'utilisation, adapter le comportement de la plate-forme à chaque système rendrait difficile l'utilisation de ces outils. Nous proposons alors d'introduire ces composants au sein du modèle lors du processus de raffinement (section 4.1) comme illustré par la figure 4.5. L'introduction des composants a pour objectif d'obtenir un modèle d'implémentation proche du code généré et dont le coût peut être évalué plus précisément que sur le modèle initial.

L'objectif est d'adapter l'implémentation en fonction de la configuration du système modélisé tout en permettant de tenir compte de son coût. L'exemple des communications est représentatif de cette problématique, la plate-forme d'exécution fournissant la plupart du temps ces mécanismes. Nous nous focalisons alors sur cet exemple pour illustrer notre approche. En particulier, nous nous intéressons aux communications différées des architectures *Time-Triggered*[55, 46, 62] (présentées en section 2.2.1) en raison de leur déterminisme. Pour illustrer ce modèle dans un contexte asynchrone, nous considérons l'ordonnancement de tâches périodiques de la figure 4.6. Chaque tâche $T_i$ possède une période $P_i$, un pire temps d'exécution $C_i$, ainsi qu'une date d'échéance $D_i$

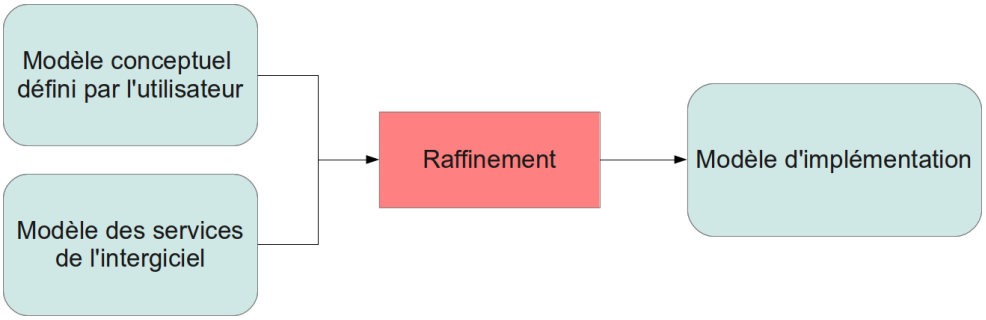



FIGURE 4.5 – Introduction des composants intergiciel dans le processus de raffinement

telle que $D_i \leq P_i$. Le jeu de tâches est constitué de $T_1$, $T_2$ et $T_3$, avec $i = 1..3$, $P_i = D_i$, $P_1 = 5$, $P_2 = 7$, $P_3 = 10$. $T_1$ et $T_2$ émettent périodiquement des messages à $T_3$ à la fin de leur exécution. La sémantique différée implique que les messages $a1$, $a2$ et $b1$ soient datés de l'échéance de la tâche émettrice, c-a-d respectivement 5,10 et 7, indépendamment de la date physique à laquelle le message a été émis. $T3$ ne peut accéder à ces messages qu'à partir de la période qui succède leur date. A $t = 10$, $T3$ commence sa $2^{eme}$ période et peut ainsi lire les messages $a_1$, $b_1$ et $a_2$, ordonnés par date. Le message $b_2$ pourtant émis avant l'activation de $T3$ ne sera disponible qu'à la prochaine période de $T3$ ($t = 20$) car $b_2$ est daté de l'échéance de $T2$, c-a-d 14, et donc supérieur à $t = 10$. Ce modèle de communication assure une exécution déterministe dans le sens où l'état des files de messages (nombre de messages et ordre) est connu statiquement à chaque instant.

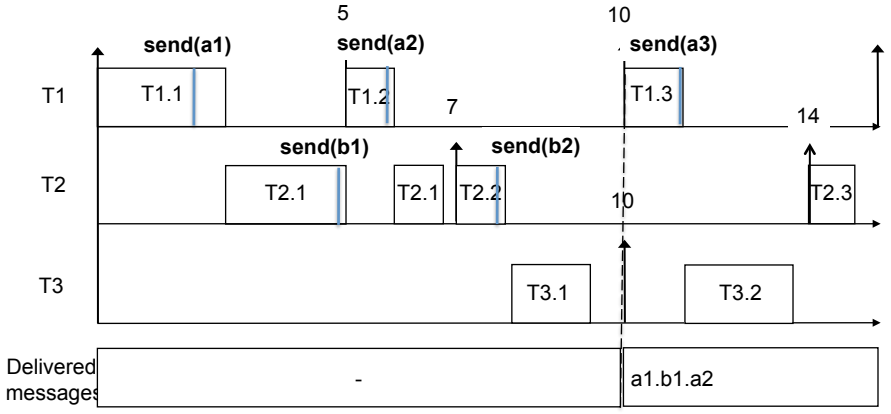



FIGURE 4.6 – Exemple d'exécution de tâches utilisant des communications différées

Nous nous intéressons à ce type de communication pour une application dans un cadre plus général que les architectures proposées. Le choix de l'implémentation d'un tel modèle de communication soulève la question du coût des mécanismes que l'on souhaite utiliser. Soient *send* et *receive* les fonctions d'envoi et de réception.

**Facilité et inconvénients d'une implémentation avec verrou** Une implémentation simple à mettre en œuvre est de réaliser une insertion triée côté tâche émettrice (fonction *send*). L'insertion triée est alors basée sur la date d'échéance courante. Côté tâche réceptrice, la fonction *receive* con-

somme le message le plus récent si sa date est inférieure à la période courante de la tâche réceptrice. Dans le cas contraire, la fonction renvoie le message précédemment consommé. De plus, la file doit être protégée par un verrou pour garantir l'exclusion mutuelle. Cette solution implique cependant des coûts particuliers qu'il faut estimer. D'une part, l'utilisation de verrou nécessite d'estimer finement le temps d'exécution des mécanismes d'exclusion mutuelle. Comme nous l'avons souligné en section 3.3, les verrous conduisent à des analyses pessimistes. D'autre part, l'utilisation d'une fonction d'insertion triée peut également représenter un coût non négligeable.

**Difficulté et avantages d'une implémentation sans verrou** La solution précédente est certes simple à implémenter statiquement au sein d'une plate-forme d'exécution, cependant les coûts induits peuvent conduire à s'orienter vers une solution alternative. Nous proposons une approche sans verrou (implémentations *lock-free*[45]) et sans insertion triée. L'implémentation précédente requiert l'utilisation de verrou à cause notamment des fonctions *send* et *receive* qui réalisent des décalages sur l'ensemble de la file. Une alternative est alors de réaliser les insertions et les lectures à des indices pré-déterminés. Ces indices assurent à la fois l'exclusion mutuelle (deux tâches ne peuvent accéder au même indice pendant un intervalle de temps donné) et l'ordre de réception des messages basé sur la date. Cette solution requiert donc une méthode de calcul d'indices qui assure ces deux propriétés. Pour réaliser ce calcul, nous pouvons nous baser sur l'ordre global des échéances de l'ensemble des tâches sur l'hyperpériode. Les échéances sont alors numérotées. En cas d'échéances simultanées, une règle arbitraire définie statiquement doit attribuer un ordre déterministe. A chaque période, à chaque tâche et à chaque file de message on détermine alors le numéro de l'échéance. Ce dernier est ensuite utilisé dans le calcul de l'indice à laquelle la tâche va lire ou écrire. Le calcul d'indice peut alors être soit réalisé à l'exécution ou hors-ligne. Dans le second cas, on définit pour chaque tâche et chaque file de message, un tableau d'indices. La taille du tableau est alors le nombre d'activations de la tâche sur l'hyperpériode.

En termes de performances, chacune de ces trois solutions (utilisation de verrou, indices calculés à l'exécution, indices calculés hors-ligne) peut représenter un coût non négligeable. D'une part, la seconde solution peut représenter un coût plus important en temps d'exécution que d'utiliser des verrous et des insertions triées. Néanmoins, cela dépend du coût de ces mécanismes qui varie selon les plates-formes d'exécution. Il faut donc envisager chaque cas. D'autre part, la troisième solution est nettement moins coûteuse en temps d'exécution. Cependant, les tableaux nécessaires pour stocker ces indices peuvent avoir une empreinte mémoire importante si l'hyperpériode est relativement grande. Le coût de cette autre alternative dépend donc des périodes des tâches. Cette dernière solution est de plus difficile à maintenir : si le jeu de tâches évolue, le tableau d'indices doit être revu pour tenir compte des nouvelles tâches ou des modifications sur les propriétés temporelles des tâches existantes. Chaque solution est donc envisageable selon les caractéristiques du système et de la plate-forme d'exécution (mécanismes fournis, propriétés du jeu de tâche, etc...) et des efforts de maintenance. La mise en place d'un générateur de code fournissant les trois alternatives semble donc appropriée à la problématique de *l'adaptabilité du code généré*.

Finalement, définir statiquement les fonctionnalités au sein d'une plate-forme d'exécution conduit à choisir des algorithmes simples à mettre en œuvre. Néanmoins, ceux-ci ne sont pas nécessairement les plus efficaces dans toutes les situations. Nous l'avons illustré avec le cas des communications. La génération de code peut cependant aider à la mise en place de ces fonctionnalités mais peut, par la même occasion, compliquer l'analyse du système, voire rendre les outils d'analyses inopérants si l'impact du code généré n'est pas mesurable. Pour favoriser l'adaptation des

composants intergiciel tout en garantissant la capacité d'analyse, nous proposons de générer ces composants à travers des raffinements de modèle. Nous proposons ce processus dans le cas des communications différées. Cette approche sera détaillée dans les chapitres suivants.

## 4.4 Conclusion

Face aux problématiques soulevées liées au développement de générateur de code pour SETRC, nous définissons une démarche de conception de générateur de code.

L'écart de performances entre système modélisé et système généré peut être évalué grâce aux techniques de raffinements de modèle qui permettent d'obtenir un modèle détaillé analysable. Pour maîtriser le coût du code généré, nous formalisons le processus de génération à travers une *chaîne de raffinement* afin de fournir des stratégies de génération/raffinement alternatives dans le cas où la stratégie choisie produit un code remettant en cause les contraintes du système. L'évaluation du coût n'est pas dépendante de la capacité d'analyse de l'utilisateur en favorisant la réutilisation des outils de validation utilisés sur le modèle initial. Cela assure par ailleurs une démarche de validation cohérente en reposant sur un même langage standardisé avec une sémantique bien définie.

La mise en place d'un tel processus de génération peut soulever des difficultés en terme de maintenance et d'évolution. Le développement indépendant de stratégies de génération pouvant notamment être relativement coûteux. Nous proposons des patrons de transformation pour favoriser la réutilisation de stratégies existantes afin de les dériver. Chaque étape de transformation du processus est alors conçue en couches qui définissent ou redéfinissent une partie de la transformation. Nous nous inspirons des patrons de conception de l'orienté-objet qui répondent à des objectifs similaires.

Enfin, la capacité à maîtriser le coût du code exécutable étant limitée par les choix d'implémentation imposés par la plate-forme d'exécution, certains composants intergiciel ne peuvent être adaptés à chaque système. Par conséquent, leur coût peut conduire à la violation des contraintes du système dans certains cas, alors qu'adapter ces composants garantirait le respect de ces contraintes. Nous proposons alors une démarche pour générer des composants intergiciel adaptés qui facilite néanmoins la prise en compte de leur coût lors de la validation. Nous avons illustré cette démarche autour de la gestion des communications entre tâches. Chaque implémentation a un impact différent sur les performances selon les caractéristiques du système modélisé (*e*.g. nombre de tâche, hyperpériode, coût des mécanismes utilisés) qui doit être évalué dans chaque configuration. La mise en place d'un processus adaptant les composants intergiciel rend donc possible l'optimisation du coût du code généré, et par conséquent facilite la validation du système.

Le chapitre suivant détaille ces trois contributions. La première section décrit la formalisation du processus de raffinement incrémental. Ensuite, nous détaillons et illustrons l'utilisation des patrons de transformation. Enfin, présentons plus en détails l'utilisation de composants intergiciel adaptés à travers la mise en œuvre des communications entre tâches.

# Chapitre 5

# Contribution

Le développement de générateur de code pour SETRC soulève des problématiques qui ont été soulevées au chapitre 3. Nous avons répondu à ces problématiques au chapitre 4 en proposant un processus global. Ce chapitre détaille l'application de notre approche.

Premièrement, face à la problématique de la *rupture d'analyse entre système modélisé et système généré*, nous avons proposé un *processus de raffinement incrémental* (section 4.1) dont l'objectif est d'assurer que le code généré garantit le respect des contraintes du système. Afin de mettre en œuvre ce processus, nous le formalisons par une *chaîne de raffinement* dont nous définissons le méta-modèle en section 5.1. Le chapitre suivant détaillera sa mise en œuvre au sein d'un environnement de conception de SETRC.

Deuxièmement, en réponse à la problématique de l'*adaptabilité du processus de génération*, des patrons de transformation ont été proposé en section 4.2 pour réaliser une *adaptation de transformation* afin d'obtenir des stratégies alternatives de raffinements en favorisant la réutilisation de transformations existantes. Nous présentons plus en détail ces patrons de transformation en section 5.2, en donnant pour chacun un cas d'utilisation, une méthodologie et un exemple d'application afin de démontrer leur applicabilité et de faciliter leur prise en main.

Troisièmement, pour répondre à la problématique de l'*adaptabilité du code généré*, une démarche de *génération de composants intergiciels adaptés* a été proposé en section 4.3. L'intergiciel imposant l'utilisation systématique des mêmes composants, ces derniers limitent l'optimisation du code généré en fonction de l'architecture modélisée. Nous avons alors proposé d'intégrer ces composants dans le processus de raffinement dans le but de faciliter leur analyse et leur adaptation. Nous précisons dans ce chapitre le processus d'intégration des composants intergiciels ainsi que leur modélisation et leur évaluation dans le cadre des communications entre tâches.

Notre approche nécessite un langage de transformation afin de définir les étapes de raffinement du processus ainsi que les patrons de transformations. Elle requiert également un langage de modélisation de SETRC qui supporte le raffinement de modèle conceptuel en modèle d'implémentation, afin d'assurer une compatibilité avec les outils d'analyse. Pour illustrer et expérimenter

notre approche, nous avons choisi :
– le langage de transformation **ATL** (*Atlas Transformation Language*) : ATL combine les approches de transformation relationnelles et impératives, rendant son utilisation assez simple. De plus, il est régulièrement maintenu et est intégré dans les environnements de développements tels que Eclipse.
– le langage de modélisation **AADL** (*Architecture Analysis Description Language*) : AADL offre une sémantique riche qui facilite aussi bien la spécification de modèles conceptuels que de modèles d'implémentation bas niveau. Ce langage est donc adapté au raffinement de modèles et a déjà été utilisé dans ce but dans [60].

## 5.1 Processus d'analyse et de transformation

Dans cette section nous détaillons le *processus de raffinement incrémental* que nous avons commencé à décrire en section 4.1. Il répond à la problématique de *la rupture d'analyse entre système modélisé et système généré* (section 3.1). Ce processus fournit des stratégies alternatives de raffinement pour optimiser l'implémentation, en fonction de chaque architecture modélisée, dans le but de maîtriser le coût du code généré et de garantir le respect des contraintes du système généré. En section 5.1.1, nous définissons les objectifs que doit remplir un tel processus. Nous définissons ensuite un méta-modèle de *chaîne de raffinement* en section 5.1.2 qui répond aux objectifs fixés. Nous nous appuyons sur une technique existante de composition pour faciliter l'adaptation et la réutilisation de transformations. Cette technique est abordée en section 5.1.3. Enfin, nous fournissons une méthode d'utilisation du méta-modèle en section 5.1.4 et nous illustrons son utilisation dans la section 5.1.5.

### 5.1.1 Objectifs

Pour modéliser le *processus de raffinement incrémental* introduit précédemment, nous donnons les objectifs que doit remplir un tel processus.

– **Objectif 1 : Raffiner le modèle** En réponse à la *rupture d'analyse entre système modélisé et système généré*, le principe du *processus de raffinement incrémental* est de transformer des modèles conceptuels en modèles d'implémentation proches du code final généré. Le raffinement est donc la transformation de constructions complexes, de haut-niveau d'abstraction, en plusieurs constructions moins complexes plus bas-niveau. Le premier objectif est ainsi de raffiner un modèle selon une stratégie particulière afin d'obtenir un modèle raffiné, plus proche du code final, qui soit analysable. La difficulté est alors de fournir des étapes de transformations capables de raffiner un modèle conceptuel en modèle d'implémentation proche du code généré.

– **Objectif 2 : Formaliser les stratégies en factorisant les logiques de raffinement** L'objectif précédent est de conduire à des modèles raffinés analysables. Par conséquent, le processus va déterminer si sa stratégie actuelle garantit le respect des contraintes du système. Dans le cas contraire, le but est de changer de stratégie de génération pour assurer ces contraintes. Face au coût que représente le développement indépendant de stratégies alternatives (problématique de l'*adaptabilité du processus de génération*), nous avons proposé une méthode d'*adaptation*

*de transformation*. Cette méthode favorise la réutilisation de transformations existantes pour en définir de nouvelles. Pour cela, les transformations sont définies de façon modulaire et sont adaptées à différentes stratégies à l'aide de patrons que nous avons introduit. L'objectif est alors de formaliser chaque étape de transformation du processus en spécifiant les modules qui la composent.

– **Objectif 3 : Évaluer le coût induit par le raffinement** La définition de stratégies alternatives de raffinement nécessite d'analyser les performances du modèle raffiné produit après une transformation. Les éléments particuliers introduit lors d'un raffinement vont potentiellement nécessiter de nouvelles analyses et vont conduire à réévaluer certaines propriétés du modèle. Par exemple, l'introduction de composants intergiciels au sein d'un composant modélisant une tâche nécessite une réévaluation de propriétés temporelles comme le WCET (pire temps d'exécution) requises pour réaliser une analyse d'ordonnancement qui prenne en compte l'exécution de ces composants intergiciels. Ainsi, a tout moment l'utilisateur a la possibilité d'insérer une ou plusieurs analyses après une transformation, en spécifiant à chaque fois les outils d'analyse qu'il souhaite utiliser. Le processus de raffinement incrémental doit mettre à jour les propriétés du modèle raffiné pour prendre en compte le surcoût du raffinement.

– **Objectif 4 : Changer de stratégie** Il est ainsi possible de déterminer si la stratégie courante est adaptée à la configuration du système en évaluant le modèle raffiné. Dans le cas où la stratégie courante n'est pas adaptée, il faut changer de stratégie. Par conséquent, après chaque étape d'analyse le processus va potentiellement se diviser en deux branches, l'une définit la suite du processus dans le cas où l'analyse réussie, c-a-d dans le cas où la stratégie est adaptée : on procède alors à la génération de code ou à un second raffinement qui produit un modèle raffiné plus détaillé que le précédent. L'autre définit la suite du processus dans le cas où l'analyse échoue : le processus change de stratégie pour assurer que le système final respecte les contraintes initiales. L'utilisateur spécifie ces deux branches après chaque étape d'analyse. De plus, lors d'un changement de stratégie, il est nécessaire de retrouver un modèle antérieur au dernier modèle raffiné afin de ne pas tenir compte de l'effet du dernier raffinement fautif.

– **Objectif 5 : Obtenir une trace des raffinements réalisés** Certains SETRC requièrent le respect de certaines normes afin d'apporter des preuves de confiance envers leur processus de génération de code. Une certification du processus de génération est nécessaire pour assurer une conformité du code généré vis-à-vis des exigences. Cela se traduit par une traçabilité de ces exigences depuis la spécification initiale jusqu'au code final généré. Le processus fournit des traces qui font la correspondance entre des éléments de spécification et des éléments de code généré. Dans un processus de raffinement incrémental, les exigences sont raffinées au fur à mesure des étapes de raffinement. Il est donc important de s'assurer de la conservation de ces exigences après chacune de ces étapes. De plus, l'expertise du processus nécessite une compréhension des différentes étapes qui se sont succédées, afin de justifier notamment les changements de stratégie. Cela implique un processus global fournissant des traces des différents raffinements réalisés pour donner la possibilité d'analyser la logique globale du processus.

– **Objectif 6 : Générer le code exécutable à partir du modèle raffiné** L'objectif final du processus est la production de code exécutable pour une plate-forme d'exécution donnée. Le processus

de raffinement n'étant pas dépendant du langage du code généré, le même processus doit être compatible avec des plates-formes distinctes qui ne divergent que sur la syntaxe de leur code. La plate-forme ciblée doit cependant être spécifiée pour identifier le langage cible et pour produire les fichiers de configuration et de compilation. Pour garantir une certaine cohérence du processus, le passage du modèle raffiné en code exécutable ne doit introduire aucune ressource supplémentaire qui pourrait invalider la stratégie de raffinement. Le modèle raffiné final correspond alors à un modèle d'implémentation bas niveau que l'on traduit en code exécutable.

Nous avons listé les objectifs de la *chaîne de raffinement* afin d'assurer la maîtrise du coût du code généré et de son impact sur les performances réelles du système. Un tel processus de génération est illustré par la figure 5.1. L'utilisateur fournit un modèle conceptuel au processus qui tente un ensemble de stratégies de raffinement jusqu'à aboutir à un code exécutable respectant les contraintes du système. Le processus alterne raffinement et analyses afin d'évaluer l'impact du raffinement courant sur les performances du système. L'analyse conditionne le déroulement du processus en déterminant si le prochain raffinement sera en continuité avec le précédent ou bien si le processus doit revenir en arrière pour modifier sa stratégie.

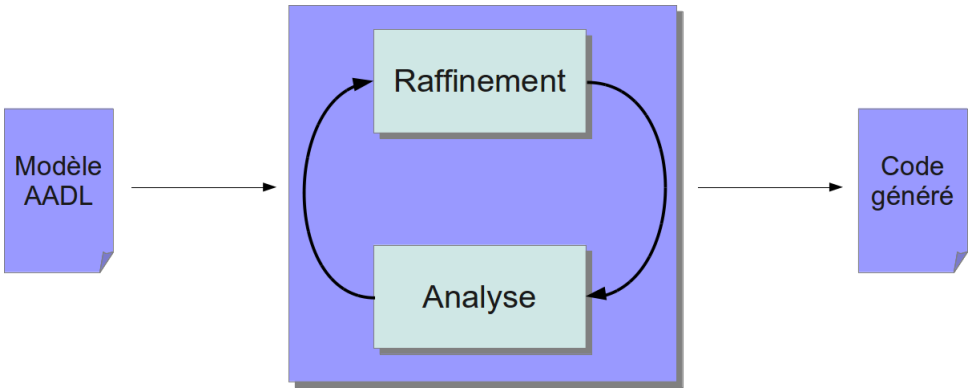



FIGURE 5.1 – Illustration du processus de raffinement incrémental

L'utilisateur doit ainsi spécifier l'enchaînement d'actions diverses telles que le raffinement, l'analyse, la production de traces et la génération de code exécutable pour la plate-forme d'exécution choisie. Nous définissons en section 5.1.2, un méta-modèle définissant un formalisme de *chaîne de raffinement* qui répond à ces objectifs.

### 5.1.2 Méta-modèle du processus

Pour mettre en place le *processus de raffinement incrémental*, nous définissons un méta-modèle de *chaîne de raffinement* en partie représenté par la figure 5.2. Nous décrivons celui-ci dans les paragraphes suivants.

La définition d'une *chaîne de raffinement* commence par la création d'un objet de type *Workflow*. Le point d'entrée du processus est déterminé par l'utilisateur : validation préliminaire du

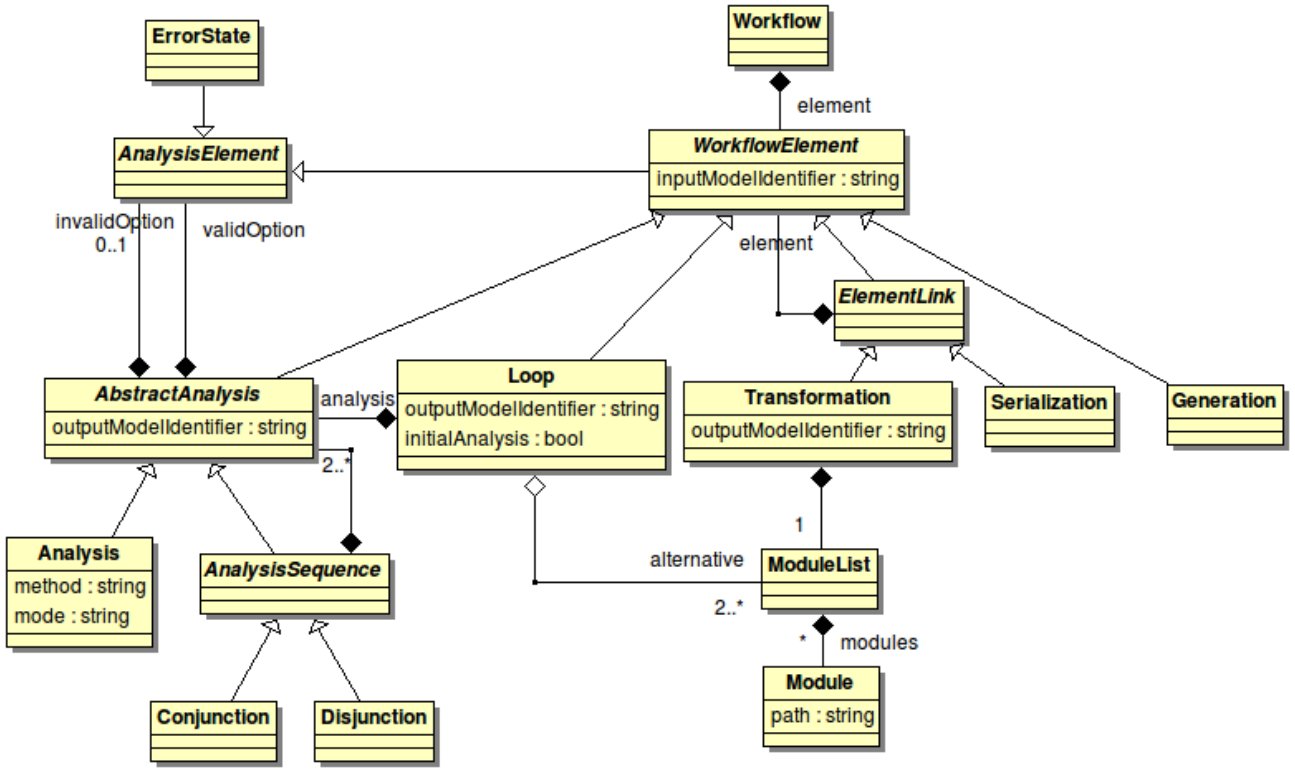



FIGURE 5.2 – Vue d'ensemble du méta-modèle de *chaîne de raffinement*

modèle initial, transformation... Dans ce cas, le méta-modèle définit le type générique *Workflow-Element* qui représente une étape du processus au sens général. Le point d'entrée est alors un élément de type *WorkflowElement* qui se décline en sous-types *Transformation*, *AbstractAnalysis*, *Serialization* pour la production d'une version textuelle du modèle raffiné, ou *Generation* pour la génération de code exécutable final. Nous citerons également le type *Loop* qui modélise de façon plus compacte l'utilisation de stratégies de transformations alternatives à une étape du raffinement donnée. Nous décrivons chacune de ces étapes dans les paragraphes suivants.

**Transformation** Afin de répondre à l'**objectif n° 1**, le raffinement du modèle est réalisé lors d'étapes de type *Transformation*. Chacune produit un modèle raffiné dans le même formalisme que le modèle source. Nous nous sommes inspirés du travail réalisé dans [59] pour obtenir des modèles d'implémentation proches du code généré. Pour répondre à l'**objectif n° 2**, chaque étape de transformation spécifie les modules qui la composent, ceux-ci facilitant l'adaptation et la réutilisation de transformations. Pour chaque module, l'utilisateur renseigne le chemin vers celui-ci (attribut *path*). Par exemple, si deux stratégies alternatives de raffinement partagent une certaine logique, alors on l'isole dans un premier module *common*, et l'on définit les spécificités de chaque alternative dans les modules *strategy1* et *strategy2*. Ainsi, selon la stratégie choisie, l'étape de transformation sera constituée des modules (*common*, *strategy1*) ou (*common*, *strategy2*). Pour faciliter l'identification des modules à utiliser, ceux-ci sont répartis par catégorie en fonction de l'objectif du raffinement et de la dépendance (ou non) à une plate-forme donnée. Par ailleurs, l'**objectif n° 4** est la capacité à modifier la stratégie du processus de raffinement. Dans un tel cas, le processus ignore le dernier modèle raffiné (qui a été invalidé) pour appliquer un raffinement alternatif à un modèle antérieur.

Pour retrouver le modèle antérieur, chaque modèle raffiné issu d'une transformation est associé à un identifiant *outputModelIdentifier*. Chaque transformation spécifie alors un attribut *inputModelIdentifier* pour déterminer le modèle à raffiner. Si aucune des stratégies de raffinement n'a été validée pour un élément particulier de l'architecture, cela signifie soit que le processus est dans l'incapacité à produire un code qui respecte les contraintes du système, soit que la combinaison des choix de raffinements précédents ont conduit à une violation des contraintes. L'utilisateur va spécifier soit une étape de type *ErrorState* stoppant le processus en indiquant la cause, soit une autre chaîne de raffinement sur des éléments précédemment raffinés. Le processus conserve ainsi une trace des modèles produits pour modifier la stratégie courante en repartant d'un modèle intermédiaire. Une autre solution serait de conserver les différences entre chaque modèle intermédiaire et le modèle final afin d'économiser de la mémoire. Nous n'avons actuellement pas expérimenté cette solution. Par conséquent nous conservons l'intégralité de chaque modèle intermédiaire.

**AbstractAnalysis** Pour répondre à l'**objectif n° 3**, à chaque transformation va éventuellement succéder une analyse évaluant l'impact du raffinement sur les performances du système. Le type *AbstractAnalysis* modélise une étape d'analyse générique. Celle-ci se décline en *Analysis* pour une unique analyse associée à une transformation ou en *AnalysisSequence* pour une succession d'analyses : on distingue les sous-types *Conjunction* et *Disjunction* qui indiquent que l'ensemble des analyses doit être validé, ou au contraire qu'une unique analyse de la séquence doit être validée, pour que la transformation soit considérée valide. On connecte l'attribut *element* d'un objet *Transformation* à un objet héritant de *AbstractAnalysis*. Par exemple, après un raffinement de la spécification métier d'une tâche et une réévaluation de son WCET, on réalise une analyse d'ordonnancement pour évaluer l'impact de cette nouvelle estimation. Une étape d'analyse spécifie alors l'identifiant de l'outil de validation que l'on souhaite utiliser (attribut *method*). L'outil doit être préalablement enregistré dans le processus avec un identifiant unique. Il peut s'agir d'un outil externe existant (*e.g.* Cheddar, AADLInspector, Bound-T) ou interne au processus (analyse de WCET à partir de modèles). Chaque analyse fournit des résultats normalisés selon le méta-modèle illustré par la figure 5.3 dans le but d'assurer une intégration et une compatibilité au sein du processus.

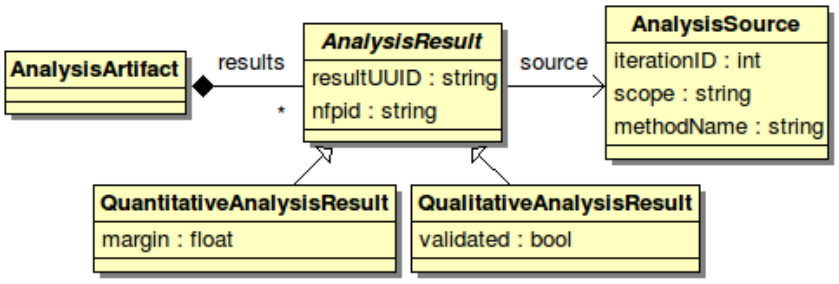



FIGURE 5.3 – Méta-modèle de *chaîne de raffinement* : normalisation des résultats d'analyse

L'ensemble des résultats d'une analyse sont modélisés par des objets de type *AnalysisResult* regroupés dans un objet de type *AnalysisArtifact*. On distingue deux types de résultat : qualitatif ou quantitatif. Dans le premier cas, il s'agit d'une réponse binaire à un test (le test a t-il réussi ?). Dans le second cas, le résultat indique une marge par rapport à une propriété particulière : par exemple, la marge entre l'échéance d'une tâche et son pire temps de réponse. Dans la première situation,

le résultat d'analyse suffit pour déterminer si le raffinement actuel est approprié. Dans la seconde, l'utilisateur va potentiellement décider si le raffinement est approprié en se basant sur les marges affichées. On spécifie alors pour chaque analyse, le mode d'exécution (automatique ou manuel) qui détermine si l'utilisateur doit intervenir dans le processus décisionnel. Afin de retracer chaque analyse, les résultats s'accompagnent d'un objet de type *AnalysisSource* qui indique notamment les éléments du modèle qui ont été analysés, avec quel outil/méthode (attribut *method*). Chaque analyse doit influencer le déroulement du processus afin de répondre à l'**objectif n° 4**. Pour cela, après chaque étape *AbstractAnalysis*, le processus se divise en deux branches. La branche sélectionnée dépend du résultat d'analyse (selon si l'analyse réussie ou échoue). Ces deux branches correspondent aux attributs *validOption* et *invalidOption* de l'objet *AbstractAnalysis*. Ainsi, dans le cas où l'analyse échoue, on modélise un changement de stratégie en reliant l'attribut *invalidOption* à une nouvelle étape de transformation. Dans le cas où l'analyse réussie, l'objet connecté à l'attribut *validOption* correspond alors à la prochaine étape du processus.

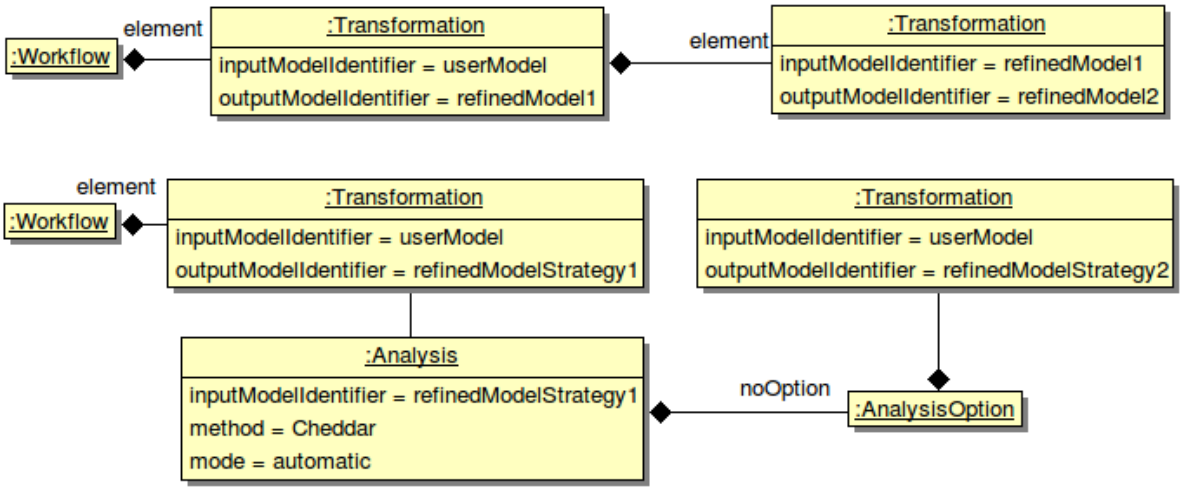



FIGURE 5.4 – Modélisation partielle de deux processus : l'un définit deux raffinements successifs, le second définit deux raffinements alternatifs

La modélisation de raffinements successifs ou alternatifs est déterminée par l'enchaînement d'étapes de type *Analysis* et *Transformation*. Cela est illustré par la figure 5.4. Celle-ci donne un exemple de deux processus. Celui du haut définit deux raffinements successifs. L'attribut *inputModelIdentifier* du second raffinement correspond à l'attribut *outputModelIdentifier* du premier. Cela signifie que le second raffinement s'applique sur le modèle raffiné issu du premier. Le processus du bas défini deux raffinements alternatifs. Après le premier raffinement, une analyse est effectuée. Si l'analyse échoue (branche *invalidOption*), un raffinement alternatif est réalisé. Dans ce cas, l'identifiant *inputModelIdentifier* des deux raffinements a la même valeur. Cela signifie que le modèle d'entrée des deux raffinements est le même (le premier modèle raffiné n'est pas pris en compte).

**Loop** En chaînant les étapes *Transformation* et *Analysis* on modélise ainsi le choix d'une stratégie parmi un ensemble de stratégies alternatives. Cependant, cette modélisation complexifie considérablement la modélisation du processus lorsque que celui-ci réalise des choix stratégiques sur

des éléments différents de l'architecture (*e.g.* communication entre tâches, communication entre noeuds, synchronisation). Pour améliorer la lisibilité du modèle, nous définissons le type *Loop* qui est équivalent à une chaîne d'étapes de raffinements alternatifs. Un objet de type *Loop* est associé à une liste de stratégies alternatives de raffinement (chacune étant modélisée par une liste de modules) qui sont évaluées les unes après les autres jusqu'à obtenir une stratégie valide. Toutes les stratégies associées à l'objet *Loop* sont validées par la même analyse associée à cet objet. La branche *validOption* de cette analyse correspond au cas où l'une des stratégies a été validée. Cet attribut indique la prochaine étape du processus dans ce cas. La branche *invalidOption* correspond au cas où aucune des stratégies alternatives n'a été validée : cela signifie qu'aucun raffinement n'est valide pour un élément de l'architecture. Cette branche est alors soit associée à un objet de type *ErrorState* soit à toute autre étape annulant un ensemble de raffinements précédents (en repartant d'un modèle intermédiaire).

**Serialization** La sérialisation est la sauvegarde d'un modèle raffiné en le traduisant en version textuelle. L'ensemble du modèle raffiné est sauvegardé afin d'éventuellement le réutiliser et l'analyser dans un autre processus, notamment pour la certification (**objectif n° 5**). L'étape de *Serialization* indique alors l'identifiant du modèle à sauvegarder (attribut *outputModelIdentifier*). L'utilisateur spécifie à quels moments du processus le modèle raffiné doit être sauvegardé. La sérialisation est explicitée par l'utilisateur. Ainsi, une étape *Serialization* succède à une étape de type *Transformation* ou *Analysis*.

**Generation** La génération de code est l'étape finale du processus et consiste à traduire le modèle raffiné en code exécutable (**objectif n° 6**). Le générateur est sélectionné en fonction de la plateforme d'exécution ciblée spécifiée à la configuration. Celui-ci traduit le modèle raffiné en langage cible et n'introduit aucune ressource supplémentaire non modélisée de manière à garantir la cohérence du processus. Chaque générateur agit de façon similaire à un *pretty-printer* et cela a déjà été expérimenté dans [59]. Dans notre cas, nous fournissons des générateurs de code pour des plates-formes ARINC653, POSIX, OSEK et Ada Ravenscar.

Nous avons introduit le méta-modèle qui modélise des *processus de raffinement incrémental*. En particulier, chaque étape de type *Transformation*, modélisant un raffinement, est définie à partir d'une liste de modules de transformations afin de réaliser *l'adaptation de la transformation* en répondant ainsi à la problématique de *l'adaptabilité du processus de génération* (abordée en section 5.2). Nous nous appuyons pour cela sur la technique existante de *superimposition* qui est introduite dans la section suivante.

### 5.1.3 Superimposition

Nous avons présenté précédemment le méta-modèle de *chaîne de raffinement* et en particulier les étapes de *Transformation* constituées d'un ensemble de modules de transformation afin de répondre à l'**objectif n° 2**. Le principe d'assemblage des modules correspond au mécanisme existant de *superimposition* [94] que nous présentons ici.

Une étape de raffinement/transformation est définie par une liste ordonnée de modules. Un module est une entité logique dans laquelle des règles de transformation sont définies. Afin de rem-

placer une règle d'une étape de raffinement, nous utilisons la technique de *superimposition* des langages comme ATL : si plusieurs règles ont la même définition (nom et type) au sein de modules différents, seule est chargée la définition du dernier module. Plus formellement, pour une étape de raffinement constituée des modules $(M, \{M_1, M_2, M_3\})$ avec $M$ le module principal et $M_1..M_3$ les modules superimposés, les règles définies dans $M$ sont remplacées par les règles de même définition dans $M_1$. Les règles définies dans $M \cup M_1$ sont remplacées par les règles de même définition dans $M_2$. Cette technique est illustrée par la figure 5.5. L'ensemble des noms de règles est *R1*, *R2*, *R3*, *R4*. Des règles de même noms sont présentes dans différents modules. La figure donne l'ensemble des règles issu de la *superimposition* de ces modules.

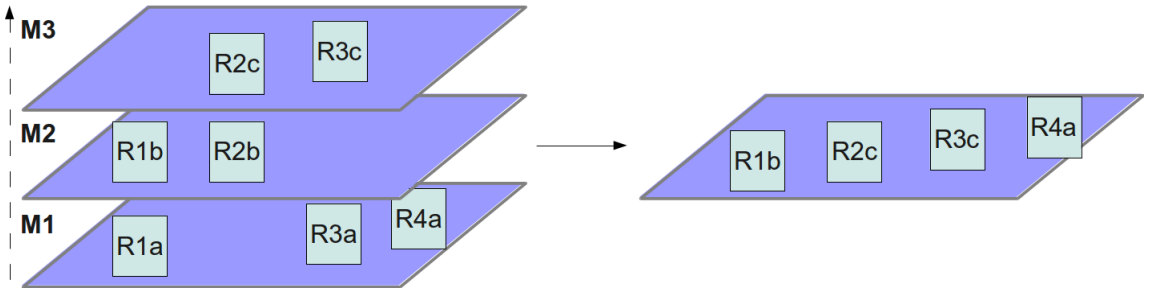



FIGURE 5.5 – Illustration de la *superimposition* sur trois modules M1, M2, M3

Cela contraint notre approche par la nécessité d'écrire ou de réécrire les règles de transformation en identifiant les parties pouvant évoluer selon le contexte. En effet, afin de bénéficier d'une règle existante que l'on souhaite altérer sans la réécrire en intégralité, il faut la scinder afin de redéfinir certaines parties et en conserver d'autres. La prochaine section fournit une méthode d'utilisation du méta-modèle.

### 5.1.4 Utilisation

Nous avons présenté globalement les étapes du processus. Nous donnons ici une marche à suivre pour faciliter la modélisation du processus. Premièrement, le modèle défini par l'utilisateur doit être validé avant de démarrer le processus. Après chaque raffinement on réalise une de nouveau une analyse pour déterminer le surcoût lié au raffinement.

1. **Chaîne de validation** Une ou plusieurs analyses sont réalisées sur le modèle actuel (modèle initial lors de la première itération). On crée un ensemble d'objets *Analysis* que l'on initialise avec notamment l'identifiant de l'outil d'analyse. Pour effectuer une chaîne d'analyse, on relie les objets *Analysis* en les reliant par leur attribut *validOption*. De cette façon, la chaîne d'analyse est interrompue si l'une des analyses échoue. Dans ce cas, l'attribut *invalidOption* de l'analyse ayant échouée détermine la prochaine étape. Pour la chaîne de validation préliminaire, on relie cet attribut à un objet *ErrorState* qui indique que le modèle initial est invalidé. Le processus stoppe dans ce cas.
2. **Chaîne de raffinement** Lorsque les analyses précédentes ont toutes réussies, le processus réalise un raffinement du modèle actuel. On rattache l'attribut *validOption* de la dernière analyse à un objet *Transformation*. L'attribut *inputModelIdentifier* de cet objet est le même que celui défini à l'étape précédente. Chaque étape de transformation étant constituée d'un

ensemble de modules, ceux-ci doivent être spécifiés. On crée alors un objet *ModuleList*, rattaché à la transformation, auquel on associe des objets de type *Module* qui correspondent aux modules de transformation que l'on va *superimposer*. L'attribut *path* de ces objets indique le chemin vers le module de transformation. On initialise l'attribut *outputModelIdentifier* de l'objet *Transformation* avec un identifiant arbitraire unique qui permettra de retrouver le modèle raffiné par la suite. Si l'on souhaite obtenir une trace du raffinement réalisé à cette étape, on associe l'attribut *element* de l'objet *Transformation* à un objet de type *Serialization*.

3. **Evaluation du raffinement et changement de stratégie** On valide le raffinement effectué à l'étape 2 en reproduisant la chaîne d'analyse de l'étape 1. Le dernier élément produit à l'étape précédente (objet *Transformation* ou *Serialization*) est connecté au premier objet *Analysis* de la nouvelle chaîne d'analyse. L'attribut *inputModelIdentifier* de ces analyses correspond alors à l'identifiant *outputModelIdentifier* du modèle raffiné issu de l'étape 2. En connectant l'objet *Transformation* à une chaîne d'analyse, cela a pour effet de stopper le processus si le modèle raffiné est invalidé. Cependant, si l'on souhaite appliquer une stratégie alternative lorsque l'analyse échoue, alors l'attribut *invalidOption* de chaque objet *Analysis* est rattaché à un autre objet *Transformation*. L'attribut *inputModelIdentifier* de cet objet référence l'identifiant d'un précédent modèle intermédiaire duquel on souhaite reprendre le processus. Dans ce cas on répète l'étape 2. Si le processus de raffinement n'est pas terminé, l'attribut *invalidOption* de la dernière analyse est rattaché à une nouvelle transformation, on répète également l'étape 2. Au contraire, si le processus de raffinement est terminé, on exécute l'étape 4.
4. **Génération de code exécutable** Lorsque la chaîne de raffinement est terminée et que le modèle raffiné est suffisamment proche d'un code exécutable, l'attribut *validOption* de la dernière analyse est rattaché à une étape *Generation* qui termine le processus en générant le code à partir du dernier modèle raffiné.

Nous avons présenté une méthode pour modéliser un processus de génération à l'aide du méta-modèle présenté précédemment. Nous donnons un exemple d'utilisation de ce processus dans la section suivante.

### 5.1.5 Exemple de modélisation de processus

Nous illustrons l'utilisation de ce méta-modèle en prenant comme exemple le générateur de code Ocarina [49]. Nous démontrons d'une part qu'il est possible de reproduire le processus d'Ocarina à l'aide du méta-modèle de la section 5.1.2. D'autre part, nous démontrons également que celui-ci peut être amélioré pour maîtriser l'impact du code généré et adapter la stratégie de génération en conséquence.

#### 5.1.5.1 Modélisation du processus initial

Comme introduit dans le chapitre 2, Ocarina est une suite d'outils de validation et de génération autour du langage de modélisation AADL et des intergiciels PolyORB-HI-C, PolyORB-HI-Ada et POK. Il fournit un processus qui valide les modèles AADL à l'aide d'outils comme Cheddar puis génère le code pour une plate-forme d'exécution sélectionnée si la validation réussie. On modélise ce processus avec notre méta-modèle comme illustré par la figure 5.6.

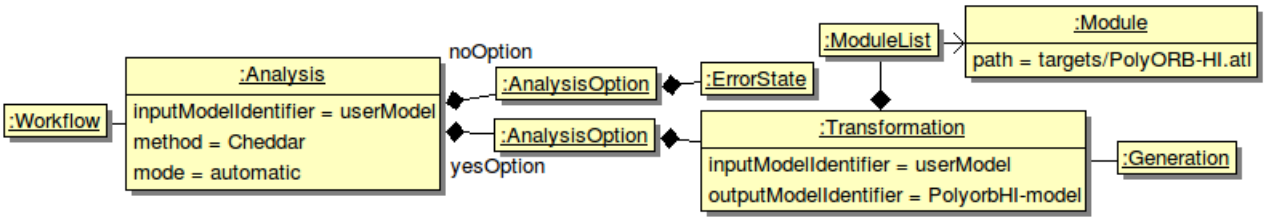



FIGURE 5.6 – Modélisation du processus initial de génération d'Ocarina

La première étape du processus est alors une phase d'analyse avec l'outil d'analyse d'ordonnancement Cheddar. On instancie la classe *Analysis* et on spécifie l'identifiant de l'outil Cheddar pour que celui-ci soit utilisé. On indique également que l'analyse s'applique sur le modèle initial, identifié par *userModel*. Ensuite, on définit la suite du processus dans le cas où l'analyse échoue et dans le cas où l'analyse réussie. Dans le premier cas, le processus stoppe en signalant que le système, tel qu'il a été modélisé par l'utilisateur, n'est pas ordonnançable. Dans le second cas, la modèle est considéré valide et on produit un modèle d'implémentation lors d'une étape *Transformation*. La transformation est décrite dans le fichier *targets/PolyORB-HI.atl*. Le modèle d'implémentation est ensuite traduit en code exécutable pour PolyORB-HI-C/PolyORB-HI-Ada.

#### 5.1.5.2 Maîtrise du coût engendré par la génération de code

Tel qu'il a été défini, le processus n'évalue pas l'écart de performances entre le modèle initial et le code généré. Si le modèle initial est validé, Ocarina produit le code exécutable mais ne garantit pas que les performances réelles sont équivalentes aux performances mesurées sur le modèle initial. Pour cela, il faut alors modifier le processus de génération d'Ocarina en conséquence. La figure 5.7 illustre la modification du processus.

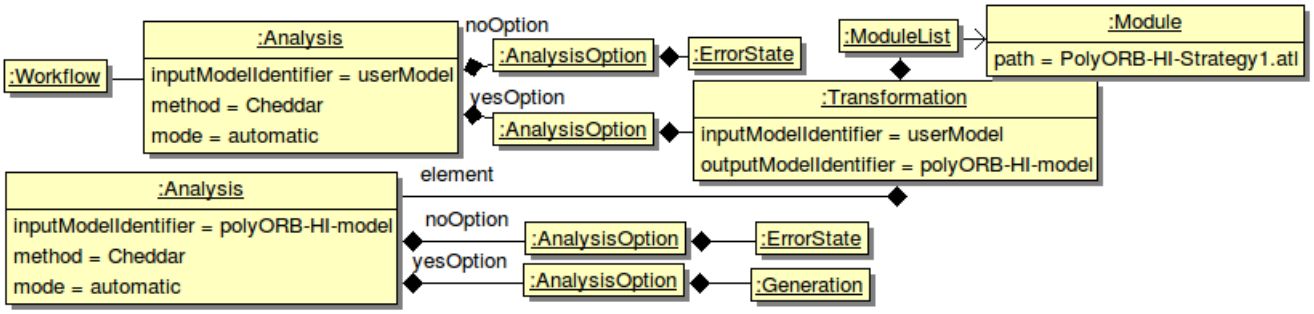



FIGURE 5.7 – Première amélioration du processus de génération d'Ocarina

Une étape d'analyse est insérée entre l'étape de raffinement et l'étape de génération de code. De cette façon, le code n'est généré que si l'analyse réussie sur le modèle d'implémentation $polyORB - HI - model$.

#### 5.1.5.3 Adaptation de la stratégie de génération

Si la seconde analyse échoue, il est souhaitable d'orienter le processus vers une implémentation alternative plutôt que de stopper la génération de code. Dans ce cas, dans la prévision où la première stratégie de raffinement échoue, on connecte à l'attribut *invalidOption* de la seconde analyse à une nouvelle étape de raffinement. Pour modéliser qu'il s'agit d'une alternative, l'attribut *inputModelIdentifier* de cette nouvelle étape correspond à l'identifiant *userModel* du modèle initial : le modèle raffiné *polyORB-HI-model* issu de la première transformation est ignoré. Le processus final est illustré par la figure 5.8.

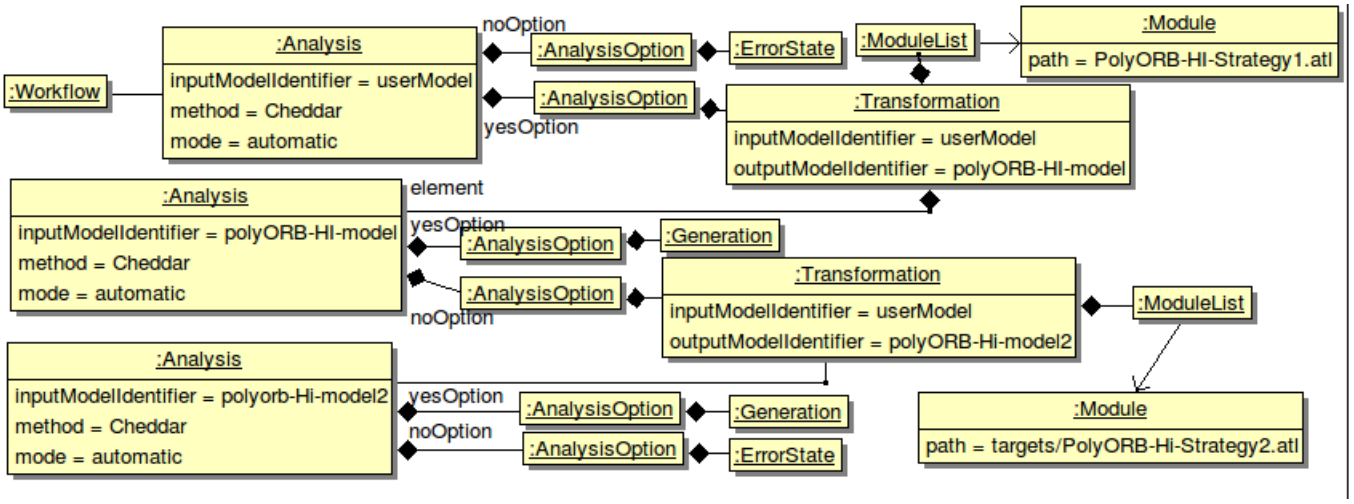



FIGURE 5.8 – Deuxième amélioration du processus de génération d'Ocarina

Cette deuxième modification du processus conduit donc à expliciter l'enchaînement de tentatives de raffinements sur le modèle initial. De plus, le même outil d'analyse est utilisé à chaque alternative. Par conséquent, on simplifie la modélisation de l'enchaînement des tentatives avec l'utilisation du type *Loop* comme illustré par la figure 5.9. L'objet *Loop* représente l'ensemble des tentatives de raffinement pour l'élément particulier de l'architecture que nous raffinons. Pour modéliser les stratégies alternatives de raffinement, on rattache l'attribut *alternative* de cet objet à l'ensemble des *ModuleList* qui définissent chaque alternative. On connecte également l'objet *Loop* à l'outil d'analyse Cheddar qui va évaluer chacune d'elles. Si l'une d'elles est validée, elle est appliquée, puis le code est généré à partir du modèle raffiné. Si aucune n'est validée, le processus entre dans un état d'erreur.

Si le processus de raffinement nécessite différentes phases de raffinements sur des éléments distincts de l'architecture, un second objet *Loop* est connecté par exemple à l'attribut *validOption* de l'analyse. Les attributs *inputModelIdentifier* et *outputModelIdentifier* identifient les modèles d'entrée et de sortie de l'étape *Loop*. Selon la valeur donnée à ces attributs, on modélise ainsi le fait que le processus poursuit une succession de raffinements ou abandonne des raffinements précédemment réalisés.

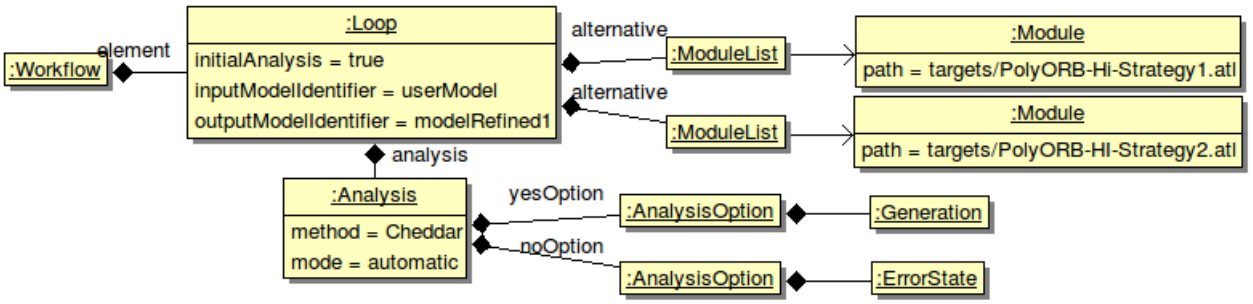



FIGURE 5.9 – Deuxième amélioration du processus de génération d'Ocarina : modèle simplifié avec l'utilisation d'un objet *Loop*

Nous avons décrit le méta-modèle de *chaîne de raffinement* , qui assure la mise en œuvre du *processus de raffinement incrémental* proposé en section 4.1, et nous l'avons illustré avec le générateur de code Ocarina. Ce méta-modèle s'appuie sur la technique de *superimposition* pour définir les transformations de façon modulaire en répondant ainsi à la problématique de l'*adaptabilité du processus de génération*. Nous avons également introduit des patrons de transformation au chapitre précédent qui répondent à cette problématique en utilisant pour certains la *superimposition*. La prochaine section donne une méthode d'utilisation de ces patrons et les illustre sur des exemples de transformation.

## 5.2 Patrons de transformation

Dans cette section nous décrivons l'implémentation des patrons de transformations introduits dans la section 4.2. Ceux-ci facilitent la mise en oeuvre du *processus de raffinement incrémental* proposé en section 5.1 en proposant des techniques pour adapter et dériver des stratégies de transformation. Nous passons en revue les quatre patrons proposés (*Stratégie, Substitution partielle, Adaptateur et Memento*) et nous les illustrons à travers le langage de transformation ATL.

### 5.2.1 Stratégie

Le patron *Stratégie* de l'orienté-objet consiste à déléguer à un objet tierce une partie du comportement de l'objet principal. On va ainsi modifier le comportement de l'objet principal en substituant l'objet tierce par un autre objet.

**Cas d'utilisation** Pour une règle de transformation, on souhaite altérer son comportement selon le contexte. Une partie du comportement de celle-ci est alors délégué à un objet tierce que l'on pourra remplacer par la suite pour modifier le comportement de cette règle. Il est donc nécessaire d'identifier les parties de la règle qui correspondent au comportement que l'on souhaite altérer. Dans un langage de transformation hybride tel que ATL, chaque règle est constituée de plusieurs clauses, en particulier la sélection des éléments d'entrée (clause *from*), la production des éléments de sortie (clause *to*) et les post-traitements appliqués après la création des éléments de sortie (clause *do*). Selon la clause sur laquelle *Stratégie* s'applique, l'objectif est d'altérer :

– La portée de la règle (clause *from*). Il s'agit alors du patron *Substitution partielle* dont l'objectif est de remplacer la règle actuelle par une nouvelle règle seulement sur un sous-ensemble des éléments d'entrée. Ce patron est présenté dans la prochaine section.
– La logique d'initialisation des éléments de sortie (clause *to*). La modification de la règle consiste à produire les mêmes éléments de sortie mais à modifier la façon dont leurs attributs sont initialisés à partir des éléments d'entrée. L'expression d'initialisation correspond par exemple à une expression arithmétique que l'on souhaite altérer dans certaines situations, ou encore l'appel à une règle secondaire.
– Les post-traitements (clause *do*). Il s'agit de modifier les traitements appliqués une fois les éléments de sortie crées et initialisés. Par exemple, on souhaite appliquer des post-conditions particulières pour certaines configurations ou certaines architectures. On va donc appliquer le patron *Stratégie* pour appliquer des vérifications particulières dans certains cas uniquement.

**Méthodologie** Le principe de *Stratégie* consiste à externaliser les expressions de la règle (à l'intérieur de la clause *to* ou *do*) que l'on souhaite remplacer par la suite. On externalise ces expressions dans des fonctions annexes que l'on définit au sein du module de transformation. Par *superimposition*, on ajoute un second module de transformation qui redéfinit ces fonctions annexes. La nouvelle transformation est alors constituée du module initial et du module *superimposé*. La règle initiale est donc modifiée lorsque le second module est ajouté au premier.

Lorsque l'on souhaite altérer l'initialisation d'un objet cible (clause *to*), la marche à suivre est la suivante :

1. Une fonction annexe doit être définie pour encapsuler l'expression d'initialisation. (*helper* en ATL).
2. La fonction annexe est associée au type de l'objet crée (mot-clé *context* associé au helper en ATL) si l'on souhaite définir plusieurs fonctions de même noms sur des types différents.
3. Un second module redéfinit la déclaration de la fonction annexe avec la nouvelle expression d'initialisation. La nouvelle fonction doit posséder la même signature (nom, paramètres et type associé).

L'application de *Stratégie* dans le cas des post-traitements (clause *do*) est similaire au cas précédent. Le post-traitement correspond à une règle exécutée par un appel explicite (*called rules* et *lazy rules* en ATL). Si l'on souhaite substituer un post-traitement par un autre selon le contexte, celui-ci doit être encapsulé dans une fonction annexe, selon le procédé précédent.

**Exemple** Le listing 5.1 définit une règle *R* prenant en entrée deux objets *a* et *b* respectivement de type *A* et *B*. On souhaite modifier le comportement de cette règle afin que l'objet *c* soit initialisé avec le produit de deux attributs plutôt que leur somme.

```
1 rule R {
2   -- selection section
3   from
4         a : MM!A,
5         b : MM!B,
6   -- creation section
7   to
8         c : MM!C (attr <- a.attrX + b.attrY),
```

```
9           d : MM!D (ref <- c)
10  }
```

Listing 5.1 – Exemple de règle ATL que l'on souhaite dériver : la règle dérivée doit initialiser l'objet c avec le produit des deux attributs au lieu de leur somme

Pour éviter une réécriture complète de la règle, on délègue l'expression concernée dans une fonction annexe que l'on pourra remplacer par la suite. Dans notre exemple, on va délèguer l'affectation de l'attribut *attr* de *c* à une fonction annexe que l'on nommera *computeAttrFrom*. Le listing 5.2 illustre cette modification.

```
1   rule R {
2     -- selection section
3     from
4           a : MM!A,
5           b : MM!B,
6     -- creation section
7     to
8           c : MM!C (attr <- c.computeAttrFrom(a,b)),
9           d : MM!D (ref <- c)
10  }
11  helper context MM!C def : computeAttrFrom(a : MM!A, b : MM!B) : Integer =
12          a.attrX + b.attrY
13  ;
```

Listing 5.2 – Réécriture d'une règle en prévision de l'application du patron *Stratégie*

Ainsi, il est maintenant possible de *superimposer* un module ATL supplémentaire afin de modifier l'initialisation de l'attribut en remplaçant la fonction existante. Ce module *superimposé* contient uniquement la nouvelle définition de cette fonction (figure 5.3).

```
1   helper context MM!C def : computeAttrFrom(a : MM!A, b : MM!B) : Integer =
2           a.attrX * b.attrY
3   ;
```

Listing 5.3 – Utilisation du patron *Stratégie* pour altérer l'initialisation des attributs

La section suivante aborde le patron *Substitution partielle* dont la mise en œuvre est similaire mais répond à des objectifs différents.

### 5.2.2 Substitution partielle

**Cas d'utilisation** Ce patron vise à assurer la compatibilité d'une transformation sur des plates-formes distinctes qui ont des sémantiques d'exécution différentes et introduisent des éléments d'implémentation particuliers. Certains concepts sont donc spécifiques à certaines plates-formes. Pour d'autres plates-formes, ces concepts particuliers peuvent être assimilés à des concepts plus généraux et ont ainsi une sémantique différente. Par conséquent, ceux-ci sont traduits différemment d'une plate-forme à l'autre. Si l'on dispose de deux générateurs de code, l'un assimilant des concepts qui sont différenciés par le second, on peut souhaiter factoriser la traduction du sous-ensemble de concepts ayant la même sémantique dans les deux cas. Pour réutiliser la traduction

faite par le premier générateur de code, il faut cependant limiter son application au sous-ensemble commun et non pas à l'ensemble initial afin d'appliquer une seconde traduction pour le sous-ensemble complémentaire spécifique au second générateur.

La mise en place de ce patron consiste à modifier la portée d'une transformation existante afin de la limiter à un sous-ensemble des éléments initiaux, dans le but d'appliquer un traitement différent au sous-ensemble complémentaire.

**Méthodologie** L'application du patron *Substitution partielle* utilise le patron *Stratégie*. Soient :
– $R$ la règle initiale dont on souhaite limiter le champ d'application.
– $S$ l'ensemble des éléments d'entrée de $R$.
– $M$ le module contenant $R$.
– $R2$ la nouvelle règle remplaçant $R$ sur un sous-ensemble $S2 \in S$ des éléments d'entrée de $R$.

1. La sélection des éléments d'entrée $S$ de la règle $R$ est déléguée à une fonction annexe $H$. On applique alors le patron *Stratégie*. La règle $R$ est réécrite en remplaçant l'expression $S$ par un appel à la fonction $H$.
2. La prise en compte de la nouvelle plate-forme d'exécution requiert l'utilisation d'un second module $M'$ *superimposé* à $M$. La règle $R2$ contenue dans $M'$ identifie les éléments d'entrée $S2$. $M'$ redéfinit la fonction annexe $H$ afin de restreindre l'application de $R$. La nouvelle définition de $H$ identifie alors l'ensemble $S1$ complémentaire à $S2$ dans $S$.

**Exemple** Soit une règle existante $R$ qui transforme les objets de type A en objets de type B. Cette règle est définie pour une plate-forme d'exécution $PF_1$ qui assimile tous les objets de type A à une unique sémantique d'exécution. Ils sont donc tous traduits en objets B. Une seconde plate-forme $PF_2$ distingue cependant des sous-ensembles d'objets de type A qui vont avoir une traduction différente selon leur initialisation : les objets $A$ dont l'attribut *value* est positif sont assimilés à la même traduction que l'ensemble des objets $A$ correspondant à la plate-forme $PF_1$. Cependant, les objets $A$ dont l'attribut *value* est négatif ont une traduction différente spécifique à la plate-forme $PF_2$ et doivent être traduits différement. On souhaite réutiliser la règle $R$ pour la seconde plate-forme $PF_2$ car elle réalise une traduction similaire. Cependant, cette règle s'applique sur l'ensemble des objets de type $A$. Pour la réutiliser, il faut donc modifier son champ d'application pour ne concerner que les objets de type $A$ dont l'attribut *value* est positif.

1. La règle $R$ est réécrite afin d'externaliser, dans une fonction annexe *isRefinableByRuleR*, l'expression identifiant l'ensemble des éléments de type $A$ qui doivent être traités par $R$. Par défaut, tous les éléments de type $A$ sont traduits par $R$ pour la plate-forme $PF_1$. Le listing 5.4 illustre cette première modification.

```
1  rule R {
2    from
3         a: MM!A (a.isRefinableByRuleR())
4    to
5         b: MM!B
6    do
7    {
8              ...
9    }
10 }
```

```
11 helper context MM!A def : isRefinableByRuleR() : Boolean =
12       true
13 ;
```

Listing 5.4 – Réécriture d'une règle pour l'application du patron *Substitution partielle*

2. Le module *superimposé* définit la règle *R2* qui traite les objets de type *A* dont l'attribut *value* est négatif. Ce second module limite la portée de la règle existante *R* sur l'ensemble des éléments dont l'attribut *value* est positif en redéfinissant la fonction annexe *isRefinableByRuleR*. La portée de la règle *R*2 est alors la négation de l'expression *isRefinableByRuleR*.

```
1  -- Reduce scope of rule R
2  helper context MM!A def : isRefinableByRuleR() : Boolean =
3          self.value >= 0
4  ;
5
6  -- Add rule for complementary set
7  rule R2 {
8    from
9          a: MM!A (not
10                   a.isRefinableByRuleR())
11   to
12         ...
13 }
```

Listing 5.5 – Module *superimposé* appliquant le patron *Substitution partielle*

Le patron *Substitution partielle* assure ainsi la compatibilité d'une transformation sur des plateformes d'exécution distinctes. Le patron suivant, *Adaptateur*, répond au problème de compatibilité des éléments transformés en homogénéisant leur interfaces.

### 5.2.3 Adaptateur

Le patron *Adaptateur* de l'orienté-objet consiste à intégrer dans une architecture logicielle des classes externes qui n'ont pas été conçues pour ce type d'architecture. De nouvelles classes, compatibles avec l'architecture, réalisent alors l'adaptation de ces classes externes afin de les intégrer et les utiliser.

**Cas d'utilisation** Pour un langage de transformation, l'objectif est de rendre réutilisable une règle de transformation pour un ensemble d'éléments de types différents. Chaque règle initialise ses éléments de sortie et leurs attributs à partir des attributs ou des fonctions fournis par les éléments d'entrée. L'écriture d'une règle de transformation est donc dépendante du type des objets pris en charge et de leur interface. Par conséquent, pour deux types d'objets d'entrée ayant des interfaces suffisamment différentes, deux règles distinctes seront nécessaires afin d'adapter l'initialisation des objets de sortie, même si les logiques de transformation sont similaires. Les incompatibilités d'interface des objets conduisent donc à écrire différentes règles de transformation. Cela complique la maintenance du processus de transformation, en imposant l'utilisation de plusieurs règles ayant une logique commune.

L'objectif du patron *Adaptateur* est alors de rendre homogène les interfaces de ces objets afin de réutiliser la même règle de transformation sur ces différents objets.

**Méthodologie** L'application du patron *Adaptateur* se fait en trois temps :

1. L'objectif étant de factoriser la production des éléments de sortie, les expressions d'initialisation doivent être identiques d'une règle à l'autre. On définit alors un ensemble de fonctions annexes associées aux types concernés afin de les rendre virtuellement homogènes (les fonctions appelées lors de l'initialisation devant être les mêmes).
2. On définit une nouvelle règle abstraite qui réalise les transformations communes aux deux types. Les interfaces étant rendues virtuellement compatibles, on utilise les mêmes expressions d'initialisation. Le type des éléments d'entrée doit être un type parent commun aux deux types. L'interface du type parent n'est pas nécessairement compatible avec celle des types hérités, le typage étant vérifié à l'exécution. Par rapport à l'orienté-objet, cette règle abstraite correspond à une classe mère qui factorise la logique commune de ses classes filles.
3. Les règles initiales sont réécrites de façon à hériter de la règle précédente. Ainsi, les règles partagent une partie de leur transformation et peuvent ajouter la production d'éléments spécifiques. De plus, pour ne s'appliquer qu'au sous-type concerné, chaque règle doit raffiner le type des éléments d'entrée.

**Exemple** Soient deux règles *R*1 et *R*2 produisant toutes les deux des objets de type *B*. Ces deux règles s'appliquent sur des objets de types différents *A* et *C* ayant des interfaces différentes. L'initialisation de l'objet *B* est donc différente dans les deux cas. On souhaite cependant factoriser ces deux règles car elles produisent les mêmes éléments. On applique alors le patron *Adaptateur* pour rendre les interfaces des types *A* et *B* compatibles afin d'utiliser une règle commune.

```
1  rule R1 {
2    from
3          a: MM!A
4    to
5          b: MM!B (x <- a.z * a.t),
6          ...
7  }
8  rule R2 {
9    from
10         c: MM!C
11   to
12         b: MM!B (x <- c.x),
13         ...
14 }
```

Listing 5.6 – Deux règles illustrant le cas d'utilisation du patron *Adaptateur*

1. *R*1 et *R*2 divergent sur l'initialisation de l'attribut *x* de l'objet *b*. On crée alors des fonctions annexes associées aux types *A* et *C* afin d'obtenir la même expression d'initialisation de l'attribut *x*.

```
1  helper context MM!A def : getX() : Integer = self.z * self.t;
2
3  helper context MM!C def : getX() : Integer = self.x;
```

Listing 5.7 – Adaptation des interfaces en définissant des fonctions annexes

2. Les objets de type *A* et *C* peuvent ainsi être traités par la même règle en initialisant l'attribut *x* de *b* en appelant la fonction *getX*(). On réécrit la règle initiale *R*1 afin de la rendre réutilisable pour le type *R*2. On la redéfinit comme étant une règle abstraite dont le type des objets d'entrée est un type parent à *A* et *B*. Ici, le type parent est *O*. On note que l'interface de *O* n'a pas la nécessité d'être compatible, le typage étant vérifié à l'exécution.

```
1 abstract rule R1_or_R2 {
2   from
3         o: MM!O
4   to
5         b: MM!B (x <- o.getX()),
6         ...
7 }
```

Listing 5.8 – Application du patron *Adaptateur* en réécrivant la règle initiale

3. On redéfinit ensuite les deux règles *R*1 et *R*2 en les faisant hériter de la règle précédente. On a alors factorisé la logique de transformation commune à *R*1 et *R*2 malgré les interfaces incompatibles des éléments d'entrée. Pour un exemple plus complexe, considérons que *R*1 et *R*2 font chacune 20 lignes de code, et que parmi ces lignes 16 correspondent à une même logique malgré des syntaxes différentes dues aux interfaces hétérogènes des éléments d'entrée. L'application du patron rend possible la factorisation des 16 lignes en homogénéisant la syntaxe dans les deux cas grâce à des méthodes virtuelles rattachées types. La règle abstraite *R*1_*or*_*R*2 factorise ainsi ces 16 lignes. Ensuite, en faisant hériter *R*1 et *R*2 de cette nouvelle règle, ces dernières sont considérablement réduites.

```
1 rule R1 extends R1_or_R2 {
2   from
3         o: MM!A
4 }
5 rule R2 extends R1_or_R2 {
6   from
7         o: MM!C
8 }
```

Listing 5.9 – Application du patron *Adaptateur* en dérivant les règles de la règle initiale

Le patron *Adaptateur* facilite ainsi la factorisation du code de transformation sur des types d'interfaces incompatibles. Les trois patrons présentés facilitent l'intégration de nouvelles stratégies, ainsi que la maintenance du processus de raffinement incrémental. Le patron suivant, *Memento*, répond au besoin de certification du processus.

### 5.2.4 Memento

Dans l'orienté-objet, le patron *Memento* permet d'annuler une action en restaurant l'état précédent d'un objet. On sauvegarde alors l'état courant de l'objet avant modification afin de retourner dans cet état si la modification doit être annulée.

**Cas d'utilisation** Dans le contexte d'un processus de transformation, un cas d'utilisation de ce patron est l'exploration de solutions de raffinements. Il s'agit de garder trace des relations entre les éléments sources et les éléments cibles. Ces traces sont ensuite analysées pour étudier l'impact

de chaque élément source sur le modèle cible (*e.g.* ressources générées) et donc sur les propriétés du système. De plus, lorsque plusieurs transformations sont chaînées, il est important de retrouver l'ensemble des éléments du modèle source ayant aboutis à la production des éléments finaux. L'application du patron *Memento* consiste alors à produire des traces qui aideront l'utilisateur à retrouver les correspondances entre le modèle initial et le modèle cible. Cela renforcera par ailleurs la compréhension du processus de raffinement et vérifiera que les spécifications, et donc les exigences, sont correctement traduites dans le modèle final.

**Méthodologie** La réalisation de ce patron consiste à modifier chaque règle de transformation de façon à produire des éléments supplémentaires qui enregistrent la liste des éléments d'entrée et la liste des éléments de sortie. En se basant sur le méta-modèle de traçabilité fourni en section 4.2, on produit un ensemble d'éléments de type *ElementTrace*. Pour chaque règle, on crée un élément de type *ElementTrace* et on initialise ses différents attributs :
– *ruleName* est initialisé avec le nom de la règle,
– *From_* est initialisé avec la liste des éléments sources,
– *To_* est initialisé avec la liste des éléments cibles

**Exemple** Le listing 5.10 illustre ce patron. Chaque élément d'entrée *a* de la règle *E* donne lieu à la production d'un élément *b*. La règle produit également un élément *trace* qui enregistre la trace de la transformation actuelle. Après la transformation, l'utilisateur pourra alors déterminer qu'il s'agit de l'objet *a* qui a donné lieu à la production de l'objet *b* par l'exécution de la règle *R*.

```
1  rule R {
2     from
3          a : MM!A
4     to
5          b : MM!B (x <- a.getX()),
6
7          trace : Trace!TraceLink (
8              ruleName <- 'R',
9              sourceEntities <- Sequence {a},
10             targetEntities <- Sequence {b}
11         )
12 }
```

Listing 5.10 – Utilisation du patron Memento

Nous avons décrit la mise en œuvre des patrons de transformation introduits en section 4.2. Ceux-ci facilitent la mise en oeuvre du processus de raffinement incrémental présenté en section 4.1 en proposant des mécanismes pour altérer le comportement d'une transformation afin de répondre à la problématique de *l'adaptabilité du processus de génération* (section 3.2). Par ailleurs, face aux limitations imposées par la plate-forme d'exécution, l'utilisation de patrons n'est pas suffisant pour assurer la maîtrise du coût du code généré (problématique de *l'adaptabilité du code généré*). Nous avons ainsi proposé une *génération de composants intergiciels adaptés*. Nous abordons cette troisième contribution dans la prochaine section.

## 5.3 Génération de composants de communication adaptés

Face à la problématique de *l'adaptabilité du code généré*, nous avons proposé une *génération de composants intergiciels adaptés* (section 4.3). En particulier, nous avons présenté des mises en œuvre alternatives des communications qui pouvaient être utilisées dans certaines situations pour maîtriser le coût du code généré sur les performances du système. Dans cette section nous détaillons cette démarche autour des composants intergiciels de communication entre tâches. Premièrement, nous présentons brièvement ces composants intergiciels. Ensuite, nous décrivons le principe du processus de raffinement des communications pour faciliter la prise en compte de leur coût. Enfin, nous décrivons plus en détails le modèle de communication sans verrou que nous avons introduit précédemment.

### 5.3.1 Composants standard de communication

L'adaptation du code généré est motivé par le besoin d'optimiser son coût selon les caractéristiques du système modélisé. Cependant, la plate-forme d'exécution impose l'utilisation de composants qui ne sont pas toujours adaptés et peuvent conduire à de mauvaises performances dans certaines situations. Nous proposons de générer ces composants et de les adapter selon le contexte, tout en prenant en compte leur coût. Nous illustrons notre approche autour du langage de modélisation AADL, standard dans le domaine des SETRC.

Pour prendre en compte les composants intergiciels lors de la validation du système modélisé, il est nécessaire de les introduire au sein du modèle raffiné. Pour modéliser l'implémentation des communications, le langage AADL définit un ensemble de composants de type *subprogram* qui représentent les composants standard de communication :
– *Put_Value* : ajout d'un message dans une file.
– *Send_Output* : mise à disposition du destinataire les messages précédement ajoutés.
– *Receive_Input* : gel d'un ou plusieurs ports d'entrée. Les messages transmis après le *Receive_Input* sont ignorés jusqu'au prochain appel à cette fonction.
– *Next_Value* : retourne le prochain message contenu dans la file.

L'implémentation concrète de ces composants est dépendante de la plate-forme d'exécution. Par conséquent, chacun va exister dans plusieurs versions. Leur prise en compte nécessite alors de les introduire lors du raffinement du modèle initial (un modèle AADL dans notre cas). Une fois le modèle raffiné, celui-ci est évalué pour déterminer l'impact des composants choisis sur les performances du système. Nous décrivons ce raffinement dans la prochaine section.

### 5.3.2 Raffinement simple des communications et prise en compte de leur coût

Pour prendre en compte le coût des communications, nous procédons à un raffinement du modèle initial afin d'intégrer les composants de communication introduits dans la section précédente. Nous illustrons ce raffinement à l'aide de la notation graphique d'AADL décrite par la figure 5.10.

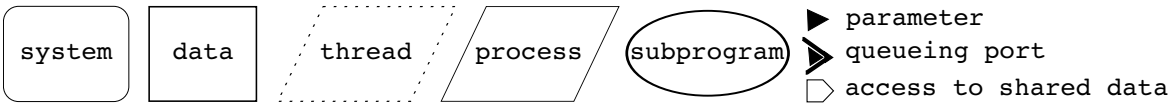



FIGURE 5.10 – Notation graphique d'AADL

La figure 5.11 est un modèle AADL défini par l'utilisateur qui décrit une communication asynchrone entre la tâche émettrice *T_Sender* et la tâche réceptrice *T_Receiver*. *T_Sender* exécute une fonction métier *Compute_Data*, implémentée par l'utilisateur, et envoie la valeur de retour au port *datain* de *T_Receiver*. Ici le port *datain* représente une file de messages.

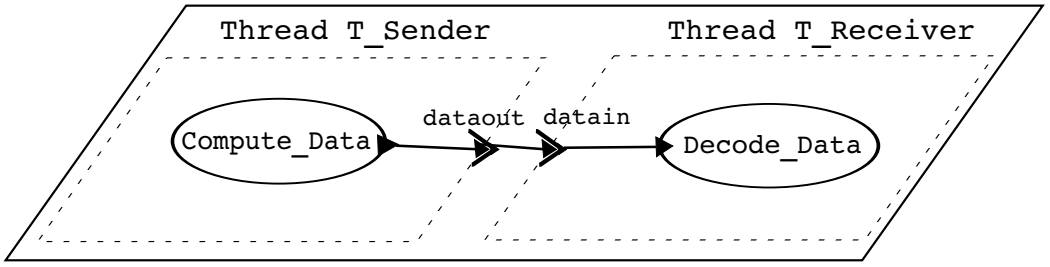



FIGURE 5.11 – Modèle initial défini par l'utilisateur : communication asynchrone entre deux tâches

Le raffinement de ce modèle consiste à introduire les composants de communication précédents. La figure 5.12 correspond au modèle AADL raffiné. La connexion entre ports est remplacée à un accès à une variable partagée *T_Receiver_datain* qui représente le port *datain* de la tâche *T_Receiver*. Le type de *T_Receiver_datain* est dépendant de l'implémentation choisie (*e.g.* tableau, liste chaînée). Le composant *T_Receiver_datain* est annoté de façon à préciser la structure utilisée. Des appels aux composants *Put_Value* et *Send_Output* sont ajoutés à la tâche émettrice *T_Sender*. *Put_Value* prend en entrée la donnée produite par *Compute_Data* et l'insère dans la file *T_Receiver_datain*. Une fois toutes les insertions réalisées (ici une seule), l'appel à *Send_Output* met à disposition du destinataire les messages ajoutés. Les composants *Next_Value* et *Receive_Input* réalisent les opérations inverses et sont ajoutées à la tâche *T_Receiver*.

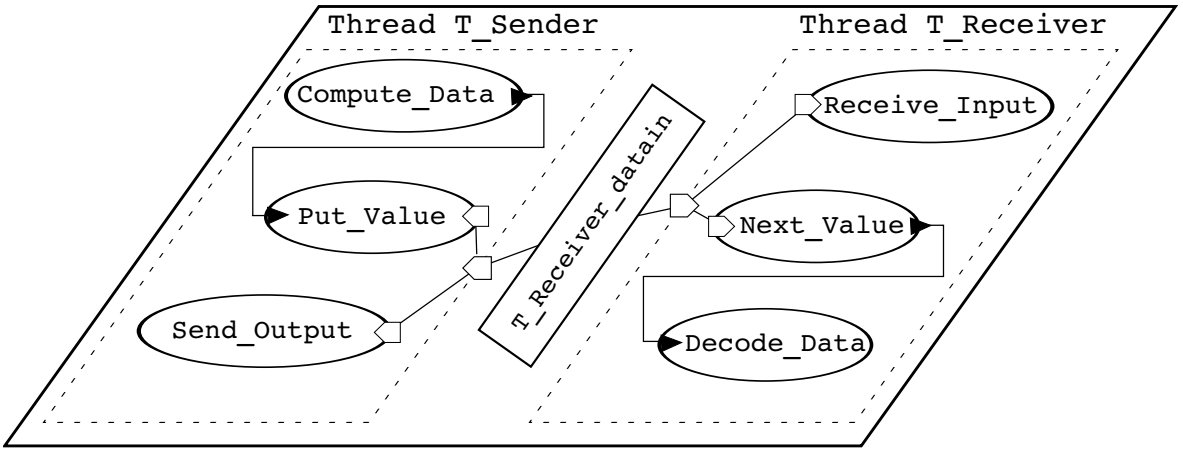



FIGURE 5.12 – Modèle raffiné introduisant les composants de communication

Le raffinement réalisé dans cet exemple va potentiellement varier selon l'implémentation. Par exemple, l'implémentation de *Send_Output* peut prendre en paramètre une unique file de message ou une liste de files de messages. Lorsque la tâche émettrice écrit sur plusieurs ports, il faut réaliser dans le premier cas plusieurs appels à ce composant alors qu'un seul est nécessaire dans le second cas. La variabilité du processus de raffinement n'est donc pas seulement le choix de la version des composants à utiliser, mais également le nombre d'appels, ou encore le nombre et le type des paramètres d'entrée/sortie. En effet, les composants standard de communication AADL sont partiellement définis et ne précisent pas par exemple la modélisation des files de messages ou la prise en compte de paramètres supplémentaires nécessaires aux opérations réalisées par ces composants.

Cette variabilité a été illustré sur l'implémentation des communications différées pour répondre à *l'adaptabilité du code généré*. Des implémentations alternatives ont été introduites en section 4.3. Pour évaluer le coût de chacune, le processus de raffinement explicite les ressources mises en œuvre (*e.g. données, composants intergiciels*). Ainsi, l'une ou l'autre des alternatives est sélectionnée selon les ressource mises en œuvre et les propriétés du système. Pour cela, on doit disposer pour chaque alternative d'un processus de raffinement ainsi qu'un ensemble de composants intergiciels. La section suivante présente plus en détails ces composants.

### 5.3.3 Raffinement détaillé des communications

Dans le chapitre précédent, nous avons illustré trois implémentations différentes des fonctions de communication différée : sans verrou avec indices calculés à l'exécution (*PDC_Indices*), sans verrou avec indices pré-calculés et stockés dans des tableaux (*PDC_Indices_Array*), ou bien insertion triée dans une liste chaînée protégée par un verrou (*PDC_Insertion_Sort*). On décrit chacun de ces modèles à l'aide d'un ensemble de composants AADL de type *subprogram* qui sont décrit à l'aide par exemple de l'annexe comportementale d'AADL, comme illustré par les listings 5.11, 5.12 et 5.13.

Le premier listing décrit l'implémentation du composant *Put_Value* avec le calcul d'indice à l'exécution. Les éléments déclarés dans la section *features* sont les paramètres d'entrée/sortie de la fonction *Put_Value*. Pour le calcul d'indice, la fonction a besoin notamment des propriétés temporelles de l'ensemble du jeu de tâche et de l'identifiant de la tâche courante. Ces paramètres correspondent aux *features context* et *taskID*. L'implémentation du service est spécifiée à l'aide de l'annexe comportementale (section *behavior_specification*). Le service exécute en premier la fonction *Compute_CDW* (*CDW : Current Deadline Write*) qui réalise le calcul d'indice d'écriture. L'indice calculé est retourné dans la variable *CDWIndex* déclarée dans l'annexe comportementale. Ensuite, la donnée *value* est écrite à l'emplacement *CDWIndex* du tableau *buffer* modélisant la file de message du consommateur. Le type de la donnée *value* est dépendant de l'architecture modélisée. Il n'est donc pas connu statiquement. On le déclare alors dans la section *prototypes* (*t_value*) pour indiquer qu'il devra être précisé lorsque le composant *Put_Value* sera intégré à l'architecture.

```
1 subprogram Put_Value
2   prototypes
3     t_value: data;
4     t_context : data;
5   features
```

```
6       taskID : in parameter Base_Types::Integer_16;
7       context : requires data access t_context;
8       IterationCounter: in parameter Base_Types::Integer_16;
9       value: in parameter t_value;
10      buffer: requires data access ArrayDataType {Access_Right=>write_only;};
11    annex behavior_specification {**
12      states  s: initial final state;
13      variables  CDWIndex : Base_Types::Integer_16;
14      transitions
15        t: s -[]-> s {
16          -- CDWIndex : out parameter
17          Compute_CDW!(taskID, IterationCounter, context, CDWIndex);
18          buffer[CDWIndex] := value
19        };
20    **};
21  end Put_Value;
```

Listing 5.11 – Composant Put_Value selon l'implémentation PDC_Indices

Le listing 5.12 décrit *Put_Value* pour la seconde implémentation (modèle PDC_Indices_Array). Ici, les indices sont calculés hors-ligne et stockés dans un tableau *CDW*. La fonction écrit alors à l'indice $CDW[IterationCounter]$ du tableau $buffer$, $IterationCounter$ étant le compteur de périodes de la tâche courante. Par rapport à la première version, cette deuxième version du composant *Put_Value* utilise des paramètres différents notamment le tableau *CDW* stockant les indices.

```
1   subprogram Put_Value
2     prototypes
3       t_value: data;
4     features
5       IterationCounter: in parameter Base_Types::Integer_16;
6       CDW: in parameter ArrayDataType;
7       value: in parameter t_value;
8       buffer: requires data access ArrayDataType {Access_Right=>write_only;};
9     annex behavior_specification {**
10      states  s: initial final state;
11      transitions
12        t: s -[]-> s {
13          buffer[CDW[IterationCounter]] := value
14        };
15    **};
16  end Put_Value;
```

Listing 5.12 – Composant Put_Value selon l'implémentation PDC_Indices_Array

Le listing 5.13 fournit la description du composant *Put_Value* pour la troisième implémentation (modèle PDC_InsertionSort). Ce composant réalise une insertion triée sur le port, modélisé par une liste chaînée et protégé par un verrou à l'aide des fonctions *Get* et *Release*. Ces dernières doivent être fournis au composant pour qu'il s'adapte aux différentes plates-formes et mécanismes d'exclusion mutuelle (*e.g.* Mutex POSIX, Semaphores ARINC653). Par conséquent, les fonctions *Get* et *Release* sont déclarées dans la section *prototypes*. L'annexe comportementale décrit cette troisième implémentation : l'appel à la fonction *findNextLink* détermine à quel maillon doit être rattaché le maillon à insérer, les maillons étant ordonnés par date (paramètre *currentDeadline*) et par priorité (paramètre *uniqueIndex*) en cas de dates égales. Par conséquent, ces paramètres supplémentaires *currentDeadline* et *uniqueIndex* sont rajoutés dans la section *features*. Ensuite,

la fonction *findFreeLink* trouve un maillon libre de la liste chaînée. Ce dernier est réutilisé. Il est initialisé avec la donnée et connecté au maillon précédent.

```
1  subprogram Put_Value
2    prototypes
3      t_value: data;
4      Get : subprogram Get_Resource;
5      Release : subprogram Release_Resource;
6    features
7      value: in parameter t_value;
8      currentDeadline : in parameter Base_Types::Integer;
9      uniqueIndex : in parameter Base_Types::Integer;
10     buffer: requires data access LinkedList {Access_Right=>write_only;};
11   annex behavior_specification {**
12     states
13       s: initial final state;
14     variables
15       freeLink : LinkedListLink;
16       nextLink : LinkedListLink;
17     transitions
18       t: s -[]-> s {
19          Get!(buffer);
20          findNextLink!(buffer,currentDeadline,uniqueIndex,nextLink);
21          findFreeLink!(buffer,freeLink);
22          if (freeLink.IS_AVAILABLE) {
23             freeLink.value := value;
24             freeLink.timestamp := currentDeadline;
25             freeLink.uniqueIndex := uniqueIndex;
26             moveLink!(freeLink,nextLink)
27          };
28          Rel!(buffer)
29       };
30   **};
31 end Put_Value;
```

Listing 5.13 – Composant Put_Value selon l'implémentation PDC_InsertionSort

La modélisation des composants intergiciels rend possible leur analyse et l'évaluation de leur impact sur les performances du système.

- **Empreinte mémoire** L'ajout de paramètres spécifiques à chaque implémentation nécessite des données supplémentaires. Par exemple, l'implémentation *PDC_Indices_Array* requiert des tableaux stockant les indices d'écriture pour chaque tâche. Ces données sont introduites dans le modèle raffiné et correspondent à des composants *data* dont les propriétés précisent leur type et leur taille. On évalue alors l'empreinte mémoire du modèle raffiné en parcourant les composants *data* du modèle raffiné.
- **Surcoût en temps d'exécution** L'introduction des composants intergiciels au sein de la description des tâches (composants *thread*) nécessite une réévaluation du pire temps d'exécution (WCET) ce ces dernières. En considérant qu'une première évaluation du WCET est connue et annoté à chaque tâche, il faut y ajouter le WCET des composants intergiciels introduits lors du raffinement. En disposant de leur description comportementale qui détaille précisément leur implémentation, celle-ci est traduite en graphe d'exécution à partir duquel le WCET du composant peut être calculé. Cette valeur est ensuite ajoutée au précédent WCET de la tâche courante.

Le choix de l'implémentation adaptée à l'architecture est ainsi rendu possible grâce à l'introduction des composants intergiciels dans le processus de raffinement. De plus, l'évaluation du coût à partir du modèle raffiné assure une démarche de validation cohérente puisque ce dernier est directement traduit en code exécutable à l'aide d'un *pretty-printer*. Ce processus de raffinement sera détaillé en section 7.2. Dans la prochaine section nous détaillons l'implémentation sans verrou (*PDC_Indices*, *PDC_Indices_Array*) avec notamment les calculs d'indices.

### 5.3.4 Composants de communication pour l'implémentation sans verrou

Nous décrivons plus en détails la mise en œuvre de l'implémentation sans verrou des communications différées que nous avons présentés précédemment.

Nous utilisons un tampon circulaire pour implémenter les files de messages sans verrou. Le tampon circulaire est une structure qui permet l'accès concurrent sans exclusion mutuelle à la file entre un producteur et un consommateur. En particulier ils accèdent à des zones distinctes et ne réalisent aucun décalage de donnée. Il faut cependant assurer l'exclusion mutuelle entre les producteurs. Nous avons proposé en section 4.3 d'assurer l'exclusion mutuelle par l'attribution des indices d'écriture aux producteurs. Nous décrivons ici la mise en œuvre de ce modèle de communication. Pour des raisons de clarté, l'émission et la réception seront explicitées par les instructions *send* et *receive*.

**Numérotation des messages** Une manière d'éviter les conflits entre les opérations *send* est de calculer un unique indice auquel un message sera écrit. Cet indice spécifique assure un accès exclusif à l'emplacement désigné pour un intervalle de temps défini. Si l'intervalle d'exclusion est choisi suffisamment grand, alors les accès en écriture peuvent être complètement ordonnés. Nous formalisons le calcul des indices de la façon suivante : chaque fois qu'une opération *send* est invoquée, un numéro de séquence $i_m$ est fourni afin de stocker le message à l'indice $i_m \ mod \ L$, avec $L$ la taille de la file. Pour toute paire de messages distincts, si leurs numéros de séquence sont égaux modulo $L$, cela signifie qu'au moins l'un des deux est périmé (et peut être écrasé).

Le numéro de séquence d'un message $i_m$ est le cardinal de l'ensemble des messages reçus strictement avant celui-ci. Pour définir cet ensemble, nous allons utiliser les notations suivantes :
- Une file de message $q$ de taille $L$ est associée à un ensemble $PT_q$ de tâches émettrices, et une unique tâche réceptrice, $T_r$.
- La date d'échéance du $k^{eme}$ job de $T_j$ est notée $JD(j,k)$.
- Un ordre arbitraire $\prec$ est utilisé pour ordonner des tâches distinctes.
- $|X|$ est le cardinal de l'ensemble $X$.

**Modèle de communication** Une émission de message par une tâche $T_j$ à la tâche $T_i$ est modélisé par une connexion entre les ports de $T_j$ et de $T_i$. Toute tâche $T_j$ écrivant sur son port de sortie $p$, émet vers les $n$ ports qui y sont connectés. L'écriture d'un message sur un port d'émission $p$ signifie que la tâche émettrice écrit directement le message sur l'ensemble des ports connectés à $p$. Afin de détecter des comportements anormaux, toute tâche $T_j$ émettant plusieurs fois sur son port $p$ par période déclenche une erreur. Par conséquent, au sein d'une même période, plusieurs appels à

*send* ne provoquent pas l'ajout de plusieurs messages : le dernier appel à *send* écrase le précédent message écrit dans la même période et provoque une erreur.

Les numéros de séquence ne doivent alors pas être affectés par la concurrence entre plusieurs opérations *send*. Concernant la lecture, puisqu'une file n'est associée qu'à un seul lecteur, les opérations *receive* successives accèdent à des indices de lecture consécutifs. Nous traitons l'émission et la réception dans les sections qui suivent.

#### 5.3.4.1 Emission

Ce modèle de communication se traduit à la ligne 13 de l'implémentation de *Put_Value* du listing 5.12 par l'affectation $buffer[CDW[IterationCounter]] := value$. *IterationCounter* est le compteur de périodes pour la tâche courante. Celui-ci est alors incrémenté à la fin de chaque période par le composant *Send_Output* (listing 5.14).

```
1  subprogram Send_Output
2    features
3      IterationCounter: in out parameter Base_Types::Integer_16;
4      CDWSize: in parameter Base_Types::Integer_16;
5    annex behavior_specification {**
6          states s: initial final state;
7          transitions
8            t: s -[]-> s { IterationCounter := (IterationCounter+1) mod CDWSize };
9    **};
10 end Send_Output;
```

Listing 5.14 – Composant Send_Output selon l'implémentation PDC_Indices_Array

A chaque période, le producteur écrit à l'emplacement $CDW[IterationCounter]$. Le tableau *CDW* indique alors l'ensemble des emplacements auxquels le producteur pourra écrire successivement d'une période à une autre. Nous décrivons la méthode pour calculer les valeurs contenues dans ce tableau.

***Définition : ordre de réception global*** L'attribution des emplacements d'écriture aux différents producteurs doit garantir l'exclusion mutuelle. La sélection de l'emplacement doit alors reposer sur un ordre de réception global. Soient $m1$ et $m2$ deux messages envoyés sur la même file. $T_{s1}$ (resp. $T_{s2}$) envoie le message $m1$ (resp. $m2$) lors de son $k1^{eme}$ (resp. $k2^{eme}$) job. $m1$ est reçu strictement avant $m2$ si et seulement si les deux conditions suivantes sont vérifiées :
– l'échéance du $k1^{eme}$ job de $T_{s1}$ est antérieure ou égale à l'échéance du $k2^{eme}$ job de $T_{s2}$ : $JD(s1,k1) \leq JD(s2,k2)$ (condition C1),
– les échéances sont égales et $T_{s1}$ précède $T_{s2}$ selon l'ordre arbitraire $\prec$ (condition C2).
Cette définition donne les indications pour calculer le cardinal de l'ensemble des messages reçus avant le message $m2$. Premièrement, nous calculons le nombre de paires $(s1,k1)$ tels que $JD(s1,k1) \leq JD(s2,k2)$. Ce nombre correspond au cardinal de l'ensemble des jobs satisfaisant la condition C1. Ensuite, nous retranchons le nombre de jobs ne respectant pas la condition C2.

**Jobs satisfaisant C1** Soit *SEJD* le cardinal de l'ensemble des messages reçus avant la date $t$. Cela correspond au cardinal de l'ensemble des échéances de job antérieurs à $t$.

$$SEJD(q,t) = \sum_{j \in PT_q} (\lfloor \frac{t - D_j}{P_j} \rfloor + 1)$$

**Jobs ne respectant pas C2** Il faut cependant attribuer des indices distincts aux jobs de même échéance afin de garantir l'exclusion mutuelle. Nous considérons maintenant l'ensemble des jobs avec une échéance égale à l'échéance du job de $T_j$. Nous désignons par *Followers* de $T_j$ l'ensemble des tâches qui succèdent à $T_j$ selon l'ordre $\prec$, et qui ont une échéance à la $k^{eme}$ période de $T_j$.

Soit $\mathbb{N}$ l'ensemble des naturels. Notons que si $\frac{t - D_r}{P_r}$ est un naturel, cela signifie qu'il existe un entier $k$ tel que $t = k * P_r + D_r$. $Followers(q, j, k)$ est alors défini comme :

$$Followers(q,j,k) = \sum_{r \in PT_q, j \prec r} \begin{cases} 1 & si \quad \frac{JD(j,k) - D_r}{P_r} \in \mathbb{N} \\ 0 & sinon \end{cases}$$

Enfin, lorsque une opération *send* est invoquée par le $k^{eme}$ job de la tâche $T_j$, son numéro de séquence, noté $MSN(q, j, k)$ est obtenu de la façon suivante :

$$MSN(q,j,k) = SEJD(q, k * P_j + D_j) - Followers(q,j,k)$$

Le tableau *CDW* du producteur $T_j$ associé au port $q$ est alors initialisé selon la formule suivante (k étant le numéro de la période courante du producteur et L la taille du port $q$) :

$$\text{CDW(q,j,k)} = \text{MSN(q,j,k)} \mod \text{L}$$

Notons que ce calcul ne nécessite l'utilisation d'aucune variable d'état partagée entre les tâches. Aucun verrou n'est alors nécessaire. Nous avons présenté les deux fonctions d'envoi. Nous traitons maintenant les fonctions de réception.

#### 5.3.4.2 Réception

Implémenter l'opération *receive* nécessite d'identifier les messages disponibles. Soit $PR(r,t)$ la date d'activation courante de la tâche $T_r$ :

$$PR(r,t) = \lfloor \frac{t}{P_r} \rfloor * P_r$$

La figure 5.13 montre le status des messages à une date $t$. Les messages disponibles pour la lecture à $t$ doivent être transmis avant la date $PR(r,t)$. Cela signifie que leurs indices sont plus petits ou égaux à $SEJD(q, PR(r,t))$. De plus, les messages transmis avant la date $PR(r,t) - P_r$ sont périmés. En conséquence, l'opération *receive* délivre les messages dont le numéro de séquence (MSN) est compris entre $SEJD(q, PR(r,t) - P_r) + 1$ et $SEJD(q, PR(r,t))$, nommés $first$ et *last* par la suite.

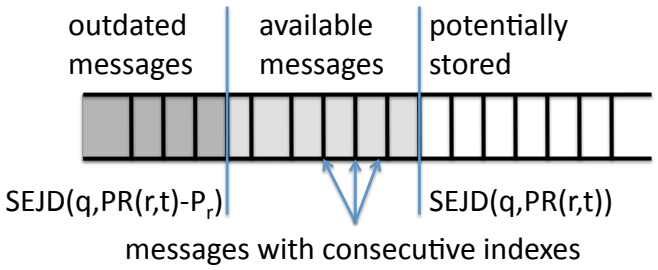



FIGURE 5.13 – Status des messages selon leurs indices

Le composant *Receive_Input*, appelé au début de chaque job, détermine les délimitations de la zone de la file accessible au consommateur durant son job courant (listing 5.15). Pour cela, la fonction utilise le tableau *CPR* qui contient les valeurs successives de *last* (précalculées) pour l'ensemble des jobs du consommateur. Si la valeur de *last* est identique entre deux jobs successifs, alors on considère que les messages sont périmés pour le job actuel. *first* et *last* sont alors réinitialisés à une valeur négative.

```
1  subprogram Receive_Input
2    features
3      iteration_counter: in out parameter Base_Types::Integer_16;
4      CPR: in parameter ArrayDataType;
5      CPRSize: in parameter Base_Types::Integer_16;
6      first: requires data access Base_Types::Integer_16;
7      last: requires data access Base_Types::Integer_16;
8      bufferSize: in parameter Base_Types::Integer_16;
9      annex behavior_specification {**
10       variables prev : Base_Types::Integer_16;
11         states s: initial final state;
12         transitions
13           t2: s -[]-> s {
14             prev := (iteration_counter-1) mod CPRSize;
15             if(CPR[prev] != CPR[iteration_counter])
16               first :=(last+1) mod bufferSize;  last:= CPR[iteration_counter]
17             else first := -1; last  := -1 end if;
18             iteration_counter := (iteration_counter+1) mod CPRSize
19           };
20     **};
21 end Receive_Input;
```

Listing 5.15 – Composant Receive_Input selon l'implémentation PDC_Indices_Array

Ensuite, pour chaque message que le consommateur souhaite consommer, celui-ci appelle le composant *Next_Value* dont l'implémentation est donnée dans le listing 5.16. Si $first$ est positif, cela signifie qu'au moins un message est disponible et n'est pas périmé. Dans ce cas, la fonction renvoie la donnée contenue dans la file à l'indice $first$. S'il reste des messages disponibles ($first \neq last$), l'indice $first$ est incrémenté.

```
1  subprogram Next_Value
2    prototypes
3      output_type: data;
4    features
5      first: requires data access Base_Types::Integer_16;
```

```
6       last: requires data access Base_Types::Integer_16;
7       buffer: requires data access ArrayDataType;
8       output: requires data access output_type;
9       bufferSize: in parameter Base_Types::Integer_16;
10    annex behavior_specification {**
11      states  s: initial final state;
12      transitions
13        t: s -[]-> s {
14          if(first >=0)
15            output := buffer[first];
16            if(first!=last)
17              first := (first+1) mod bufferSize
18            else  ... -- error port and send and error
19            end if
20          end if
21        };
22    **};
23  end Next_Value;
```

Listing 5.16 – Composant Next_Value selon l'implémentation PDC_Indices_Array

Nous avons précisé l'implémentation sans verrou des communications différées, notamment sur le calcul d'indice et la modélisation des composants intergiciels. Cette implémentation nécessite cependant des files de messages de taille suffisante pour assurer l'exclusion mutuelle. La prochaine section fournit son calcul et démontre la faisabilité de cette approche.

#### 5.3.4.3 Calcul de la taille de la file

Pour déterminer la taille de la file de messages, nous calculons d'abord l'intervalle de numéros de séquence utilisé pour stocker les messages non périmés à un instant $t$. Si la taille de cet intervalle est strictement inférieure à la taille de la file, alors chaque emplacement de la file contient au maximum un message produit avant $t$.

La borne inférieure a été calculée dans la section précédente, nous la dénotons $ILB(q,t)$ :

$$ILB(q,t) = SEJD(q, PR(r,t) - P_r)$$

Concernant la borne supérieure, soit une opération *send* invoquée par $T_s$ à l'instant $t$. Le message est délivré à la date $PR(s,t) + D_s$. Alors, le numéro de séquence est inférieur ou égal à $SEJD(q, PR(s,t) + D_s)$. Soit $D_{max}$ la plus grande date d'échéance des tâches émettrices. Notons que $SEJD(q,t)$ est une fonction monotone croissante, c-a-d $PR(s,t) \leq t \leq PR(r,t) + P_r$. Par conséquent, $SEJD(q, PR(s,t) + D_s)$ est toujours inférieur ou égal à $SEJD(q, PR(r,t) + P_r + D_{max})$, dénoté $IUB(q,t)$.

$$IUB(q,t) = SEJD(q, PR(r,t) + P_r + D_{max})$$

Par conséquent, l'écart maximal entre les bornes $ILB(q,t)$ et $IUB(q,t)$ détermine la taille de l'intervalle d'indices utilisés pour stocker les messages non-périmés. Elle est alors inférieure à la valeur maximale de $IUB(q,t) - ILB(q,t) - 1$. Ainsi, si la taille $IS(q)$ de l'intervalle est choisi selon la formule 5.1, cela est une condition suffisante pour assurer qu'aucun message non-périmé ne sera perdu.

$$IS(q) = max_{t \in \mathbb{R}+}(IUB(q,t) - ILB(q,t)) \tag{5.1}$$

La valeur exacte est calculable par des solveurs numériques. Nous donnons néanmoins une autre borne facilement calculable à la main et légèrement plus grande que la valeur exacte. Nous partons du fait que $\lfloor \frac{a+b}{c} \rfloor - \lfloor \frac{a}{c} \rfloor \leq \lfloor \frac{b}{c} \rfloor + 1$ pour des entiers positifs $a$, $b$ et $c$. Appliquons ce résultat avec :

$$\begin{aligned} a_j &= PR(r,t) - P_r - D_j \\ b_j &= 2 * P_r + D_{max} \\ c_j &= P_j \end{aligned}$$

Les bornes $ILB(q,t)$ et $IUB(q,t)$ peuvent être réécrites ainsi :

$$IUB(q,t) = SEJD(q, PR(r,t) + P_r + D_{max}) = \sum_{j \in PT_q} \lfloor \frac{a_j + b_j}{c_j} \rfloor \tag{5.2}$$

$$ILB(q,t) = SEJD(q, PR(r,t) - P_r) = \sum_{j \in PT_q} \lfloor \frac{a_j}{c_j} \rfloor \tag{5.3}$$

$$IUB(q,t) - ILB(q,t) \leq \sum_{j \in PT_q} (\lfloor \frac{2 * P_r + D_{max}}{P_j} \rfloor + 1) \tag{5.4}$$

La formule 5.4 fournit une estimation de la taille de l'intervalle stockant les messages non-périmés légèrement plus élevée que la valeur exacte.

Nous avons démontré la faisabilité de modéliser l'implémentation concrète des composants intergiciels à l'aide d'un langage de modélisation comme AADL. Cette capacité à modéliser précisément la mise en œuvre de ces composants par la plate-forme d'exécution facilite la maîtrise du coût des composants choisis et permet ainsi de les adapter.

## 5.4 Conclusion

Pour répondre aux problématiques du chapitre 3, nous avons défini une démarche autour de la mise en place d'un processus de génération de code basé sur des stratégies alternatives de génération, des patrons de transformation pour faciliter l'adaptation du processus ainsi que des composants intergiciels générés dans le but de maîtriser le coût du code généré.

Pour mettre en place le *processus de raffinement incrémental*, introduit en section 4.1, nous avons défini un méta-modèle de *chaîne de raffinement* . Nous avons démontré l'applicabilité des patrons de transformations introduits en section 4.2 et avons défini une méthodologie pour les appliquer. Enfin, l'introduction des composants intergiciels au sein du processus de raffinement assure une maîtrise de l'implémentation et du coût engendré. Nous avons illustré ce point sur la gestion des communications et leur modélisation. Dans le prochain chapitre nous décrivons la mise en œuvre de l'ensemble de cette démarche au sein de l'environnement de conception OSATE.

# Chapitre 6

# Mise en oeuvre : définition du framework RAMSES

Nous avons proposé une démarche de génération de code favorisant l'adaptation de la stratégie grâce à la définition d'un *workflow* et à l'utilisation de l'ingénierie dirigée par les modèles (chapitre 5). Ce chapitre présente la mise en œuvre de cette démarche à travers le framework RAMSES (*Refinment of AADL Models for Synthesis of Embedded Systems*) que nous avons mis en place. Ce framework est intégré à l'environnement de conception OSATE [32]. OSATE s'appuie sur des travaux antérieurs autour du générateur de code Ocarina [49] et du langage de modélisation AADL [83]. Dans ce contexte, RAMSES est une surcouche à OSATE qui enrichit ce processus de génération pour renforcer la maîtrise du code généré : notamment RAMSES propose une architecture flexible fondée sur des techniques récentes de l'IDM. Au chapitre précédent, nous l'avons illustré autour des langages AADL et ATL. Nous pourrions également l'adapter au langage MARTE.

Ce chapitre décrit l'architecture du framework RAMSES. En 6.1, nous présentons brièvement l'environnement OSATE sur lequel s'intègre RAMSES. Ensuite en 6.2, nous présentons l'architecture de RAMSES. En 6.3, nous décrivons la démarche d'utilisation de RAMSES et comment définir un processus de génération de code. Enfin, les sections 6.4, 6.5 et 6.6 précisent respectivement les procédés de transformation, d'analyse et de génération de code mis en œuvre.

## 6.1 OSATE

OSATE (*Open Source AADL Tool Environment*) est un environnement de développement open-source autour du langage de modélisation AADL développé par le SEI (*Software Engineering Institute*). Il s'adresse aux concepteurs de SETRC et fournit des outils de conception et d'analyse. Son architecture est illustrée par la figure 6.1. OSATE est construit autour de l'environnement Eclipse et de son framework de méta-modélisation EMF (*Eclipse Modeling Framework*). Eclipse fournit un ensemble d'abstractions pour spécialiser l'environnement à une application particulière. EMF est une surcouche pour la définition de langages spécifiques et pour la transformation de modèles : EMF inclut notamment le méta-métamodèle Ecore et le support du format d'échange XMI. Pour supporter l'intégration de plug-ins, OSATE définit également des abstractions pour mettre en place des outils d'analyse de modèles AADL ainsi que des générateurs de code pour des plates-formes d'exécution spécifiques.

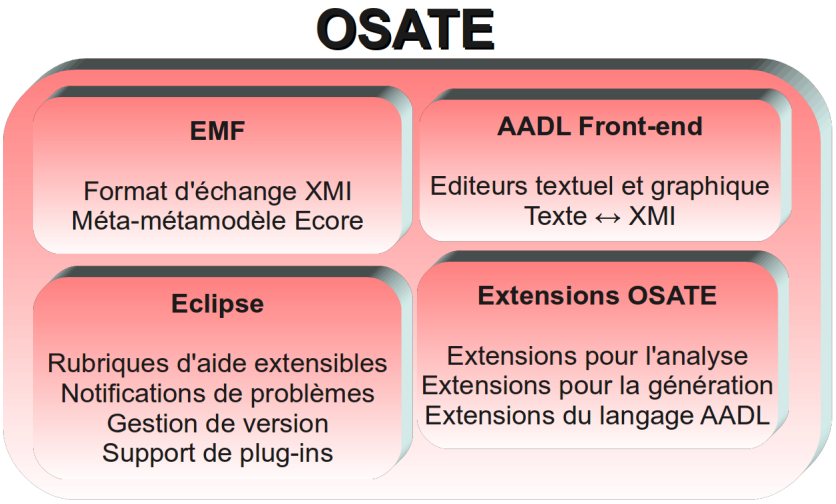



FIGURE 6.1 – Architecture d'OSATE

OSATE est une spécialisation du framework Eclipse [2] pour AADL. Le code source d'OSATE s'organise en un ensemble conséquent de 46 projets Java qui s'insèrent au-dessus du framework Eclipse/EMF. Parmi ces 46 projets, 12 sont dédiés à l'analyse (*e.g.* analyse architecturale, latence, consommation d'énergie...). Les projets restants définissent l'architecture d'OSATE. Notamment :

- ***org.osate.aadl2*** définit le méta-modèle AADL au format Ecore [17]. Ce projet contient également l'ensemble des classes Java qui constituent les éléments du méta-modèle. Elles sont automatiquement générées à partir du méta-modèle Ecore.
- ***org.osate.aadl2.instanciation*** implémente l'instanciation de modèles AADL, c'est-à-dire la sélection du composant AADL principal et la simplification du modèle (suppression des composants non utilisés, fusion des types hérités...).
- ***org.osate.aadl2.modelsupport*** fournit des mécanismes pour faciliter la mise en place de plug-ins d'analyse et de génération : méthodes de parcours des éléments d'un modèle AADL, notifications d'erreurs, utilisation des ressources internes à l'environnement (éditeur textuel, explorateur de projets).
- ***org.osate.annexsupport*** permet la définition et l'intégration d'annexes au langage AADL (*i.e.* extensions du langage). Il fournit les types génériques qu'il faut spécialiser pour assurer un

support complet d'une annexe.
La figure 6.2 donne un aperçu de l'environnement Eclipse dans lequel les développeurs créent leurs plug-ins OSATE (*i.e. back-end*).

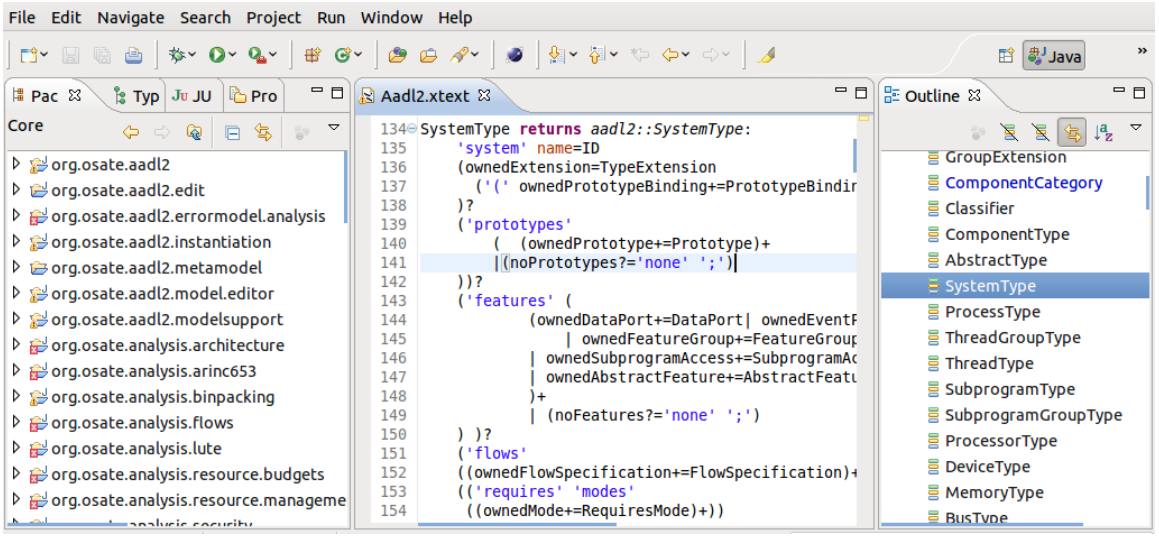



FIGURE 6.2 – Back-end d'OSATE

Cet ensemble de plug-ins construit l'interface utilisateur (*i.e. front-end*) illustrée par la figure 6.3. Cette interface utilisateur est constituée d'un explorateur de projets AADL, un éditeur textuel, un éditeur graphique, un arbre d'objets AADL ainsi qu'une console de notifications. Les plugins d'analyse et de génération de la *back-end* sont disponibles à l'utilisateur via des menus contextuels ou des boutons.

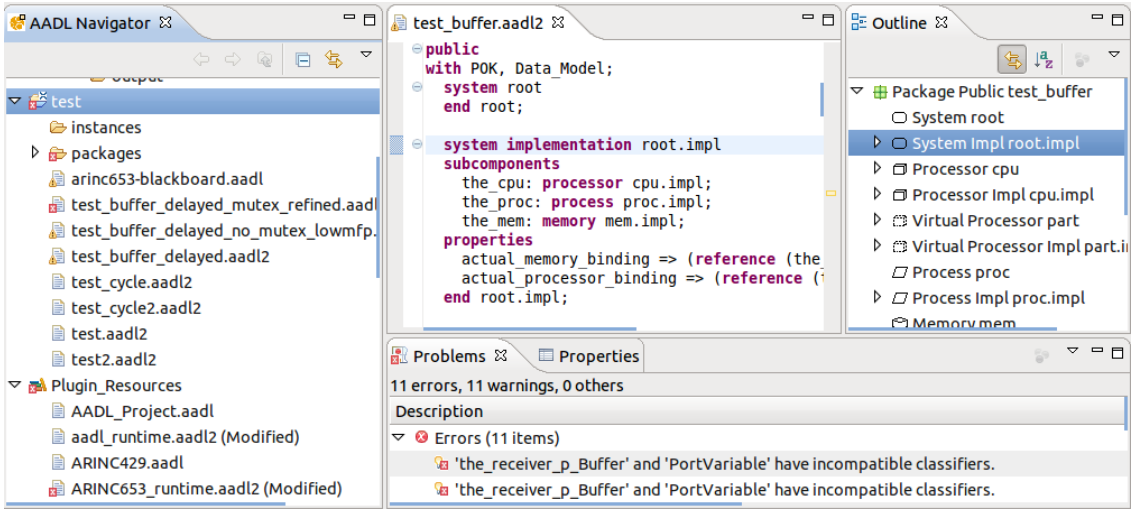



FIGURE 6.3 – Front-end d'OSATE

Cet environnement se constitue d'une surcouche à Eclipse développé essentiellement en langage Java. Nous avons contribué au développement d'OSATE via notre framework RAMSES

qui représente 26 projets Eclipse supplémentaires qui s'intègrent à l'architecture existante. Nous présentons RAMSES dans la section suivante.

## 6.2 Architecture de RAMSES

Le framework RAMSES est un ensemble de modules pour l'environnement OSATE. La figure 6.4 donne une vue d'ensemble de l'architecture de RAMSES. Celle-ci est constituée de quatre branches : *Lancement*, *Transformation*, *Analyse* et *Generation*. Celles-ci sont décrites dans les paragraphes suivants.

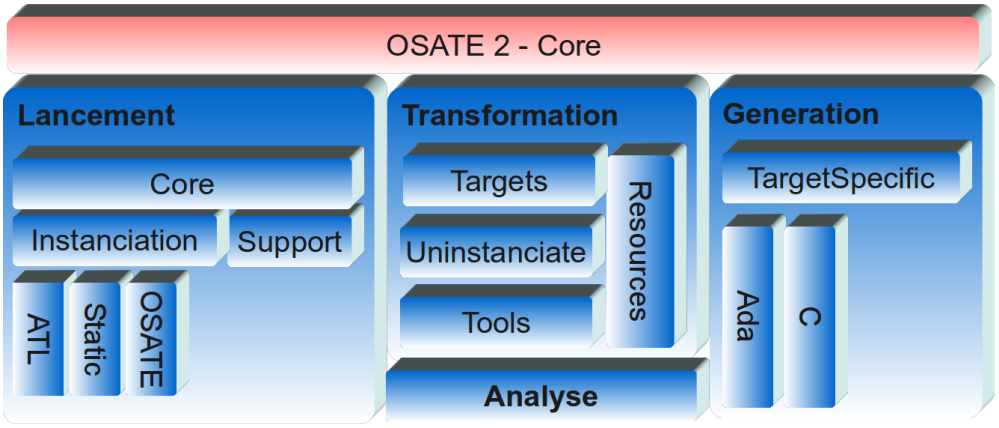



FIGURE 6.4 – Architecture de RAMSES

**Lancement** Cette branche a pour rôle de piloter le *processus de raffinement incrémental* et de coordonner ses différentes étapes (*e.g.* transformation, analyse, génération). Il s'agit du cœur du framework. Elle implémente la logique d'exécution du processus et contient les ressources nécessaires à son fonctionnement : méta-modèle du processus, moteur de transformation. Les plugins *Core* et *Support* assurent respectivement l'initialisation du processus en ligne de commande et son exécution. L'initialisation est réalisée en ligne de commande par l'utilisateur qui spécifie : une liste de modèles AADL, le code source métier ainsi que le fichier décrivant le processus. Le lancement est détaillé en section 6.3.

**Transformation** Cette branche réalise les étapes *Transformation* du processus en s'appuyant sur plusieurs bibliothèques :

– *Une bibliothèque de modules de transformation* (*Targets*) : ceux-ci réalisent le raffinement de différents éléments AADL. Pour favoriser la réutilisation et l'adaptation de raffinements, ces modules sont classés par plate-forme d'exécution (*e.g.* common, arinc653, osek) et par objectif de raffinement (*e.g.* communications, architecture). Ils sont référencés par l'utilisateur dans la description du processus pour définir les étapes de transformation : chaque étape référence une liste de modules qui seront *superimposés* lors de l'exécution du processus.
– *Une bibliothèque de fonctions d'interrogation* (*Tools*) : les fonctions d'interrogation sont des expressions OCL qui modélisent des contraintes plus ou moins complexes pour filtrer les éléments AADL sur lesquels les transformations sont appliquées. Ces interrogateurs améliorent ainsi la lisibilité des transformations.

– *Une bibliothèque de transformations identités* (*Uninstanciate*) : cette bibliothèque assure la conservation des éléments AADL lors du séquencement des étapes de transformation du processus (séparation des préoccupations). Ainsi, chaque étape de transformation s'applique potentiellement sur un sous-ensemble d'éléments du modèle. Le sous-ensemble complémentaire est cependant conservé pour être traité lors de la prochaine étape de transformation.
– *Une bibliothèque de modèles d'intergiciels* (*Resources*) : le raffinement ayant pour objectif d'intégrer au modèle les composants intergiciels mis en œuvre par la plate-forme d'exécution, ces derniers sont formalisés dans une bibliothèque de modèles AADL. Ces modèles de composants sont classés par plate-forme d'exécution et par objectif. Par exemple, le modèle *ARINC653.aadl* spécifie l'ensemble des composants représentant les fonctions et les types de données du standard ARINC653. Cette bibliothèque facilite la maintenance de ces composants indépendamment des modules de raffinement.

La branche *Transformation* est détaillée en section 6.4.

**Analyse** Cette branche réalise les étapes *Analysis* du processus. Elle contient les modules d' analyse de modèles AADL. Chaque objet *Analysis* référence l'identifiant de l'outil d'analyse (*i.e.* attribut *method*). Cet identifiant renvoie à un objet de type *Analyser* qui implémente l'analyse en elle-même. Ainsi, pour être intégré au sein d'un processus, chaque module d'analyse doit hériter du type *Analyzer*, comme le montre la figure 6.5. La méthode *getAnalyzerID()* renvoie l'identifiant de l'outil, correspondant à celui indiqué par l'attribut *method* de l'étape correspondante du processus. La classe doit également fournir les méthodes *performAnalysis()* et *setParameters()* pour respectivement lancer l'analyse et paramétrer l'outil. La méthode *setParameters()* est également utilisée pour stocker les résultats de l'analyse dans un paramètre d'entrée/sortie. Le type *Analyzer* assure donc l'intégration d'outils externes existants (*e.g.* AADL Inspector) et la définition d'analyses internes à RAMSES (*e.g.* calcul de WCET de modèles AADL comportementaux).

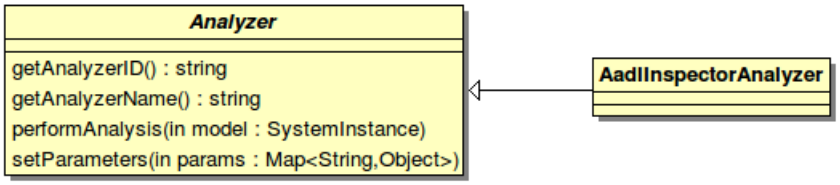



FIGURE 6.5 – Modélisation des outils d'analyse au sein de RAMSES

Nous prenons par exemple l'outil d'analyse *AADL Inspector* [1]. Cet outil réalise diverses analyses de modèles AADL, notamment des analyses d'ordonnancement à l'aide d'autres outils comme Cheddar. Nous souhaitons l'appeler depuis RAMSES : la ligne de commande pour appeler AADL Inspector est donnée par le listing 6.1. L'utilisateur souhaite réaliser une analyse d'ordonnancement du SETRC formalisé par les modèles AADL *mainmodel.aadl* et *runtime.aadl* (ligne 1). Pour cela, il a besoin du module Cheddar (ligne 2). Les résultats seront enregistrés dans un fichier *output.xml* (ligne 3). De plus, l'outil est exécuté en mode non interactif (ligne 4).

```
1 AADLInspector -a mainmodel.aadl, runtime.aadl
2         --plugin /home/myself/AADLInspector/config/plugins.common/cheddar.aip
3         --result /home/myself/output.xml
4         --show no
```

Listing 6.1 – Ligne de commande pour AADL Inspector

Pour intégrer cette ligne de commande au sein de RAMSES, une classe héritant de *Analyzer* doit être définie : la classe *AADLInspectorAnalyzer* (listing 6.2). La méthode *performAnalysis* appelle *AADL Inspector* en ligne de commande comme nous l'avons illustré précédemment. Les résultats d'analyses contenus dans le fichier XML de sortie sont alors extraits et mis en forme. La branche *Analysis* est détaillée en section 6.5.

```
1  public class AADLInspectorAnalyzer extends Analyzer {
2    public String getAnalyzerID(){ return "AADLInspector"; };
3    public String getAnalyzerName(){ return "AADL Inspector"; };
4
5    public void performAnalysis(SystemInstance model){
6       String modelsPaths = serializeAndReturnPaths(model);
7       Process p = Runtime.getRuntime().exec(new String[] {
8            "AADLInspector"                    ,
9            "-a", modelsPaths, "--plugin", PATH+"config/plugins.common/cheddar.aip",
10           "--result", OUTPUT_FILE_PATH, "--show", modeOption});
11      int exitValue = p.waitFor();
12      ...
13      getResults(new File(OUTPUT_FILE_PATH));
14   }...
15 }
```

Listing 6.2 – Intégration de l'outil AADLInspector

**Generation** Cette branche réalise les étapes *Generation* du processus. Il s'agit des modules produisant le code source exécutable à partir des modèles AADL raffinés. Pour chaque plate-forme d'exécution, une classe spécifique héritant de *AadlTargetUnparser* réalise la génération de code exécutable. Ainsi, les classes *AadlToPokCUnparser* et *AadlToOSEKCUnparser* réalisent la génération de code respectivement pour les plates-formes POK et OSEK. Par exemple, *AadlToPOKCUnparser* s'appuie sur une génération conforme au standard ARINC653 mais également sur une génération de fichiers de compilation (*makefiles*) spécifiques à POK.

Nous avons présenté l'architecture du projet RAMSES qui est une mise en œuvre du *processus de raffinement incrémental* que nous avons proposé au chapitre 4 et détaillé au chapitre 5. Les prochaines sections décrivent plus en détails les branches *Lancement* (6.3), *Transformation* (6.4) et *Generation* (6.6).

## 6.3 Lancement

Comme précisé précédemment, RAMSES est une surcouche à l'environnement OSATE et intègre le processus de raffinement incrémental décrit dans les chapitres précédents. Il dispose ainsi de l'environnement graphique d'OSATE : entre autres, un éditeur AADL ainsi que des analyseurs syntaxiques et sémantiques. Cependant, nous souhaitions automatiser des tests de non-régression. Par conséquent, RAMSES s'exécute principalement en ligne de commande. Le listing 6.3 donne un exemple de ligne de commande pour lancer un processus de raffinement incrémental avec RAMSES. Les options suivantes doivent être spécifiées :

– la liste des fichiers AADL définis par l'utilisateur (option *-m*),
– le nom du composant AADL principal englobant l'ensemble du système (option *-s*),

– l'identifiant de la plate-forme d'exécution ciblée (option *-g*), le chemin vers le répertoire dans lequel seront produits les modèles raffinés ainsi que le code généré (option *-o*),
– le nom du fichier XMI décrivant le *workflow* (option *–workflow*)

```
1 ToolSuiteLauncher -m ./input/model.aadl2 -s root.impl -g pok -o ./output/
2 --workflow=./input/Workflow.xmi
```

Listing 6.3 – Ligne de commande de lancement de RAMSES

Dans l'exemple donné par le listing 6.3, le modèle dont le chemin est *./input/model.aadl2* sera instancié à partir de son composant *root.impl*. Le modèle instancié est ensuite pris en entrée du processus de raffinement incrémental décrit dans le fichier *./input/Workflow.xmi*. Le code final sera généré pour la plate-forme d'exécution POK dans le répertoire *./output/*. Le fichier XMI décrivant le processus est une instanciation du méta-modèle de *workflow* comme indiqué précédemment. Le listing 6.4 en est un exemple. Dans cet exemple, le workflow est constitué d'une unique étape de transformation (ligne 7). Entre les balises *<list>* et *</list>* (lignes 9 à 13), l'utilisateur spécifie l'ensemble des modules de transformation qui composent cette étape. Elle est compilée juste avant son exécution en *superimposant* les modules dans l'ordre de déclaration. L'utilisateur indique ensuite que le modèle raffiné sera ensuite sauvegardé au format textuel (ligne 15) puis analysé par le logiciel *AADLInspector* (ligne 16). Si l'analyse réussie, le code est généré pour la plate-forme choisie au préalable (ligne 17). Sinon, le processus entre dans un état d'erreur (aucune stratégie alternative de raffinement n'étant formalisée, ligne 18).

```
1  <?xml version="1.0" encoding="ASCII"?>
2  <rwf:Workflow xmi:version="2.0"
3   xmlns:xmi="http://www.omg.org/XMI"
4   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5   xmlns:rwf="http://fr.tpt.aadl.ramses.control.workflow/Ramses/1.0"
6   xsi:schemaLocation="http://fr.tpt.aadl.ramses.control.workflow/Ramses/1.0
7     ../../../../fr.tpt.aadl.ramses.control.support/model/RamsesWorkflow.ecore">
8    <element xsi:type="rwf:Transformation">
9     <list>
10      <file path="ACG/targets/shared/SubprogramCallsCommonRefinementSteps"/>
11      <file path="ACG/targets/shared/PortsCommonRefinementSteps"/>
12      <file path="ACG/targets/shared/BehaviorAnnexCommonRefinementSteps"/>
13      <file path="ACG/targets/arinc653/ExpandThreadsDispatchProtocol"/>
14      ...
15    </list>
16    <element xsi:type="rwf:Serialization">
17      <element xsi:type="rwf:Analysis" method="AADLInspector" mode="automatic">
18        <yesOption><element xsi:type="rwf:Generation"/></yesOption>
19        <noOption><Error/></noOption>
20      </element>
21    </element>
22   </element>
23 </rwf:Workflow>
```

Listing 6.4 – Exemple de fichier de workflow

Les actions du processus sont directement visibles depuis l'explorateur d'OSATE (figure 6.6). L'utilisateur a fourni les ressources d'entrée du processus dans le répertoire *input* (modèles AADL, code source métier et description du processus). A l'exécution du processus, les étapes de type *Serialization* sauvegardent les modèles raffinés dans le répertoire *output/refined-models* tandis que

l'étape *Generation* génère le code exécutable dans le répertoire *output/generated-code*. Les modèles sauvegardés dans *refined-models* peuvent être exploités par un processus tierce pour vérifier la cohérence du *processus de raffinement incrémental*. Le code généré dans *generated-code* est automatiquement compilé pour produire un binaire enregistré dans ce même répertoire.

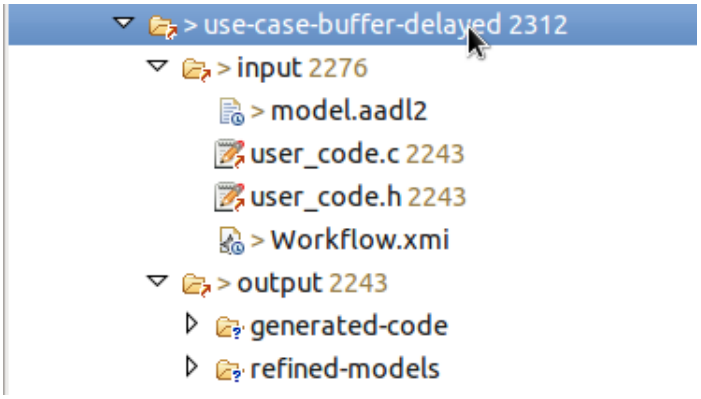



FIGURE 6.6 – Explorateur d'OSATE : organisation des fichiers générés par RAMSES

Nous avons donné une vue générale de RAMSES à travers les modules de lancement. Dans la prochaine section nous abordons la branche *Transformation* de RAMSES qui correspond à la mise en oeuvre des étapes de transformation.

## 6.4 Transformation

Les transformations de modèles réalisent des raffinements de modèles pour différentes plates-formes d'exécution. Le raffinement ayant pour objectif d'obtenir un modèle proche de l'implémentation réelle, il s'appuie en particulier sur une modélisation :

– *des ressources* fournies par la plate-forme d'exécution/l'intergiciel (*e.g.* composants intergiciels, structures de données) et introduits lors de la génération de code pour mettre en œuvre les abstractions du modèle.
– *du comportement des tâches* selon leur type d'activation (*e.g.* périodique, sporadique). La modélisation comportementale a pour objectif d'assurer que le code généré est cohérent avec la politique d'activation. Elle consiste à générer pour chaque tâche un automate comportemental correspondant à sa politique d'activation.

**Modélisation des ressources** En AADL, on modélise les ressources précédentes à l'aide de composants *subprogram* pour les fonctions et de composants *data* pour les types de données. La figure 6.7 donne un aperçu de la modélisation en AADL des ressources assurant l'exclusion mutuelle pour différentes plates-formes (ARINC653, POSIX et OSEK). Les variables partagées AADL (*data access*) sont traduites différement selon la cible : en sémaphore pour ARINC653, en mutex pour POSIX ou en ressource pour OSEK. De la même manière, chaque plate-forme fournit ses propres fonctions relatives aux variables partagées (initialisation, prise/relâche de verrou).

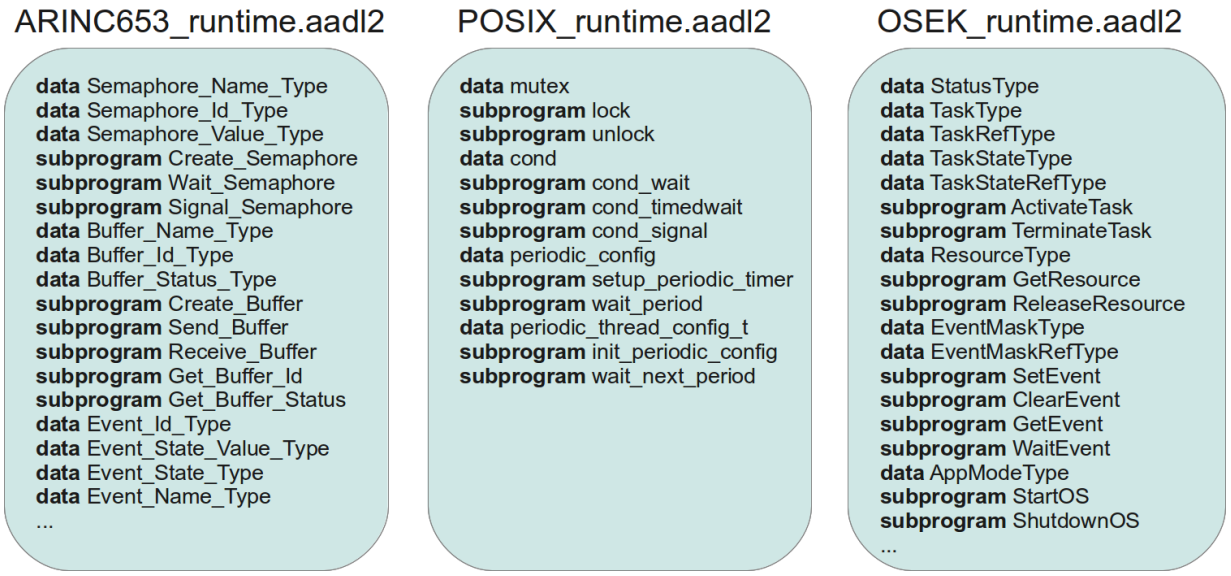



FIGURE 6.7 – Modélisation de différentes plates-formes d'exécution

L'implémentation concrète de chaque composant est spécifiée par l'utilisateur. Par exemple, le listing 6.5 décrit en AADL une partie des composants POSIX. La correspondance entre le composant AADL et les éléments de la plate-forme cible est spécifiée à l'aide des propriétés *Source_Name* et *Source_Text*, comme le montre l'exemple. Le composant AADL *mutex* (ligne 4) correspond au type *pthread_mutex_t* (ligne 6) qui est défini dans le fichier *pthread.h* et qui nécessite l'inclusion de l'archive *libpthread.a* (ligne 7). L'utilisateur peut également utiliser d'autres notations pour détailler l'implémentation du composant. Celles-ci seront abordées à la prochaine section.

```
1  data mutex
2  properties
3    Source_Name => "pthread_mutex_t"; Source_Text => ("pthread.h", "libpthread.a");
4  end mutex;
5  subprogram lock
6  features
7    m: in parameter mutex;
8  properties
9    Source_Name => "pthread_mutex_lock"; Source_Text => ("pthread.h", "libpthread.a");
10 end lock;
```

Listing 6.5 – Modélisation en AADL de composants POSIX

Ces modèles décrivant les composants de la plate-forme ciblée sont ensuite référencés par le processus de raffinement. Le modèle raffiné incorpore alors ces ressources pour se rapprocher de l'implémentation réelle.

**Modélisation comportementale** Le raffinement introduit également les ressources liées à l'activation des tâches. Ces ressources sont des modèles comportementaux qui correspondent au code généré pour l'activation des tâches. Chaque tâche a une politique d'activation (*e.g.* périodique) qui est spécifiée via la propriété *Dispatch_Protocol*. Dans [60] est proposée une démarche pour

expliciter le comportement des tâches : la valeur de la propriété *Dispatch_Protocol* est traduite en automate comportemental associé à la tâche concernée.

Prenons une tâche périodique initialement modélisée par un composant *thread* pour lequel est spécifié la propriété *Dispatch_Protocol* avec la valeur *Periodic*. La traduction de l'activation périodique en automate comportemental est donnée par le listing 6.6. L'automate est modélisé via l'annexe comportementale à partir de la ligne 1. Un automate est constitué d'un ensembles d'états, de transitions et éventuellement de variables locales. L'état initial est défini par le mot-clé *initial* tandis que les états terminaux sont définis via le mot-clé *final*. Un automate peut également avoir des états *complete* qui représentent une exécution sujette à interruptions/reprises basées sur des conditions de déclenchement externes. Dans notre exemple, une tâche périodique est modélisée via l'automate du listing 6.6. Celui-ci est constitué d'un unique état *stExec* (ligne 3). Il s'agit d'un état *complete*, dont l'unique transition sortante est donc déclenchée par la condition *on dispatch* (ligne 5). Cette condition signifie que la transition est franchie lorsque la tâche démarre une nouvelle activation (périodique). Cette transition est constituée d'un bloc d'actions défini entre accolades. Ces actions correspondent aux opérations réalisées par la tâche lorsque la transition est franchie (*e.g.* appels de *subprograms*, affectations, boucles internes, branches conditionnelles).

```
1 annex behavior_specification {**
2   states
3     stExec: initial complete final state;
4   transitions
5     stExec_Exec: stExec -[on dispatch]-> stExec
6       {  Send!(Data_Source); }
7 **};
```

Listing 6.6 – Automate comportemental d'une tâche périodique

Le listing 6.7 est un second exemple d'automate pour une tâche sporadique. La condition de déclenchement de la transition diffère de la précédente (ligne 5) car la tâche n'est pas activée périodiquement mais lors d'une mise à jour de la donnée partagée *Data_Sink*.

```
1 annex behavior_specification {**
2   states
3     stExec: initial complete final state;
4   transitions
5     stExec_Exec: stExec -[on dispatch Data_Sink]-> stExec
6       {  Get_Resource!(Data_Sink);
7          Update!(Data_Sink);
8          Release_Resource!(Data_Sink); }
9 **};
```

Listing 6.7 – Automate comportemental d'une tâche sporadique

La transformation consiste ainsi à raffiner certaines abstractions du modèle comme les moyens de communication et la sémantique d'exécution. Cela est réalisé grâce à des bibliothèques de composants AADL ainsi qu'à des automates comportementaux. Il est ensuite nécessaire d'évaluer l'impact de ce raffinement vis à vis des propriétés du modèle initial. Cela est abordé dans la section suivante.

## 6.5 Analyse

Le processus de raffinement introduisant des appels à des composants spécifiques, comme illustré précédemment, le surcoût engendré doit être évalué (*subprogram* : temps d'exécution, *data* : empreinte mémoire). En particulier, l'impact sur le temps d'exécution nécessite de réévaluer le WCET des tâches et les éventuels accès à des données protégées en exclusion mutuelle pour lesquels il faut identifier les dates de début et de fin des sections critiques. Deux méthodes de calcul de WCET sont proposées : la méthode dynamique [88] et la méthode statique [5, 4]. La méthode dynamique nécessite d'exécuter le code généré et est cependant souvent sujette aux imprécisions. A l'opposé, la méthode statique ne requiert pas l'exécution du code généré et se base sur des modèles mathématiques. Par conséquent, nous orientons notre démarche d'analyse sur cette seconde méthode.

### 6.5.1 WCET fixe ou variable

Pour tenir compte de l'impact des composants intergiciels sur le WCET des tâches, il est nécessaire d'évaluer le WCET de ces composants. Ces derniers sont modélisés par des *subprograms* AADL comme nous l'avons indiqué en section 6.4. Leur WCET est spécifié via la propriété *Compute_Execution_Time*. Cependant, certains composants ont un WCET variable. Par exemple, la fonction *Put_Value*, réalisant l'insertion d'une donnée dans une file, a un WCET qui dépend de la taille des données (temps de copie) et potentiellement de la taille de la file (temps de parcours). Nous considérons alors deux cas de figure : le WCET fixe et le WCET variable.

#### 6.5.1.1 WCET fixe

Le WCET est fixe et est connu avant raffinement. Le *subprogram* est alors annoté de la propriété *Compute_Execution_Time* comme illustré par le listing 6.8 (ligne 9). Cette propriété indique alors les bornes minimum (*BCET*) et maximum (*WCET*) du temps d'exécution du composant. Dans ce cas de figure, la fonction peut être implémentée directement dans le langage cible. Le modèle indique alors le chemin vers le fichier source ainsi que le nom de la fonction réelle à l'aide des propriétés *Source_Text* et *Source_Name* comme illustré (lignes 7 et 8).

```
1  subprogram Create_Semaphore
2    features
3      SEMAPHORE_NAME      : in parameter Semaphore_Name_Type;
4      ...
5      RETURN_CODE         : out parameter Return_Code_Type;
6    properties
7      Source_Name => "CREATE_SEMAPHORE";
8      Source_Text => ("arinc653/semaphore.h");
9      Compute_Execution_Time => 0 ms .. 1 ms;
10 end Create_Semaphore;
```

Listing 6.8 – Annotation du WCET d'un composant subprogram

#### 6.5.1.2 WCET variable

Le WCET est variable. Dans ce cas de figure, illustré par le listing 6.9, le WCET dépend de paramètres qui ne sont connus qu'après raffinement du composant. Le composant *Put_Value* de l'exemple (ligne 6) réalise l'insertion de la donnée *value* dans le tableau *buffer* de type *BufferType*. La taille du tableau, spécifiable via la propriété *Dimension*, n'est cependant pas indiquée car le composant n'est pas spécifique à un modèle particulier. La propriété *Dimension* sera indiquée sur l'élément du modèle connecté au composant. Cet exemple montre qu'ici le WCET ne peut être spécifié par une propriété.

```
1  data BufferType
2  properties
3    Data_Model::Data_Representation => Array;
4  end BufferType;
5
6  subprogram Put_Value
7  features
8    value:  requires data access ValueType;
9    buffer: requires data access BufferType;
10 ...
11 end Put_Value;
```

Listing 6.9 – Modélisation d'un composant dont le WCET n'est connu qu'après raffinement

Pour évaluer le WCET lors du raffinement, il est nécessaire de détailler l'implémentation du composant. Lors du raffinement, cette description sera traduite en graphe d'exécution sur lequel sera calculé le WCET. L'annexe comportementale étant adaptée à la description fine de l'implémentation, le listing 6.10 reprend l'exemple précédent en y ajoutant cette annexe. Elle détaille une implémentation possible du composant *Put_Value* : celui-ci détermine d'abord à quel indice la donnée doit être insérée dans le tableau *buffer* selon la datation de la donnée (ligne 16). Les données du tableau sont ensuite décalées vers la droite pour insérer la nouvelle donnée (lignes 21 et 22). Pour déterminer le WCET de *Put_Value*, il faut notamment connaître le nombre maximum d'itérations réalisées sur la boucle *while* (ligne 16). Cette borne dépend de la taille du tableau *buffer* qui ne sera connue que lorsque ce paramètre sera précisé lors de l'appel au composant *Put_Value*. La spécification du composant étant incomplète (paramètres définis partiellement), on déclare un ensemble de prototypes (ligne 2) qui devront être précisés pour compléter la définition du composant.

```
1  subprogram Put_Value
2    prototypes
3      valueType : data ValueType;
4      arrayType : data BufferType;
5    features
6      value : in parameter valueType;
7      buffer : requires data access arrayType;
8    annex behavior_specification {**
9      states
10       s : initial final state;
11     variables
12       index: Integer;
13     transitions
14       t : s-[]->s {
15         index := 0;
16         while ((buffer[index].timestamp < value.timestamp)
```

```
17                  and (index < buffer.size)){
18              index := index + 1
19            }
20            if (index < bufferSize) {
21              shiftValues!(buffer,index);
22              buffer[index] := value
23            }
24          }
25    **};
26  end Put_Value;
```

Listing 6.10 – Modélisation d'un composant à WCET variable à l'aide de l'annexe comportementale

Le listing 6.11 donne un second exemple dans lequel ce composant est intégré. Les paramètres sont raffinés à la ligne 3 à l'aide des prototypes définis précédemment. En particulier, on précise le type exact du tableau. Il s'agit du type *TheReceiver1_datain* déclaré à la ligne 7. A ce moment, la taille du tableau et le type des données sont indiqués via les propriétés *Dimension* et *Base_Type*.

```
1  subprogram Put_Value_TheSender1
2  extends RuntimeExample::Put_Value
3  (valueType => data IntegerValue, arrayType => TheReceiver1_datain)
4  end Put_Value_TheSender1;
5
6  data IntegerValue extends ValueType ...
7  data TheReceiver1_datain extends BufferType
8  properties
9    Data_Model::Dimension => (10);
10   Data_Model::Base_Type => (classifier (IntegerValue));
11 end BufferType;
```

Listing 6.11 – Modèle raffiné : intégration du composant *Put_Value*

Une alternative à l'utilisation de l'annexe comportementale est l'utilisation d'une séquence d'appel. Celle-ci décompose la fonction en une séquence de fonctions secondaires et donne la possibilité de séparer les sous-fonctions de WCET fixe de celles de WCET variables. Le listing 6.12 illustre ce second exemple : une séquence d'appel scinde la fonction *Compute* en deux sous-fonctions *Compute_1* et *Compute_2*. La première a un WCET fixe annoté avec la propriété *Compute_Execution_Time* (ligne 13) tandis que la seconde a un WCET variable et son implémentation est détaillée avec l'annexe comportementale (ligne 17). Le WCET du composant global est alors calculé en faisant la somme des WCET des sous-fonctions.

```
1  subprogram implementation Compute.impl
2  calls
3    seq : { call1 : subprogram Compute_1;
4            call2 : subprogram Compute_2; };
5  connections
6    cnx1: data access inputValue -> call1.inputValue;
7    cnx2: data access call1.outputValue -> call2.inputValue;
8  end Compute.impl;
9
10 subprogram Compute_1
11 ...
12 properties
13   Compute_Execution_Time => 0 ms .. 1 ms;
```

```
14 end Compute_1;
15 subprogram Compute_2
16 ...
17 annex behavior_specification { ... };
18 end Compute_2;
```

Listing 6.12 – Description d'un composant à WCET variable à l'aide de la séquence d'appel

**Performances de la plate-forme d'exécution** L'évaluation du WCET doit également tenir compte des instructions de base de la plate-forme d'exécution. Par exemple, l'utilisation de l'annexe comportementale introduit des instructions spécifiques (*e.g.* affectations, opérations arithmétiques, comparaisons) dont il faut évaluer le temps d'exécution. Pour cela, le langage AADL fournit des propriétés standard pour préciser les performances de la plate-forme d'exécution, et permettant de calculer la durée de ces instructions. Ces propriétés sont réparties sur les différents composants. Le tableau de la figure 6.8 liste les propriétés que nous prenons en compte dans l'analyse de WCET. Celles-ci modélisent à gros grain les performances.

| **Propriété** | **Composant** | **Description** |
|---|---|---|
| Source_Data_Size | Data | Taille du type de donnée |
| Write_Time | Memory | Vitesse d'écriture en mémoire d'une donnée |
| Read_Time | Memory | Vitesse de lecture en mémoire d'une donnée |
| Word_Size | Memory | Taille de la plus petite donnée qu'il est possible de stocker en mémoire. Cette constante détermine notamment la taille en mémoire des opérateurs de base. |
| Assign_Time | Processor | Vitesse de chargement d'un bloc d'octet sur le processeur. |

FIGURE 6.8 – Propriétés AADL spécifiant les performances de la plate-forme d'exécution

Ces propriétés sont utilisées lors du processus d'évaluation du WCET, notamment lors de la construction d'un graphe d'exécution pour lequel les sommets sont caractérisés par une durée. Nous avons abordé les différents éléments de modélisation AADL qui seront utilisés pour calculer le WCET des tâches après leur raffinement. Nous présentons le processus d'évaluation dans la section suivante.

### 6.5.2 Processus d'évaluation

Le processus d'évaluation de l'impact du raffinement sur le temps d'exécution est illustré par la figure 6.9. Le principe de ce processus est d'analyser l'impact du raffinement sur les propriétés telles que le WCET et les dates des sections critiques. Pour cela, il crée un graphe d'exécution qui rassemble de manière uniforme l'ensemble des descriptions comportementales utilisées (*e.g.* automate comportemental, séquence d'appel).

Le processus se déroule en trois étapes. Premièrement, on vérifie que le modèle AADL raffiné est suffisamment déterministe pour être analysé. Pour cela, nous restreignons l'ensemble des automates comportementaux à ceux respectant le profil *Ravenscar*. Deuxièmement, on rassemble au sein d'un unique graphe d'exécution l'ensemble des descriptions comportementales de la tâche

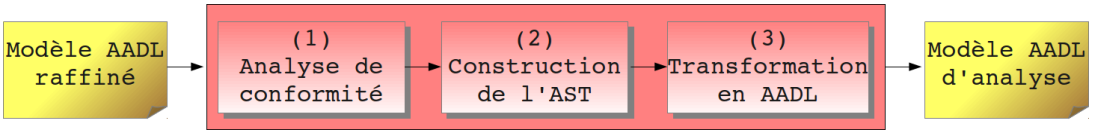



FIGURE 6.9 – Processus d'évaluation des surcoûts temporels

courante des composants appelés. Le graphe est réduit (*e.g.* dépliage de boucles, fusion de blocs de calculs) afin d'obtenir une modélisation simplifiée. Enfin, le graphe simplifié est traduit en automate comportemental simplifié. Le processus produit alors un modèle AADL d'analyse pour lequel les détails de mise en œuvre sont remplacés par des automates comportementaux simplifiés dédiés à l'analyse.

**1. Analyse de conformité *Ravenscar*** La première étape est d'assurer que l'automate comportemental est déterministe. Pour cela, le profil *Ravenscar* définit un ensemble de restrictions : (R1) aucune allocation dynamique de mémoire, (R2) la tâche réalise une boucle infinie déclenchée par un unique événement, (R3) la tâche ne se termine pas, (R4) les données partagées sont accédées en exclusion mutuelle.

On vérifie alors que l'automate respecte ces restrictions. Concernant (R1), le langage AADL ne permet pas de modéliser de l'allocation dynamique étant donné qu'il s'adresse aux systèmes temps-réel critiques. Ensuite, on analyse l'ensemble des transitions de l'automate pour détecter s'il existe une unique boucle principale (R2). On en déduit l'état qui modélise le comportement périodique (on sépare la phase d'initialisation de la phase de régime permanent). Pour s'assurer que la tâche ne se termine jamais (R3), on vérifie qu'il n'y a aucune transition vers l'état final de l'automate. Enfin, concernant les données partagées (R4), nous traitons exclusivement les files de messages pour lesquelles nous avons proposé plusieurs mécanismes assurant l'exclusion mutuelle (section 5.3) : l'utilisation de verrou ou une attribution des indices qui garantit des intervalles d'exclusion.

**2. Construction du graphe d'exécution** Lorsque l'automate est validé, on le convertit en graphe d'exécution. Etant donné les différents types d'éléments de modélisation qui renseignent l'implémentation d'une tâche (*e.g.* annexe comportementale, séquence d'appel, propriétés), le graphe d'exécution a pour objectif d'obtenir un unique modèle comportemental qui rassemble l'ensemble des notations utilisées. La figure 6.10 donne un exemple de modèle AADL décrivant le comportement d'une tâche à l'aide de différentes notations imbriquées. La tâche est décrite à l'aide d'un automate comportemental. Celui-ci référence des *subprograms* qui sont également décrits à l'aide d'un automate comportemental ou d'une séquence d'appel. La figure 6.11 donne le modèle de graphe d'exécution utilisé. Le type *ASTNode* modélise un noeud du graphe. Son attribut *kind* indique s'il s'agit d'une action quelconque (*Compute*) ou bien d'un début/fin de section critique (*CriticalSectionStart/CriticalSectionEnd*). Dans le cas d'une section critique, l'attribut *sharedData* référence la donnée partagée.

Le processus va alors analyser l'ensemble des descriptions comportementales associées à chaque tâche pour produire le graphe d'exécution. Celui-ci est crée avec un ensemble de nœuds connectés

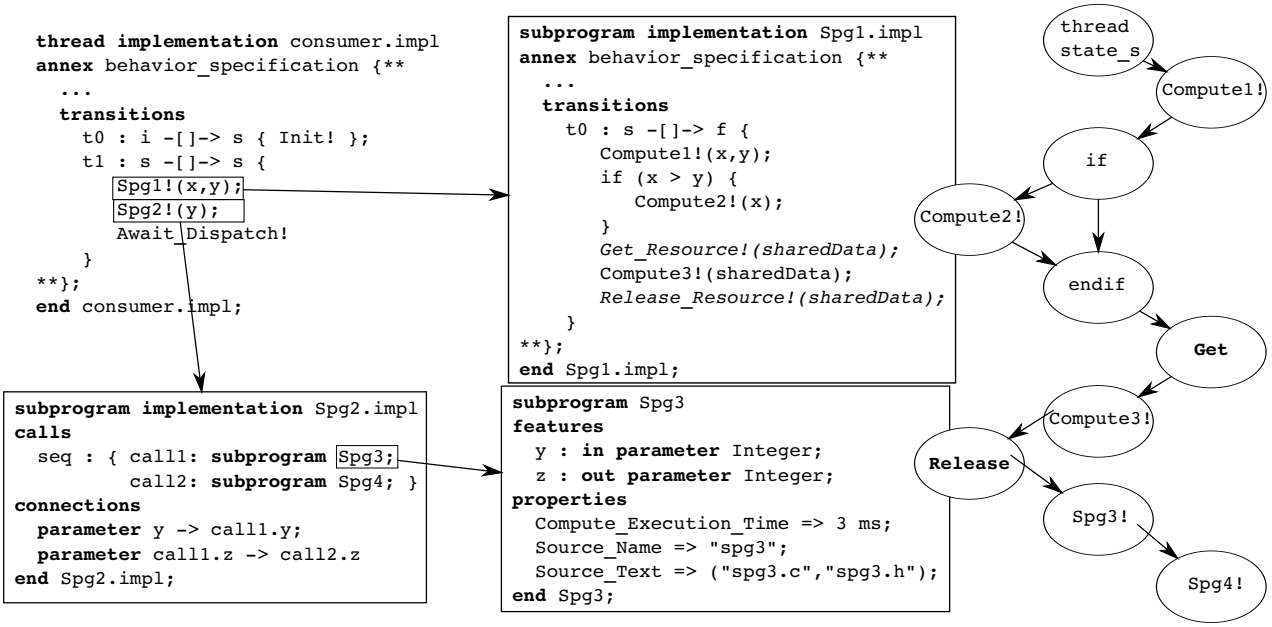



FIGURE 6.10 – Imbrication des descriptions comportementales AADL

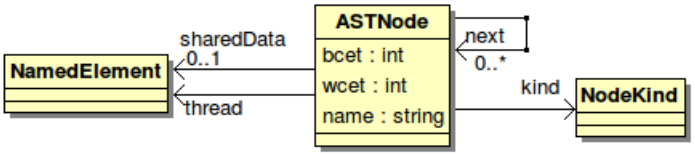



FIGURE 6.11 – Modélisation du graphe d'exécution pour le calcul du WCET

qui correspondent aux états et transitions de l'automate. On traite ensuite les blocs d'instructions de chaque transition. On considère alors différents types d'instructions :

– **Affectation** : un nœud est ajouté au graphe d'exécution et la propriété *wcet* est initialisée avec la durée maximum de l'affectation. La durée de l'affectation tient compte de trois durées :

  1. ***Assign_Time*** : le temps pour charger les instructions sur le processeur,
  2. ***Read_Time*** : le temps pour lire les opérateurs et le contenu des opérandes,
  3. ***Write_Time*** : le temps pour écrire le résultat dans une variable.

  Chaque expression d'affectation est décomposée en sous-expressions de la forme $A := B\ op\ C$ pour lesquelles est calculé le WCET à partir de ces trois durées :

$$
\begin{aligned}
WCET &= Assign\_Time + Read\_Time + Write\_Time \\
Assign\_Time &= Assign\_Time(A) + Assign\_Time(B) + Assign\_Time(op) + Assign\_Time(C) \\
Read\_Time &= Read\_Time(B) + Read\_Time(C) \\
Write\_Time &= Write\_Time(B\_C) + t_{op}
\end{aligned}
$$

  Chaque type d'opération nécessite un temps d'opération variable. Pour chaque opérateur $op$, on tient compte de la durée $t_{op}$ pour réaliser l'opération. Ces paramètres sont fixés par l'utilisateur selon l'architecture.

- **Conditionnelle** : les blocs conditionnels sont traduits en un sous-graphe. Il se compose d'un nœud *if* relié à deux sous-graphes *then* et *else*. Les sous-graphes se rejoignent au noeud *endif* modélisant la fin du bloc conditionnel. Lorsque les sous-graphes sont construits, on détermine celui dont le WCET est le plus grand par une analyse de graphe. L'ensemble du bloc conditionnel est alors remplacé par un unique nœud dont l'attribut *wcet* correspond au WCET précédent auquel est ajouté la durée d'évaluation de la condition du *if*.
- **Boucle** : chaque boucle est dépliée avant d'être intégrée au graphe d'exécution. On suppose que le nombre maximum d'itérations est fourni par une variable. Il peut s'agir d'une boucle de type *for* ou de type *while*. Dans le second cas, on cherche une sous-condition (*AND*) du while qui utilise un compteur d'itérations sous la forme $iter < expr$. On vérifie dans le corps de la boucle que le compteur est incrémenté avec une constante en cohérence avec l'opérateur de comparaison. Le nombre maximum d'itérations est déterminé par la valeur de l'expression de la sous-condition. Un exemple d'une telle boucle est donné en listing 6.10. L'expression doit correspondre soit :
  - à une constante explicite,
  - à une constante fournie par un paramètre d'entrée de type *data*. On considère que le composant *data* est une constante s'il n'est pas typé et qu'il possède la propriété *Initial_Value* indiquant sa valeur initiale. Ce second cas est illustré par le listing 6.13.

```
1 data MyConstant
2 properties
3   Data_Model::Initial_Value => ("10");
4 end MyConstant;
```

Listing 6.13 – Modélisation d'une constante en AADL

  - à une constante implicite *size* pour toute donnée de type tableau. Cette constante renvoie la taille du tableau, spécifiée via la propriété *Dimension*. Un exemple a été donné précédemment en listings 6.10 et 6.11.
- **Appel de *subprogram*** : on distingue plusieurs cas. Si la propriété *Compute_Execution_Time* est annotée au *subprogram*, alors on crée un noeud dont l'attribut *wcet* est initialisé avec la valeur de cette propriété. S'il n'en a pas, on analyse sa description comportementale. S'il n'en possède aucune, le processus notifie un avertissement et considère sa durée nulle. S'il possède un automate comportemental, alors on extrait le graphe d'exécution de celui-ci avec la même méthode de construction. Sinon, on analyse sa séquence d'appel. Pour chaque *subprogram* appelé, on extrait son sous-graphe et on l'incorpore dans le graphe principal. On tient compte également de la durée du passage de chaque paramètre au *subprogram*. Cette durée est calculée de la façon suivante. Pour chaque paramètre : 1) on détermine quelle variable est connectée au *subprogram* 2) on obtient la taille de la variable à partir de sa propriété *Source_Data_Size* 3) on calcule le temps pour charger la variable selon les propriétés *Assign_Time* et *Read_Time* (voir paragraphe affectation).
- **Appel aux *subprograms Get_Resource* et *Release_Resource*** : ces fonctions correspondent respectivement au début et fin de section critique. On effectue la même opération que pour un appel de *subprogram*. Cependant, on insère un nœud en début et un second en fin du sous-graphe pour modéliser la section critique. On initialise d'une part leur attribut *kind* respectivement avec les constantes *CriticalSectionStart* et *CriticalSectionEnd*, et d'autre part leur attribut *sharedData* avec une référence vers la donnée partagée.

**3. Traduction du graphe d'exécution en automate comportemental** Lorsque ce processus a été réalisé sur l'ensemble des tâches du modèle AADL, on réalise alors la production d'un modèle AADL d'analyse. Celui-ci remplace les descriptions comportementales des tâches par des

automates comportementaux utilisant des notations simplifiées. Ceux-ci sont ensuite interprétés par *AADLInspector* pour évaluer les écarts de performance avec le modèle non raffiné. Le procédé de traduction du graphe d'exécution en automate comportemental est le suivant :

- On produit un automate comportemental constitué de trois états $i$, $s$ et $f$ modélisant l'initialisation, l'exécution et la terminaison (état jamais atteint, en accord avec le profil Ravenscar). L'état $s$ boucle sur lui-même pour modéliser l'exécution sans fin de la tâche.
- On parcourt les nœuds du graphe d'exécution. Ce dernier étant préalablement réduit (boucles, conditionnelles), il est parcouru à la manière d'une liste. On traduit chaque nœud en instruction selon l'attribut *kind* :
  - *Compute* : le noeud est traduit en instruction *Computation*. Celle-ci prend en paramètre la valeur de l'attribut *wcet*.
  - *CriticalSectionStart* : le noeud est traduit en instruction $*!<$ (notation réduite de *Get_Resource*).
  - *CriticalSectionEnd* : le noeud est traduit en instruction $*!>$ (notation réduite de *Release_Resource*).

Le processus d'analyse est illustré par la figure 6.12. Premièrement, le graphe d'exécution est construit à partir du modèle AADL. Ensuite, celui-ci est transformé en une simple séquence afin de réduire sa complexité. Enfin, le graphe d'exécution réduit est traduit en automate comportemental AADL réduit.

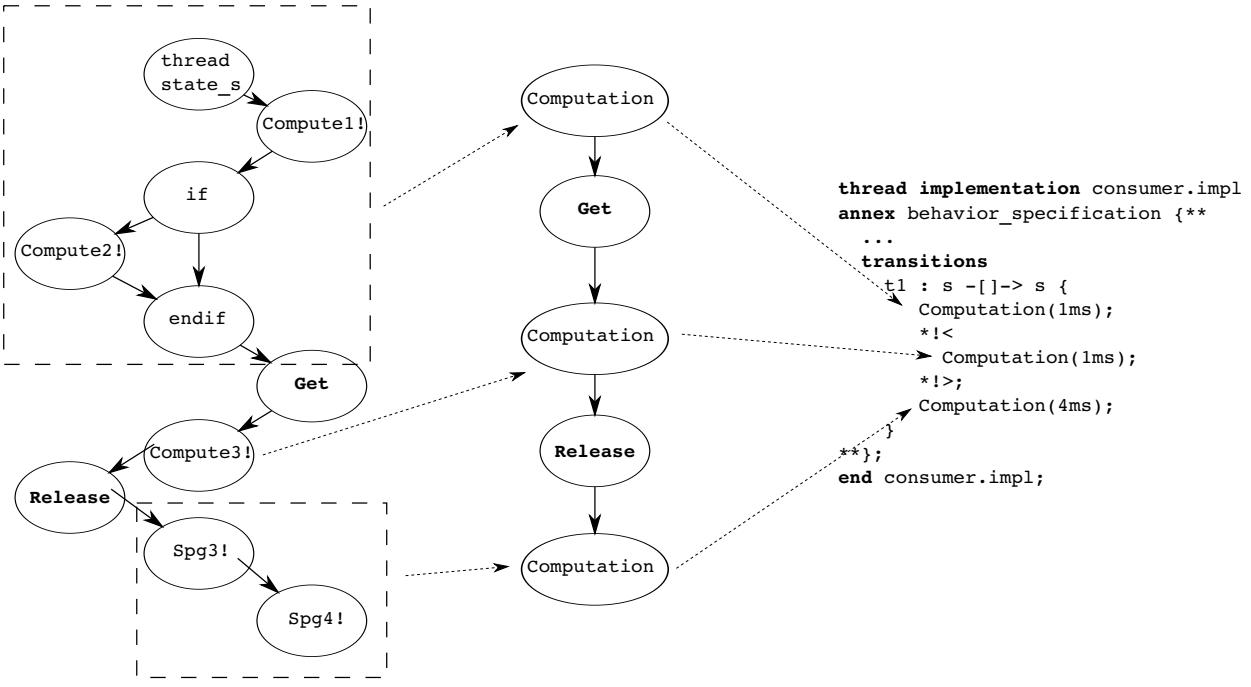



FIGURE 6.12 – Transformation du graphe d'exécution en modèle AADL d'analyse

Nous avons décrit le processus d'évaluation des surcoûts temporels lors du raffinement. Le raffinement du modèle AADL intégrant de nouveaux composants, ceux-ci sont pris en compte en construisant un graphe d'exécution intermédiaire qui est ensuite traduit en automate comportemental AADL. Le modèle AADL résultant de ce processus intègre alors les surcoûts temporels et peut être analysé. Le processus de génération peut alors être maîtrisé grâce à cette technique. La prochaine section aborde la phase finale de génération de code.

## 6.6 Génération de code

La dernière étape du processus est la génération de code exécutable. RAMSES fournit pour chaque plate-forme d'exécution un générateur de code spécifique. Le générateur respecte une architecture particulière illustrée par la figure 6.13. Le type *AadlTargetUnparser* définit la classe mère pour l'ensemble des générateurs de code développés pour RAMSES. Ce type assure une certaine organisation du code généré.

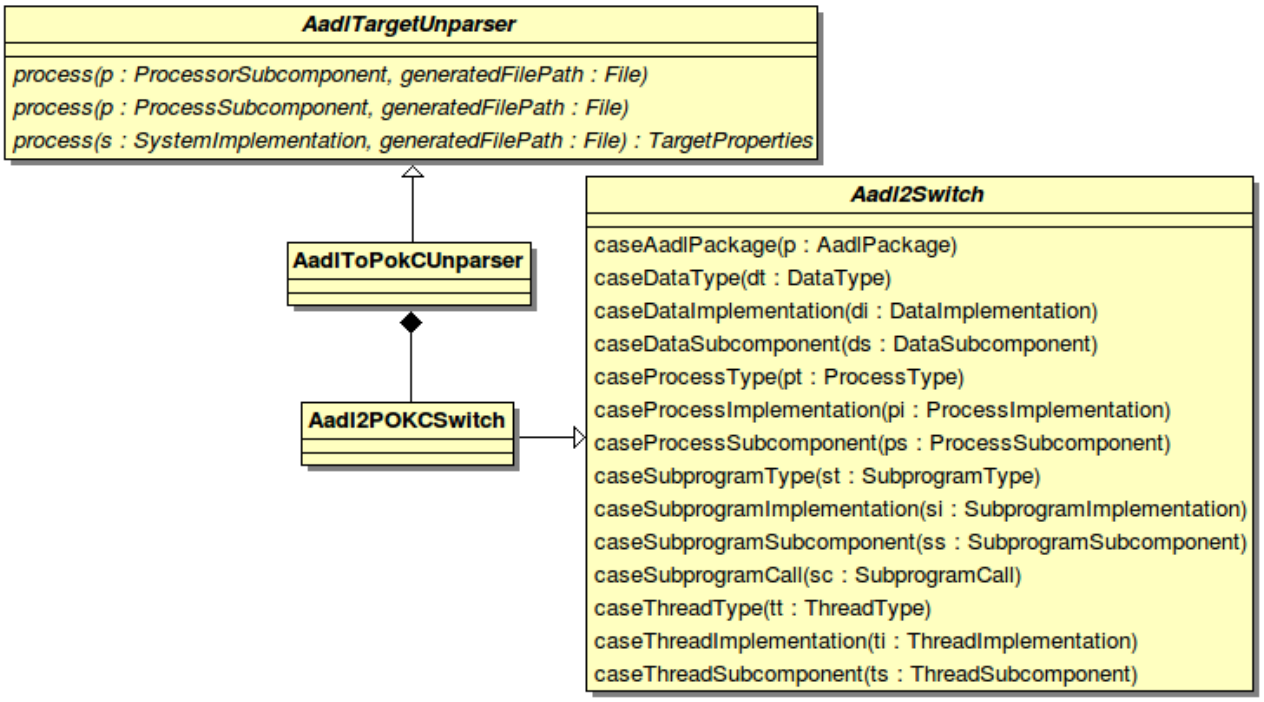



FIGURE 6.13 – Architecture des générateurs de code de RAMSES

Le code généré par RAMSES reprend l'organisation des fichiers générés fournie par Ocarina [49, 26]. Celle-ci est déterminée par les composants AADL. Les composants *system, processor et process* représentent respectivement le système global, un noeud du système (distribué), et une unité de compilation contenant des activités concurrentes. Le code généré va donc être organisé selon le schéma $< system > / < processor > / < process > /$. Le dernier répertoire, associé à un *process*, contient l'implémentation des tâches et des ressources associées à celui-ci. La classe *AadlTargetUnparser* est alors dérivée pour chaque plate-forme, par exemple *POKCUnparser* pour POK. De plus, ces classes s'aident de visiteurs dérivés de la classe *Aadl2Switch* qui spécifient le code à générer pour chaque type d'élément AADL parcouru. Cependant, contrairement à Ocarina, RAMSES factorise une grande partie de la génération de code commune à l'ensemble des plates-formes d'exécution grâce aux transformations de modèle indépendantes de la syntaxe du langage cible. Les classes héritant de *AadlTargetUnparser* ne se contentent ainsi que de traduire les éléments du modèle dans une syntaxe propre à la plate-forme visée.

Nous avons ainsi présenté le framework RAMSES que nous avons développé pour mettre en place le processus introduit dans les chapitres précédents. Nous avons décrit son architecture ainsi que son utilisation. Le prochain chapitre illustre l'application de RAMSES sur le raffinement des communications.

# Chapitre 7

# Expérimentation de RAMSES

Nous avons défini le framework RAMSES qui implémente le processus de génération de code que nous avons proposé aux chapitres précédents. Ce framework s'appuie sur la formalisation d'un *workflow* qui orchestre le processus de génération en définissant différentes stratégies de génération aux différentes étapes de raffinement. Cette démarche a été intégrée au sein de l'environnement OSATE.

Dans ce chapitre, nous expérimentons ce framework sur le raffinement des communications entre tâches. Premièrement, la section 7.1 illustre les patrons de transformation dans ce contexte particulier. Nous démontrons ainsi l'applicabilité de ces patrons sur des cas concrets d'utilisation. Ensuite, la section 7.2 donne un exemple de processus mis en œuvre par RAMSES. Nous formalisons ce processus à l'aide du modèle de *workflow* que nous avons défini aux chapitres précédents et nous démontrons l'intérêt d'un tel processus capable d'adapter la génération de code selon les propriétés du système modélisé.

## 7.1 Cas d'applications des patrons de transformation

L'utilisation des patrons de transformation a permis de faciliter la mise en place de différents processus de génération de code. Nous illustrons chacun de ces patrons sur le raffinement des communications :

- *Stratégie* est appliqué pour faire varier le nombre d'appels à un composant selon l'implémentation choisie. Ainsi, *Stratégie* considère un même ensemble d'éléments d'entrée pour lequel le raffinement va être altéré selon le contexte.
- *Adaptateur* est illustré sur le raffinement des données partagées de types hétérogènes *data* et *port*. L'interface de ces types est virtuellement modifiée pour les rendre semblables. *Adaptateur* considère des éléments d'entrée de natures différentes pour appliquer le même traitement.
- *Substitution partielle* est appliquée sur le raffinement des connexions de ports entre threads. Ce patron consiste à altérer l'ensemble des éléments d'entrée d'un raffinement afin d'exclure un sous-ensemble et à appliquer un raffinement différent pour ce dernier.
- *Memento* est appliqué sur le raffinement des ports pour apporter des preuves que la génération ne viole pas les exigences. A la règle initiale est ajoutée la production d'éléments supplémentaires pour conserver trace du raffinement.

### 7.1.1 Stratégie

Le patron *Stratégie* s'applique pour modifier une partie du comportement d'une règle de transformation. Son principe est de déléguer certains traitements ou certaines expressions à une fonction annexe que l'on remplace selon le besoin. Nous l'avons exploité pour faire varier le nombre d'appels au composant *Send_Output* selon l'implémentation au sein d'une règle principale intégrant l'ensemble des composants de communication. Cette règle est illustrée par le listing 7.1 et concerne les implémentations *PDC_Indices* et *PDC_Indices_Array*. Pour ces deux versions, le composant *Send_Output* prend en paramètre le port de réception du destinataire. Pour cette raison, la règle parcourt 1) l'ensemble des ports d'émission de l'émetteur (ligne 16) puis 2) l'ensemble des ports de réception connectés à ce dernier (ligne 18). Enfin, 3) *Send_Output* est appelé et prend en paramètre le port de réception courant (lignes 19 et 20).

```
1  rule Insert_Communication_Services_In_CallSequence (
2     thread: AADL!ComponentInstance, behavior : AADL!SubprogramCallSequence){
3    do
4    {
5      -- 1) Append calls to Receive_Input
6      for(inputPort in thread.features->select(f| f.isInput()
7              and f.isPeriodicDelayedEventDataPort())) {
8        seq <- seq.append(thisModule.resolveTemp(
9             Sequence{inputPort, behavior} ,'callReceiveInputs'));
10     }
11     -- 2) Append calls to Next_Value
12     ...
13     -- 3) Append calls to Put_Value
14     ...
15     -- 4) Append calls to Send_Output
16     for(outputPort in thread.features->select(f| f.isOutput()
17             and f.isPeriodicDelayedEventDataPort())) {
18       for(cnxInst in outputPort.srcConnectionInstance) {
19         seq <- seq.append(thisModule.resolveTemp(
20               Sequence{outputPort, behavior, cnxInst} ,'callSendOutput'));
21       }
```

```
22      }
23    }
24  }
```

Listing 7.1 – Règle de transformation introduisant les appels aux services de communication dans la séquence d'appel de chaque thread

Si nous souhaitons maintenant mettre en œuvre *PDC_InsertionSort*, il faut noter que le composant *Send_Output* ne prend aucune file de message en paramètre (voir section 5.3.4) : par conséquent il ne doit être appelé qu'une seule fois par période. La règle du listing 7.1 n'est donc pas réutilisable telle quelle. On peut cependant vouloir la réutiliser étant donné que les lignes 1 à 15 sont identiques (le nombre d'appels aux autres composants de communication étant le même entre les trois implémentations). Pour l'implémentation *PDC_InsertionSort*, les lignes 16 à 22 doivent être modifiées. Nous utilisons le patron *Stratégie* pour remplacer cette portion de code tout en conservant la logique globale de la règle existante. Cette portion de code est déléguée à une règle annexe (listing 7.2) et remplacée par un appel à cette dernière (listing 7.3).

```
1  rule Insert_Send_Output_Service_In_CallSequence (
2     thread: AADL!ComponentInstance, behavior : AADL!SubprogramCallSequence){
3    do  {
4      for(outputPort in thread.features->select(f| f.isOutput()
5              and f.isPeriodicDelayedEventDataPort())) {
6        for(cnxInst in outputPort.srcConnectionInstance) {
7          seq <- seq.append(thisModule.resolveTemp(
8                Sequence{outputPort, behavior, cnxInst} ,'callSendOutput'));
9        }
10     }
11   }
12 }
```

Listing 7.2 – Règle de transformation annexe introduisant *Send_Output* pour chaque onnexion

```
1  -- 4) Append calls to Send_Output
2  thisModule.Insert_Send_Output_Service_In_CallSequence (thread, behavior)
```

Listing 7.3 – Délégation du bloc de code (lignes 14 à 20) à une règle annexe

Il suffit alors de *superimposer* cette règle annexe pour modifier ces quelques lignes de code et ainsi obtenir un unique appel au composant *Send_Output*. Ainsi, dans le cas de *PDC_InsertionSort*, une nouvelle définition de la règle *Insert_Send_Output_Service_In_CallSequence* est *superimposée* (listing 7.4) afin de modifier le comportement de la règle *Insert_Communication_Services _In_CallSequence*. La ligne 4 du listing 7.4 insère ainsi le composant *Send_Output* une seule fois par tâche.

```
1  rule Insert_Send_Output_Service_In_CallSequence (
2     thread: AADL!ComponentInstance, behavior : AADL!SubprogramCallSequence){
3    do  {
4      seq <- seq.append (thisModule.resolveTemp(thread, 'callSendOutput'));
5    }
6  }
```

Listing 7.4 – Règle de transformation annexe introduisant *Send_Output* une fois par tâche

En appliquant *Stratégie* sur la règle *Protected_Shared_Data*, nous avons pu réutiliser une grande partie de la règle *Insert_Communication_Services _In_CallSequence* au lieu d'en définir une seconde similaire dans le cas de l'implémentation *PDC_InsertionSort*.

### 7.1.2 Adaptateur

Le patron *Adaptateur* vise à réutiliser une règle existante pour des éléments dont la logique de raffinement est similaire aux éléments d'entrée de la règle mais dont l'interface les rend incompatibles avec cette dernière. Le principe est de rendre virtuellement compatible l'interface des nouveaux éléments en définissant des fonctions annexes rattachées au type de ces éléments. Nous avons expérimenté ce patron sur le raffinement des ressources partagées entre tâches. Le modèle AADL du listing 7.5 spécifie une donnée *sharedData* (ligne 3) partagée en exclusion mutuelle entre les tâches *reader* et *writer* (ligne 5). La propriété *Concurrency_Control_Protocol* précise le protocole d'exclusion utilisé à la ligne 4.

```
1  process implementation p.impl
2    subcomponents
3      sharedData : data Base_Types::Float_32 {
4        Concurrency_Control_Protocol => PRIORITY_INHERITANCE_PROTOCOL; };
5      writer : thread writer.impl;  reader : thread reader.impl;
6    connections
7      acc1: data access sharedData -> writer.sharedData;
8      acc2: data access sharedData -> reader.sharedData;
9  end p.impl;
10
11 thread writer features
12   sharedData : requires data access Base_Types::Float_32;
13 end writer;
```

Listing 7.5 – Variable partagée et protocole d'exclusion mutuelle

Ce modèle utilise une notation implicite de l'exclusion mutuelle en se limitant à spécifier une propriété d'exclusion. Un raffinement possible d'un tel modèle serait d'expliciter les mécanismes utilisés pour délimiter les sections critiques. Ces mécanismes sont modélisés par les composants *Get_Resource* et *Release_Resource* que nous avions introduits aux chapitres précédents. La règle *Protected_Shared_Data* du listing 7.6 illustre ce raffinement : pour chaque donnée partagée, on la conserve (ligne 10 : copie de la donnée, ligne 11 : copie de son type) et on ajoute les composants *Get_Resource* et *Release_Resource* (lignes 12 et 22). Ces composants sont raffinés à partir de la ligne 12 notamment pour préciser le type de la donnée en entrée (ligne 17 : raffinement du prototype *resource_type* du composant *Get_Resource*).

```
1  rule Protected_Shared_Data {
2    from
3      e : AADL!Data (e.isSharedAndProtected())
4    using
5    {
6      protocolName : String = e.ownedPropertyAssociation->any(
7          pa|pa.property.name='Concurrency_Control_Protocol').ownedValue;
8    }
9    to
10     sharedData: AADL!DataSubcomponent (dataSubcomponentType <- sharedDataType),
11     sharedDataType: AADL!DataClassifier (name <- e.classifier.name, ...)
12     getResource: AADL!SubprogramType (
```

```
13          name <- 'Get_Resource',
14          ownedExtension <- thisModule.GetSubprogram ('Get_Resource', protocolName),
15          ownedPrototypeBinding <- Sequence {getResourceParam}
16      ),
17      getResourceParam: AADL!ComponentPrototypeBinding (
18          formal <- Get_Resource_PrototypeSpg.ownedPrototype
19                      ->any(e|e.name = 'resource_type'),
20          actual <- thisModule.CreateDataComponentPrototypeActual (sharedDataType)
21      ),
22      releaseResource: AADL!SubprogramType (...)
23  }
```

Listing 7.6 – Cas d'utilisation du patron *Adaptateur* pour factoriser le raffinement des ports et données partagées

Dans cet exemple, le patron *Adaptateur* consiste à réutiliser cette règle pour d'autres types de ressources partagées entre les tâches : les *ports*. En effet, le raffinement d'un port consiste à le traduire par exemple en un tableau accompagné de fonctions d'insertion et de suppression, ainsi que des composants *Get_Resource* et *Release_Resource* pour le protéger en exclusion mutuelle. Dans l'état actuel, la règle n'est pas réutilisable pour l'élément *port* : la propriété *Concurrency_Control_Protocol* n'étant pas applicable sur ce type d'élément (notamment par le fait que le protocole est souvent imposé par le support d'exécution dans le cas des ports et n'est pas paramétrable). Si l'on souhaite étendre la règle *Protected_Shared_Data* pour les ports, son exécution va renvoyer une erreur aux lignes 6 et 7 car le nom du protocole ne sera pas spécifié.

On définit alors deux fonctions annexes *getConcurrencyProtocol()* (listing 7.7), l'une obtenant le nom du protocole via la propriété *Concurrency_Control_Protocol* des composants *Data* (lignes 1 à 3), l'autre l'obtenant à partir d'une constante définie dans le module de transformation dans le cas des *Ports* (lignes 4 et 5). Le mot-clé *context* spécifie à quel type d'élément la fonction est rattachée.

```
1  helper context AADL!Data def : getConcurrencyProtocol() : String =
2    self.ownedPropertyAssociation->any(
3      pa|pa.property.name='Concurrency_Control_Protocol').ownedValue;
4  helper context AADL!Port def : getConcurrencyProtocol() : String =
5    'PRIORITY_CEILING_PROTOCOL';
```

Listing 7.7 – Application du patron *Adaptateur*

Les lignes 6 et 7 de la règle *Protected_Shared_Data* sont alors remplacées par un appel à la fonction annexe *getConcurrencyProtocol()*. Ainsi, ces lignes ne sont plus spécifiques aux composants *data* et sont également applicables pour les *ports*. On réécrit alors cette règle afin de la rendre générique et réutilisable pour les *data* et *ports*. Cette règle est renommée *Protected_Shared_ Resource* (listing 7.8) et reprend l'intégralité de la règle précédente. Le type d'entrée est modifié (ligne 2) par un type parent afin qu'il puisse être raffiné en port ou en data, et l'obtention du protocole (ligne 3) est également modifié afin d'utiliser les fonctions annexes précédentes.

```
1  abstract rule Protected_Shared_Resource {
2    from e : AADL!Element
3    using { protocolName: String = e.getConcurrencyProtocol(); }
4    to sharedData: AADL!DataSubcomponent (...),
```

```
5       ...
6   }
```

Listing 7.8 – Application du patron *Adaptateur* : définition d'une règle générique

Enfin, la règle générique est étendue pour préciser les spécificités de chaque type d'élément : data (listing 7.9) et port (listing 7.10).

```
1  rule Protected_Shared_Data extends Protected_Shared_Resource {
2    from e : AADL!Data (e.isSharedAndProtected())
3  }
```

Listing 7.9 – Application du patron *Adaptateur* : réécriture de la règle *Protected_Shared_Data*

Dans le cas des ports (listing 7.10), la nouvelle règle *Protected_Shared_EventDataPort* bénéficie ainsi des raffinements déjà définis (création des composants d'exclusion mutuelle) grâce à la fonction annexe *getConcurrencyProtocol()* créée précédemment. De plus, elle spécialise la règle précédente pour préciser la structure de la donnée *sharedData* qui représente le port (lignes 8 à 10 : propriétés modélisant un tableau).

```
1  rule Protected_Shared_EventDataPort extends Protected_Shared_Resource {
2    from e : AADL!EventDataPort (e.isInput())
3    to
4      sharedData: AADL!DataSubcomponent,
5      sharedDataType: AADL!DataType (
6        name <- e.name + '_Array',
7        ownedPropertyAssociation <- Sequence {
8            thisModule.CreatePA('Data_Model::Data_Representation', 'Array'),
9            thisModule.CreatePA('Data_Model::Base_Type', e.feature.dataClassifier),
10           thisModule.CreatePA('Data_Model::Dimension', e.getQueueSize())
11       }
12     ),
13 }
```

Listing 7.10 – Application du patron *Adaptateur* : extension de la règle générique

La règle générique peut être étendue pour tous les types de ressources partagées prises en compte (*e.g. Data Port* et *Event Port*) afin de factoriser la logique commune entre ces raffinements. Sans l'application de ce patron, le code de transformation aurait été dupliqué pour chaque règle.

### 7.1.3 Substitution partielle

Le patron *Substitution partielle* vise à assurer la compatibilité de plusieurs générateurs de code dont les plates-formes visées présentent des traductions différentes pour les mêmes éléments. Dans certains cas, des sous-ensembles d'éléments conduisent à des implémentations distinctes alors qu'ils seront assimilés à une unique implémentation dans d'autres cas. Nous avons expérimenté ce patron sur le partitionnement de l'espace d'adressage. Certaines plates-formes renforcent la protection des espaces d'adressage. C'est le cas des plates-formes ARINC653 qui correspondent à des architectures partitionnées. La notion de partition implique une isolation spatiale et temporelle des tâches de différentes partitions : protection contre une corruption de la mémoire causée par une autre partition et protection contre les retards éventuels de tâches de cette autre partition.

La figure 7.1 donne un modèle AADL constitué de deux espaces d'adressage *P1* et *P2* et trois tâches *P1_T1*, *P1_T2* et *P2_T1* réparties sur ces deux espaces d'adressage. La tâche *P1_T1* communique à la fois vers *P1_T2* et *P2_T1*. Dans le premier cas, *P1_T1* écrira directement dans la file de *P1_T2*. Dans le second cas, il passera par une file intermédiaire car il n'a pas le droit d'écrire dans l'espace d'adressage *P2*. Le module ARINC653 gère alors le transfert des messages de la file intermédiaire vers *P2_T1*. Pour une plate-forme d'exécution qui n'assure pas cette protection, *P1_T1* écrira par exemple directement dans la file de *P2_T1*.

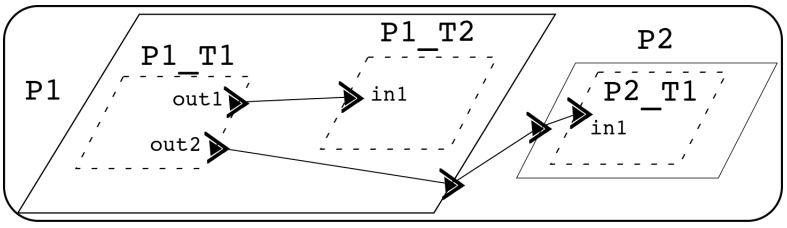
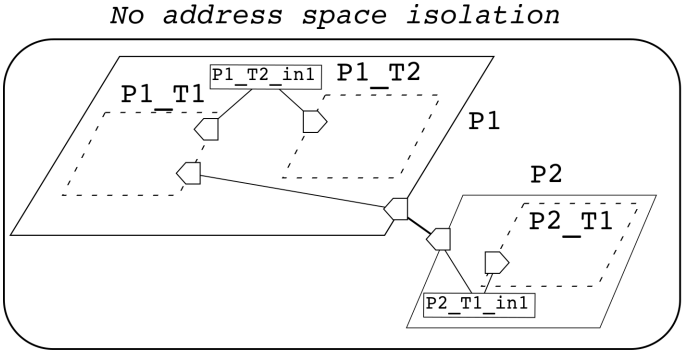



FIGURE 7.1 – Communications vers deux espaces d'adressage distincts

Cette propriété d'isolation impacte donc le raffinement des connexions. Nous distinguons deux types de connexions : *intra-process* (entre threads d'un même process) et *inter-processes* (entre threads de processes distincts). Si la propriété d'isolation est appliquée, il faut alors protéger les communications *inter-processes*. Par conséquent, chaque connexion *inter-processes* sera traduite par deux files de messages alors que chaque connexion *intra-process* sera traduite par une seule file (exemple précédent). Au contraire, dans le cas où cette propriété d'isolation n'est pas appliquée, alors toute connexion *intra-process* ou *inter-processes* sera traduite par une seule file de message. Finalement, cela revient à disposer de deux règles, l'une (R1) produisant un accès à la file de message du destinataire, l'autre (R2) produisant un accès à une file intermédiaire, et à réduire ou augmenter leur portée en fonction de la propriété d'isolation : s'il y a isolation, alors R1 se limite aux connexions *intra-process* et R2 aux connexions *inter-processes*. S'il n'y a pas isolation, R1 s'applique à l'ensemble des connexions et R2 ne s'applique pas.

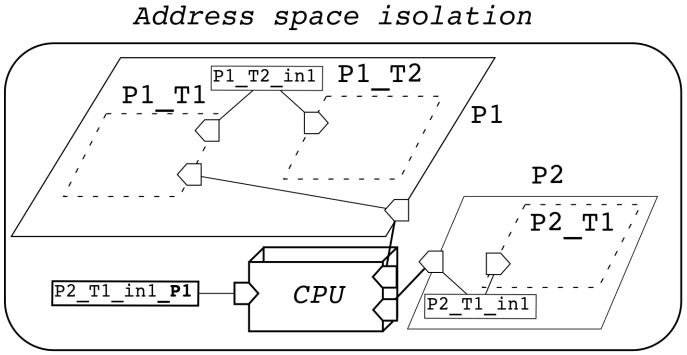
La figure 7.2 illustre la première situation : l'absence de protection (isolation) des espaces d'adressage. *R1* est appliquée à l'ensemble des connexions : la tâche émettrice *P1_T1* a accès aux files de message des destinataires (*P1_T2_in1* et *P2_T1_in1*) indépendamment du fait qu'ils soient situés dans le même espace d'adressage ou non. *R2* n'est donc pas définie dans cette situation.




```
rule R1{
 from
  c : AADL!PortConnection (
    c.isConnectedThreads() )
 to
  a : AADL!DataAccess (...),
  c2: AADL!AccessConnection(...)
}
```

FIGURE 7.2 – Raffinement des connexions : aucune protection des espaces d'adressage

Le modèle de la figure 7.3 illustre le raffinement lorsque la protection est assurée. Cette fois-ci, la connexion vers *P2_T1* doit être protégée. Elle est alors raffinée par la règle *R2* au lieu de la règle *R1* comme précédemment. Cela implique de réécrire la règle *R1* pour restreindre sa portée aux connexions *intra-process*. Une condition *isSameProcess()* est alors ajoutée à la clause *from* de *R1*. Par conséquent, *R2* s'applique sur le sous-ensemble complémentaire *not isSameProcess()*. L'application de *R2* produit ainsi une file intermédiaire *P2_T1_in1_P1* accessible à *P1_T1* et dont les messages seront transférés par le système à la file destinataire *P2_T1_in1*.




```
rule R1{
 from
  c : AADL!PortConnection (
    c.isConnectedThreads()
    and c.isSameProcess() )
 to
  a : AADL!DataAccess
      (...),
  c2: AADL!AccessConnection
      (...)
}
rule R2{
 from
  c : AADL!PortConnection (
    c.isConnectedThreads()
    and not c.isSameProcess())
 to
  -- create intermediate
  -- message queue
  d : AADL!Data (...),
  ...
}
```

FIGURE 7.3 – Raffinement des connexions : protection des espaces d'adressage

L'utilisation du patron *Substitution partielle* vise à réutiliser la règle *R1* sans avoir à la redéfinir pour y ajouter une condition supplémentaire. Les deux définitions de *R1* diffèrent par la présence ou non de la condition *isSameProcess()*. On délègue cette condition à une fonction annexe *requiresProtection()* que l'on redéfinit selon la plate-forme d'exécution. *R2* va alors s'appliquer à l'ensemble des connexions qui valident la condition *requiresProtection()* tandis que *R1* va s'appliquer sur l'ensemble complémentaire. Le listing 7.11 donne la définition de cette fonction lorsque l'isolation n'est pas assurée (ligne 1). Aucune connexion ne nécessite de protection : *R2* n'est donc jamais appliquée tandis que *R1* s'applique à toutes les connexions.

```
1 helper context AADL!PortConnection def : requiresProtection() : Boolean = false;
2 rule R1 {
3   from c : AADL!PortConnection (c.isConnectedThreads()
4                                 AND NOT c.requiresProtection())
5   ... }
```

Listing 7.11 – Délégation de la portée de *R1* à une fonction annexe au sein du module *No Isolation*

Cette première définition de *requiresProtection()* constitue le module *No Isolation* qui contient également la règle *R1* (lignes 2 à 5). Si l'isolation doit être assurée, cette fonction annexe doit être redéfinie en *superimposant* un module *Isolation* (listing 7.12). Cette redéfinition (lignes 1 et 2) augmente la portée de *R2* (préalablement nulle), et diminue celle de *R1*. Ce module contient également la définition de *R2* (lignes 3 à 6). La *superimposition* du module *Isolation* au module *No Isolation* a pour effet d'inclure la règle *R1* tout en réduisant sa portée.

```
1 helper context AADL!PortConnection def : requiresProtection() : Boolean =
2         not self.isSameProcess();
3 rule R2 {
4   from c : AADL!PortConnection (c.isConnectedThreads()
5                                 AND c.requiresProtection())
6   ... }
```

Listing 7.12 – Redéfinition de la portée de *R1* au sein du module *Isolation*

Nous avons montré l'utilisation du patron *Substitution partielle* dans un cas concret. Le processus de génération est alors configuré explicitement par l'utilisateur pour une plate-forme particulière. Cela détermine les modules de transformation à sélectionner et ainsi de redéfinir les fonctions annexes adaptant la portée des règles de transformation.

### 7.1.4 Memento

Le patron *Memento* a pour but d'aider à la traçabilité des exigences. Il apporte des preuves que le code généré ne viole pas les exigences. Pour cela, il produit des traces qui mettent en relation les éléments de la spécification initiale (le modèle initial) et les éléments raffinés modélisant le code généré. Ensuite, ces éléments sont comparés pour vérifier que certaines propriétés sont conservées. Nous avons appliqué ce patron sur l'ensemble du processus de raffinement afin de retracer l'intégralité des éléments sources : notamment pour s'assurer que chaque donnée partagée et protégée en exclusion mutuelle et pour vérifier que le dimensionnement des ressources générées correspond à celui spécifié. Nous prenons le cas du raffinement des communications à travers les *ports*. Pour chaque port est générée une file de messages (*e.g.* tableau, liste chaînée, structure quelconque). On va vouloir vérifier que la taille de la file de message générée est cohérente avec celle du modèle initial.

Notamment, dans le cas des modèles de communication que nous avons présentés en section 5.3, on vérifie que la taille est cohérente vis-à-vis des calculs de bornes. L'utilisateur peut également spécifier ces exigences, comme sur le listing 7.13. Il spécifie un composant *thread* avec une file de messages *datain* de taille 10 stockant des données de type *Float_32*.

```
1 thread consumer
2 features
3   datain: in event data port Float_32 { Queue_Size => 10; };
4 end consumer;
```

Listing 7.13 – Modélisation d'un port de taille fixée par l'utilisateur

On s'attend alors à ce que le raffinement de *datain* en tableau conserve ces propriétés comme le montre le listing 7.14. Le composant *consumer_datain_Type* modélise la file de message sous forme de donnée. Les propriétés *Data_Representation*, *Base_Type* et *Dimension* indiquent respectivement la représentation de la donnée (tableau, énumération, ...), le type des éléments contenus dans la donnée ainsi que sa taille. On souhaite obtenir des traces associant les éléments initiaux (listing 7.13) aux éléments raffinés (listing 7.14) afin de vérifier leur équivalence, et ainsi la cohérence du raffinement.

```
1  data consumer_datain_Type
2  properties
3    Data_Model::Data_Representation => Array;
4    Data_Model::Base_Type => classifier (Float_32);
5    Data_Model::Dimension => (10);
6  end consumer_datain_Type;
```

Listing 7.14 – Modélisation d'un port de taille fixée par l'utilisateur

Pour cela, on applique le patron *Memento* sur la règle réalisant le raffinement : en annotant le résultat avec une structure *TraceLink*. Dans notre exemple, il s'agit du raffinement de ports en données. Celui-ci est réalisé par la règle *Protected_Shared_Port* qui a été présentée précédemment pour le patron *Adaptateur*. Cette règle est modifiée pour produire une trace du raffinement (listing 7.15) : Un élément *trace* a été ajouté (lignes 14 à 17) afin de mettre en relation l'élément source *e* et l'élément cible *sharedDataType*. Cette trace permettra de retrouver par la suite quel élément du modèle source est à l'origine de la création de l'élément *sharedDataType*. Cette trace permettra ensuite de vérifier notamment que la taille du tableau *sharedDataType* généré à partir du port *e* est de taille correspondant à celle spécifiée par la propriété *Queue_Size*.

```
1  rule Protected_Shared_Port extends Protected_Shared_Resource {
2    from
3      e : AADL!EventDataPort
4    using { size: Integer = getIntegerProperty('Queue_Size', e); }
5    to
6      sharedDataType: AADL!DataType (
7        name <- e.name + '_Array',
8        ownedPropertyAssociation <- Sequence {
9          thisModule.CreatePA('Data_Model::Data_Representation', 'Array'),
10         thisModule.CreatePA('Data_Model::Base_Type', e.feature.dataClassifier),
11         thisModule.CreatePA('Data_Model::Dimension', size)
12       }
13     ),
14     trace: Trace!TraceLink (
15       ruleName <- 'Protected_Shared_Port',
16       sourceEntities <- Sequence {e},
17       targetEntities <- Sequence {sharedDataType}
18     )
19 }
```

Listing 7.15 – Raffinement des queueing ports : application du patron *Memento*

Pour assurer une vérification des exigences sur l'ensemble du processus de transformation, il faut pour cela ajouter ce type de trace pour chaque règle de transformation. Cependant, cet ajout peut être réalisé automatiquement à l'aide d'une transformation d'ordre supérieur : la définition initiale de la règle *Protected_Shared_Port* (listing 7.10) est alors vu comme un modèle d'entrée d'une seconde transformation afin d'ajouter l'élément *trace* et ainsi obtenir la nouvelle règle du listing 7.15. Le listing 7.16 donne un exemple de transformation d'ordre supérieur réalisant l'ajout de traces d'exécution. La transformation utilise le méta-modèle ATL pour transformer les règles ATL (nous avons utilisé un méta-modèle ATL fictif). La règle *TransformationRule* (ligne 1) prend en entrée chaque règle ATL et en produit une identique à laquelle est rajoutée la production d'une trace dans la clause *to* de cette même règle (ligne 10).

```
1  rule TransformationRule {
2    from
3      r : ATL!TransformationRule
4    to
5      r2 : ATL!TransformationRule (
6          name <- r.name,
7          kind <- r.kind,
8          fromElements <- r.fromElements,
9          oclCondition <- r.oclCondition,
10         toElements   <- r.toElements->union(Sequence{trace}),
11         doElements   <- r.doElements ),
12     trace: ATL!OutputElement (
13         name <- 'trace',
14         type <- thisModule.GetType('Trace!TraceLink'),
15         attributeMapping <- Sequence {attrRuleName,attrFrom,attrTo} ),
16     attrRuleName: ATL!AttributeMapping (
17         name   <- 'ruleName',
18         values <- Sequence{r.name} ),
19     attrFrom: ATL!AttributeMapping (
20         name   <- 'sourceEntities',
21         values <- r.fromElements ),
22     attrTo: ATL!AttributeMapping (
23         name   <- 'targetEntities',
24         values <- r.toElements )
25 }
```

Listing 7.16 – Définition d'une transformation d'ordre supérieur

Nous avons présenté des cas d'applications pour les différents patrons de transformation introduits aux chapitres précédents. Ceux-ci simplifient le développement des générateurs de code grâce à la *superimposition* et à la réécriture des règles de transformation. Nous avons illustré ces patrons sur le raffinement des communications. La prochaine section décrit la troisième contribution sur la mise en œuvre de ce processus de raffinement des communications.

## 7.2 Processus de raffinement des communications

Dans cette section, nous illustrons notre troisième contribution : la *génération de composants de communication adaptés*. Premièrement, nous précisons en section 7.2.1 la structuration des raffinements en modules de transformations. Deuxièmement, nous expliquons en section 7.2.2 comment nous déterminons statiquement un ordre d'application de ces stratégies. Troisièmement, en section 7.2.3 nous formalisons le processus de raffinement selon l'ordre de sélection et la structuration des raffinements. Nous nous appuyons sur le méta-modèle proposé en section 5.1. Enfin, nous fournissons en section 7.2.4 différents scénarios afin d'illustrer l'application de chaque alternative : chacune est validée puis invalidée selon les contraintes du modèle initial.

### 7.2.1 Structuration des raffinements

Le processus de génération de code se constitue en particulier d'un ensemble d'étapes de raffinements des communications. Nous avons introduit trois approches différentes des communications

différées : *PDC_Indices*, *PDC_Indices_Array* et *PDC_InsertionSort*. Les deux premières correspondent à l'implémentation sans verrou que nous avons proposée aux chapitres précédents. Pour la première, les indices d'écriture/lecture sont calculés à l'exécution tandis qu'ils sont stockés en mémoire pour la seconde. La troisième correspond à une implémentation classique pour laquelle chaque file de message est une liste chaînée protégée par un verrou.

Les trois stratégies de raffinement des communications partagent cependant une logique commune. Par conséquent, celles-ci sont définies par un ensemble de modules de transformation *superimposés* et partagés. La figure 7.4 illustre la composition de ces stratégies.

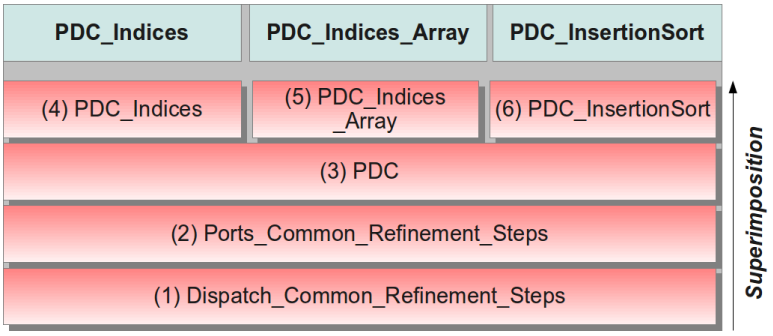



FIGURE 7.4 – Communications : structuration en modules de raffinement

– Le module 1 ***Dispatch_Common_Refinement_Steps*** correspond à l'introduction d'un gabarit d'automate comportemental pour chaque tâche selon le type d'activation de celle-ci (*e.g.* périodique, sporadique). Ce gabarit d'automate inclura par la suite des appels à différents composants de communication.
– Le module 2 ***Ports_Common_Refinement_Steps*** fournit des raffinements génériques d'éléments *port* en variables partagées dont le type est précisé par les modules suivants.
– Ensuite, le module 3 ***PDC*** contient les raffinements communs à l'ensemble des implémentations : composants *data* modélisant différentes constantes (*e.g.* taille de chaque port), le type des variables partagées modélisant les *ports* est raffiné en tampon circulaire.
– Enfin, les modules 4 et 5 (***PDC_Indices***, ***PDC_Indices_Array***) précisent les raffinements spécifiques pour les deux implémentations sans verrou. Le module 4 introduit les constantes et les fonctions nécessaires au calcul d'indice (*e.g.* identifiant de tâche, période, échéance, priorité) alors que le module 5 introduit les tableaux d'indices.
– Pour terminer, le module 6 ***PDC_InsertionSort*** est spécifique à la stratégie *PDC_InsertionSort*. Il spécialise le tampon circulaire en liste chaînée dont chaque message est daté. Il introduit également les composants associés à la liste chaînée (*e.g.* insertion triée) ainsi que les sections critiques (appels aux composants *Get_Resource*/*Release_Resource*).

La section suivante aborde l'ordre dans lequel les stratégies de raffinements sont testées.

### 7.2.2 Ordre de sélection de l'implémentation

L'objectif du processus de raffinement des communications est d'assurer une maîtrise du coût du code généré. Nous avons proposé trois implémentations alternatives. Chacune remplit des ob-

jectifs différents. D'une part, *PDC_InsertionSort* est une implémentation simple (insertion triée, verrous) qui ne nécessite aucune structure supplémentaire que la file de message. Par conséquent, celle-ci est assez simple à déboguer. Au contraire, les deux implémentations *PDC_Indices* et *PDC_Indices_Array* nécessitent des algorithmes et des structures de données supplémentaires qui complexifient l'analyse et le débogage du code exécuté. Cependant, ces deux alternatives ne nécessite aucun verrou et ont un coût potentiellement moindre que la première. Le tableau 7.1 compare les trois alternatives sur les éléments impactant l'empreinte mémoire et le temps d'exécution.

| | **PDC_Indices** | **PDC_Indices_Array** | **PDC_InsertionSort** |
|---|---|---|---|
| Débogage | Difficile | Difficile | Facile |
| Empreinte mémoire (par file) | Taille file donnée en section 5.3.4 | Idem PDC_Indices + Tableaux d'indices. Taille du code minimale (affectations). | Propriété *Queue_Size* (AADL). |
| Surcoût temporel (par file) | Calcul des indices | *Aucun* | Insertion triée (N) et verrous |

TABLE 7.1 – Éléments spécifiques à chaque alternative

Ainsi, le premier objectif que nous nous fixons consiste à fournir une implémentation simple à analyser et à déboguer. Cependant, si celle-ci représente un coût trop important, alors on s'oriente vers une implémentation optimisée mais plus difficile à déboguer. La prochaine section formalise un tel processus.

### 7.2.3 Formalisation du processus de raffinement

Cette section décrit le processus de raffinement des communications. Le processus (figure 7.5) est constitué d'une première étape de raffinement dans laquelle le squelette de l'architecture logicielle est généré. Ensuite, le processus raffine les communications avec l'objet *Loop* qui sélectionne l'une des trois implémentations alternatives, chacune étant constituée de quatre modules de transformations *superimposés* (voir 7.2.1). Chaque implémentation est évaluée par une analyse d'ordonnancement puis par une analyse d'empreinte mémoire. L'implémentation est considérée valide si ces deux analyses réussissent (séquence de type *Conjunction*). Si l'implémentation courante est validée, le processus traduit le modèle raffiné en code exécutable (étape *Generation*). Dans le cas où aucune des trois implémentations n'a été validée, le processus entre dans un état d'erreur (étape *ErrorState*).

La prochaine section illustre l'exécution de ce processus et l'application des différentes stratégies en fonction du modèle d'entrée.

### 7.2.4 Évaluation

On dispose de plusieurs implémentations alternatives des mécanismes de communication. Le processus modélisé précédemment va alors tester ces alternatives dans différentes situations afin d'illustrer la validation puis l'échec de chacune.

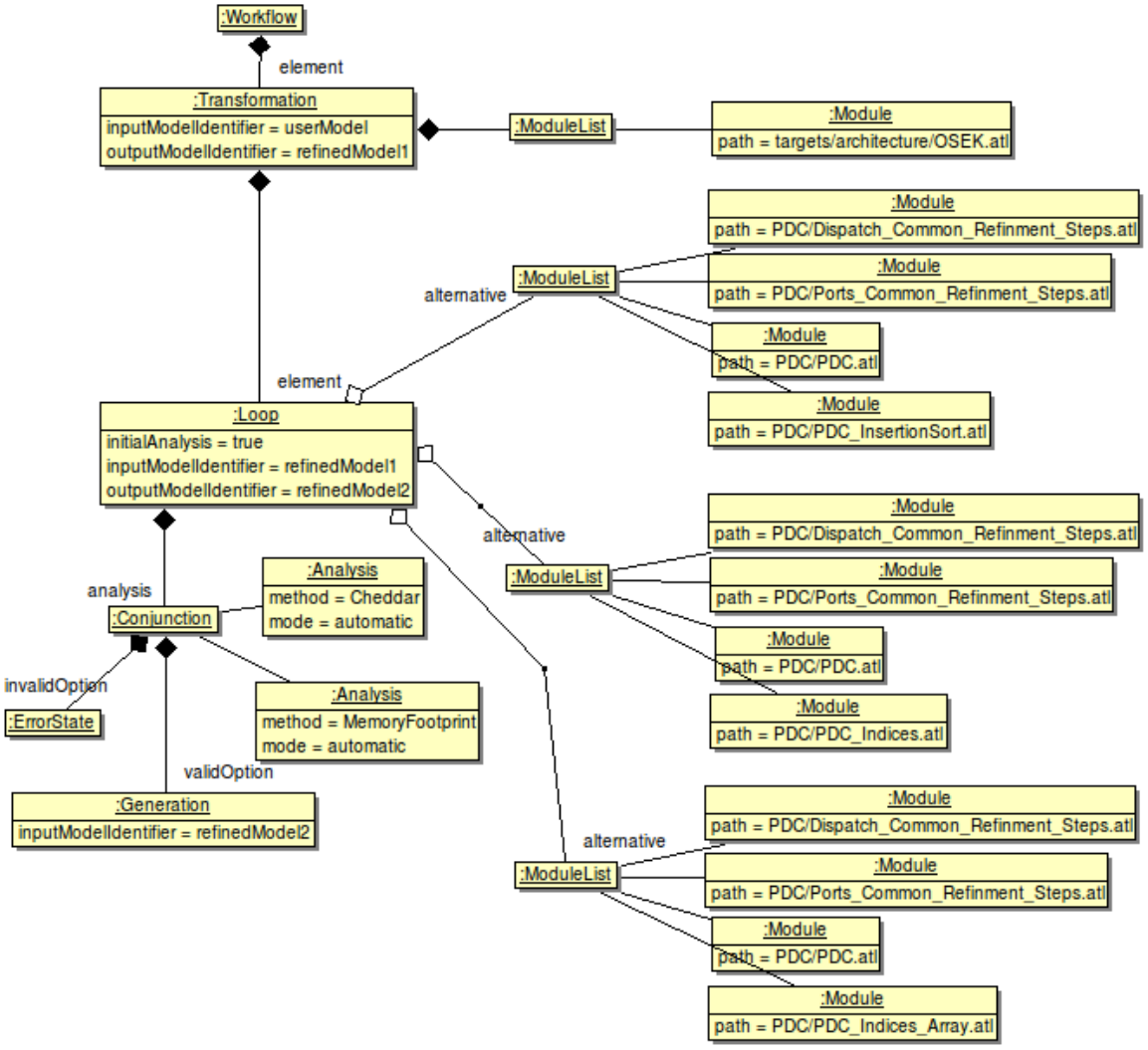



FIGURE 7.5 – Modélisation du processus de raffinement des communications

Pour évaluer notamment le temps d'exécution des modèles AADL, le processus d'évaluation nécessite une estimation des performances du système. Nous renseignons le modèle AADL initial avec une estimation des vitesses de lecture et d'écriture en mémoire. Sur la figure 7.17, nous indiquons ces propriétés pour un processeur de 200 MHz (lignes 3 et 4).

```
1 memory mainmem
2 properties
3  Read_Time  => [Fixed => 300 ns; PerByte => 300 ns;];
4  Write_Time => [Fixed => 300 ns; PerByte => 300 ns;];
5  ...
6 end mainmem;
```

Listing 7.17 – Expérimentations : modélisation des performances du système en AADL

**Jeu de tâches** Nous considérons un jeu de quatre tâches périodiques *T1*, *T2*, *T3* et *T4* ordonnancées selon *Rate Monotonic*. Le listing 7.18 spécifie les périodes et échéances des tâches ainsi que les échanges de messages, chaque message étant codé sur 4 octets. Les durées d'exécution seront fixées lors des différents scénarios proposés aux paragraphes suivants.

```
1  process implementation p.impl
2  subcomponents
3    T1: thread T1 {Period => 7 ms;  Deadline => 7 ms;};
4    T2: thread T2 {Period => 12 ms; Deadline => 12 ms;};
5    T3: thread T3 {Period => 20 ms; Deadline => 20 ms;};
6    T4: thread T4 {Period => 25 ms; Deadline => 25 ms;};
7  connections
8    cnx1: port T1.dataout -> T2.datain;  cnx2: port T1.dataout -> T3.datain;
9    cnx3: port T2.dataout -> T4.datain;
10   cnx4: port T3.dataout -> T4.datain;
11 end p.impl;
```

Listing 7.18 – Expérimentations : modélisation des communications

Nous fournissons au processus des modèles AADL basés sur ce jeu de tâches pour illustrer l'application des différentes stratégies d'implémentation. Nous faisons varier les propriétés telles que le temps d'exécution initial des tâches (*Compute_Execution_Time*) et le dimensionnement mémoire (*Word_Count*) pour aboutir à différentes implémentations.

**Scénario 1 : PDC_InsertionSort** Le premier scénario considère le WCET des tâches fixé à $C_{T1} = C_{T2} = C_{T3} = 1$ ms et $C_{T4} = 2$ ms. On obtient alors des pires temps de réponse respectivement de 1,2,3 et 5 ms : ce jeu de tâche est ordonnançable. En tenant compte du surcoût temporel lié à l'implémentation *PDC_InsertionSort*, le WCET des tâches est réévalué : on tient compte du temps nécessaire à l'envoi et à la réception d'un message. On obtient alors $C'_{T1} = 1.76$ ms, $C'_{T2} = C'_{T3} = 1.69$ ms et $C'_{T4} = 2.63$ ms. Le surcoût pour *T1* est plus important que pour *T2* et *T3* car le surcoût en émission est plus important qu'en réception. La nouvelle analyse d'ordonnancement réévalue les pires temps de réponse respectivement à 2, 4, 6 et 11 ms en se basant sur la borne supérieure des temps d'exécutions. Le système reste ordonnançable après prise en compte du surcoût temporel lié à l'implémentation *PDC_InsertionSort*. De plus, concernant l'empreinte mémoire, la taille du code généré est de 323 Ko. Cette stratégie est validée si le dimensionnement initial est suffisant.

**Scénario 2 : PDC_Indices** Ce second scénario considère un jeu de tâches similaire avec des durées d'exécution plus importantes : $C_{T1} = C_{T2} = C_{T3} = 2$ ms et $C_{T4} = 3$ ms. Ce nouveau modèle est ordonnançable. En reprenant la stratégie précédente *PDC_InsertionSort*, on réévalue ces temps d'exécution. Ainsi, on obtient $C'_{T1} = 2.76$ ms, $C'_{T2} = C'_{T3} = 2.69$ ms et $C'_{T4} = 3.63$ ms. Cela conduit à un pire temps de réponse plus pessimiste que précédemment : notamment pour *T4* on obtient 37 ms, signifiant que cette tâche a raté ses échéances. Le système n'est plus ordonnançable.

La stratégie suivante, *PDC_Indices*, est alors testée sur ce jeu de tâches. Cette nouvelle implémentation a un surcoût temporel moindre que la précédente mais a une empreinte mémoire plus importante, avec un surcoût global de 15 Ko de plus que la précédente (fonctions de communication plus volumineuses). Cette stratégie est validée notamment si le dimensionnement mémoire, spécifié initialement via la propriété *Word_Count*, est supérieur à la taille totale du code généré (338 Ko).

**Scénario 3 : PDC_Indices_Array** Le scénario précédent est donc invalidé si le dimensionnement mémoire initial est insuffisant. Dans ce cas, la troisième stratégie, *PDC_Indices_Array* est testée. Cette dernière a une empreinte mémoire moindre, s'expliquant par les fonctions de communication minimalistes qui nécessitent peu de code. Par conséquent la taille totale du code généré passe de 338 à 247 Ko. De plus, le surcoût temporel est quasiment nul. Cette dernière implémentation est donc la moins coûteuse en terme de temps d'exécution et d'empreinte mémoire. Cependant, le code généré est plus difficile à déboguer qu'avec les stratégies précédentes.

**Scénario 4 : Etat d'erreur** La dernière implémentation, *PDC_Indices_Array*, peut également être invalidée dans le cas d'un dimensionnement mémoire insuffisant. Le processus, ne disposant pas d'autres stratégies, entre dans un état d'erreur pour signaler l'incapacité à fournir une stratégie d'implémentation qui assure le respect des contraintes de temps et de dimensionnement mémoire.

Ces quatre scénarios permettent ainsi d'explorer les différentes branches du processus afin d'assurer le respect des contraintes du modèle. Nous avons donc montré une application concrète du processus de raffinement incrémental proposé et nous avons illustré sa capacité à adapter sa stratégie selon les contraintes du modèle.

# Chapitre 8

# Conclusions et Perspectives

Dans ce manuscrit, nous nous sommes intéressé à la problématique de la maîtrise du code généré pour systèmes embarqués temps réel critiques (SETRC). Nous avons mis en avant la difficulté d'assurer la conservation des exigences du modèle initial jusqu'au code final. Nous avons proposé un processus de génération de code basé sur les techniques de l'Ingénierie Dirigée par les Modèles afin de modéliser le code généré et d'évaluer son impact sur les performances. Une méthodologie a été également proposée pour faciliter la mise en œuvre d'un tel processus. Ce chapitre rappelle l'ensemble de nos contributions et conclut notre travail. Nous abordons ensuite les perspectives de notre travail de recherche.

## 8.1 Rappel des contributions

Pour répondre à la problématique, nous avons proposé un processus de génération de code basé sur des raffinements de modèle afin d'intégrer les éléments d'implémentation au sein du modèle et ainsi évaluer leur impact sur les performances du système. Pour assurer une démarche d'analyse cohérente, nous favorisons l'utilisation des mêmes outils de validation aux différentes étapes de raffinement. Pour cela, nous avons utilisé un langage de modélisation (AADL) qui supporte différents niveaux d'abstraction et qui dispose d'un format compatible avec un certain nombre d'outils d'analyse. La maîtrise du code généré s'appuie sur l'orchestration de différentes stratégies de raffinement, chacune étant appliquée et évaluée. La conservation des états successifs du modèle nous permet l'annulation de raffinements invalidés.

### 8.1.1 Processus d'analyse et de transformation

Nous avons proposé un formalisme pour modéliser un tel processus de génération de code. Au sein du processus, les étapes de raffinements du modèle sont entrelacées d'étapes d'analyse qui évaluent l'impact du raffinement sur les performances. Afin d'adapter la stratégie selon l'impact du raffinement sur les performances, chaque analyse spécifie la suite du processus dans le cas où celle-ci réussit ou échoue. Nous avons fourni un méta-modèle modélisant ce type de processus de manière concise, notamment pour définir différentes alternatives possibles à chaque étape du raffinement. Les processus existants qui intègrent des phases d'analyse afin de guider la stratégie de

génération de code ne couvrent pas ou peu les aspects comportementaux (*e.g.* simulation d'ordonnancement). Par conséquent, notre framework permet à l'utilisateur d'inclure ses propres outils d'analyse au sein du processus.

Le processus se constitue des éléments suivantes :
- **Transformation** : raffinement produisant un nouveau modèle de plus bas niveau d'abstraction.
- **Analysis** : analyse réalisée par un programme tierce ou par un module interne. L'analyse spécifie deux branches dont chacune représente la suite du processus selon le verdict donné.
- **Loop** : enchaînement de différentes tentatives de raffinement à une étape donnée.
- **Serialization** : sauvegarde d'un modèle intermédiaire dans un fichier accessible à l'utilisateur.
- **Generation** : génération de code à partir du modèle raffiné, selon la plate-forme sélectionnée au préalable.
- **ErrorState** : état d'erreur. Typiquement, lorsqu'une analyse échoue et qu'aucune alternative n'est envisagée, le processus stoppe et provoque une erreur.

### 8.1.2 Patrons de transformation

La mise en place d'un tel processus peut être relativement coûteuse. D'une part, par le développement de stratégies alternatives de raffinement. D'autre part, par la prise en compte de différents supports d'exécution qui nécessitent des raffinements spécifiques. Pour réduire ce coût, nous avons proposé des patrons de transformation dont l'objectif est de factoriser le code de génération entre les différentes stratégies et entre les différents supports d'exécution. Pour cela, nous nous approprions de manière originale la technique existante de *superimposition* qui définit la transformation en plusieurs couches.

Nous avons proposé les patrons suivants :
- **Stratégie** : altère une règle existante afin de modifier son comportement.
- **Substitution partielle** : réduit la portée d'une règle afin d'effectuer un raffinement différent sur un sous-ensemble d'éléments.
- **Adaptateur** : augmente la portée d'une règle sur de nouveaux éléments en rendant leur interface virtuellement compatible.
- **Memento** : produit des traces de transformation en ajoutant à chaque règle des éléments supplémentaires.

### 8.1.3 Génération de composants adaptés

Malgré la volonté d'optimiser le code généré pour maîtriser les performances du système, le support d'exécution impose ses propres composants logiciels qui ne sont pas toujours adaptés dans toutes les situations. Nous avons alors proposé de générer les composants logiciels et de les adapter selon les propriétés du système modélisé. Ce travail nous a permis de proposer des algorithmes de gestion de stockage de données originaux afin de consolider le déterminisme et la consommation de ressource. Ces composants sont ainsi modélisés pour être pris en compte dans le processus de validation. Nous avons expérimenté cette démarche sur les composants de communication pour lesquels nous avons proposés différentes implémentations.

## 8.2 Conclusions

Notre approche a été mise en œuvre au sein de l'environnement de développement OSATE. Nous avons développé le framework RAMSES au sein d'OSATE afin d'orchestrer les transformations de modèles AADL à l'aide du langage de transformation ATL.

### 8.2.1 Contribution autour des processus de transformation de modèle

Le processus de transformation proposé fournit des mécanismes d'analyse et de transformation qui ne sont pas exploités dans les processus existants, notamment dans le cadre des SETRC. Nous avons proposé une démarche de transformation visant à maîtriser le coût du code généré. De plus nous avons également proposé une méthodologie pour faciliter la mise en oeuvre d'un tel processus.

### 8.2.2 Expérimentation à l'aide d'AADL et d'ATL

Nous avons démontré la faisabilité d'une telle approche en la mettant en œuvre à l'aide d'un langage de modélisation et d'un langage de transformation. Nous avons expérimenté cette approche au sein d'OSATE pour la génération de code autour de différentes plates-formes d'exécution.

## 8.3 Perspectives

Nous abordons maintenant les perspectives de notre travail, notamment sur les améliorations du processus proposé. Le processus de génération tel qu'il est modélisé actuellement impose certaines limitations en terme d'orchestration des stratégies et d'adaptation du code généré.

### 8.3.1 Optimisation du code généré

La stratégie globale de génération est définie par l'ensemble des choix réalisés aux différentes étapes du processus. Une condition suffisante pour aboutir au système final est d'assurer le respect des contraintes. La stratégie globale n'est donc pas nécessairement optimale. L'obtention d'une stratégie optimale nécessiterait d'explorer l'ensemble des combinaisons possibles et cela aurait un impact important sur le temps d'exécution du processus. Cependant, une autre solution consisterait à garantir une certaine marge sur des propriétés particulières (comme le temps de réponse) en définissant une analyse spécifique qu'il suffirait d'intégrer au processus (l'analyse échouant si la marge est insuffisante, obligeant le processus à trouver une alternative).

### 8.3.2 Temps d'exploration des solutions

L'évaluation d'une stratégie de génération nécessite actuellement de réaliser la transformation et ensuite d'analyser le modèle raffiné. Le temps d'exploration des solutions doit alors prendre en

compte le temps d'exécution de chaque transformation, pouvant être non négligeable. Une amélioration possible serait de traduire automatiquement chaque transformation en un modèle d'analyse avant de commencer l'exécution du processus global. L'impact de la transformation serait ainsi connu statiquement et ne nécessiterait pas l'exécution de la transformation. On peut également s'intéresser à la cohérence de la succession des choix réalisés tout au long du processus. Actuellement, cette cohérence repose sur la capacité de l'utilisateur à identifier les phases de raffinement. Des règles de compatibilité pourraient être définies entre différents raffinements et élimineraient les choix incompatibles avec ceux réalisés aux étapes précédentes.

### 8.3.3 Granularité des stratégies de transformation

Dans le processus actuel, chaque stratégie de transformation s'applique sur un ensemble d'élément déterminé statiquement : par exemple, la stratégie d'implémentation des communications entre tâches s'applique à l'ensemble des tâches et des communications. Pour un système distribué, on va potentiellement vouloir appliquer des stratégies différentes à chaque nœud du système. Également, au sein d'un même nœud, on va souhaiter optimiser le code généré en définissant une implémentation différente pour chaque tâche selon ses performances évaluées au cours du processus (*e.g.* temps de réponse). Pour réaliser ce type de raffinement, une solution possible serait d'injecter des annotations au modèle pour filtrer les éléments sur lesquels seront appliquées les transformations.

# Publications


[1] Fabien Cadoret, Thomas Robert, Etienne Borde, Laurent Pautet et Frank Singhoff, *Deterministic Implementation of Periodic-Delayed Communications and Experimentation in AADL*, ISORC 2013

[2] Fabien Cadoret, Etienne Borde, Sébastien Gardoll et Laurent Pautet, *Design Patterns for Rule-based Refinement of Safety Critical Embedded Systems Models*, ICECCS 2012

[3] Fabien Cadoret, Laurent Pautet et Thomas Robert, *Extraction de contraintes temporelles d'un modèle d'architecture temps-réel*, École d'Été Temps Réel 2011

# Références


[1] Aadl inspector homepage http ://www.ellidiss.com/products/aadl-inspector/.

[2] Eclipse project. homepage http ://www.eclipse.org.

[3] The jamda project. homepage http ://jamda.sourceforge.net/, 2003.

[4] Absint company homepage www.absint.com, 2005.

[5] Bound-t tool homepage http ://www.bound-t.com/, 2005.

[6] Atlflow. homepage http ://opensource.urszeidler.de/atlflow/, 2013.

[7] D. H. Akehurst, B. Bordbar, M. J. Evans, W. G. J. Howells, and K. D. McDonald-Maier. Sitra : simple transformations in java. In *Proceedings of the 9th international conference on Model Driven Engineering Languages and Systems*, MoDELS'06, pages 351–364, Berlin, Heidelberg, 2006. Springer-Verlag.

[8] James H. Anderson, Srikanth Ramamurthy, and Kevin Jeffay. Real-time computing with lock-free shared objects. *ACM Trans. Comput. Syst.*, 15(2) :134–165, May 1997.

[9] C. Aussagues and V. David. A method and a technique to model and ensure timeliness in safety critical real-time systems. In *Engineering of Complex Computer Systems, 1998. ICECCS '98. Proceedings. Fourth IEEE International Conference on*, pages 2–12, 1998.

[10] T. P. Baker. Stack-based scheduling for realtime processes. *Real-Time Syst.*, 3(1) :67–99, April 1991.

[11] Daniel Balasubramanian, Anantha Narayanan, Christopher van Buskirk, and Gabor Karsai. The graph rewriting and transformation language : Great. *Electronic Communications of the EASST*, 1, 2006.

[12] Steffen Becker. Coupled model transformations. In *Proceedings of the 7th international workshop on Software and performance*, WOSP '08, pages 103–114, New York, NY, USA, 2008. ACM.

[13] Xavier Blanc, Isabelle Mounier, Alix Mougenot, and Tom Mens. Detecting model inconsistency through operation-based model construction. In *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*, pages 511–520. IEEE, 2008.

[14] Jacek Blazewicz. Scheduling dependent tasks with different arrival times to meet deadlines. In *Proceedings of the International Workshop Organized by the Commision of the European Communities on Modelling and Performance Evaluation of Computer Systems*, pages 57–65, Amsterdam, The Netherlands, The Netherlands, 1977. North-Holland Publishing Co.

[15] D. Blouin, E. Senn, and S. Turki. Defining an annex language to the architecture analysis and design language for requirements engineering activities support. In *Model-Driven Requirements Engineering Workshop (MoDRE), 2011*, pages 11–20, 2011.

[16] Eric Bruneton, Thierry Coupaye, and Jean-Bernard Stefani. The fractal composition framework. *Proposed Final Draft of Interface Specification Version 0.9, The ObjectWeb Consortium*, 2002.

[17] Frank Budinsky, Stephen A. Brodsky, and Ed Merks. *Eclipse Modeling Framework*. Pearson Education, 2003.

[18] Giorgio C. Buttazzo. *Hard Real-time Computing Systems : Predictable Scheduling Algorithms And Applications (Real-Time Systems Series)*. Springer-Verlag TELOS, Santa Clara, CA, USA, 2004.

[19] Jean Bézivin, Christian Brunette, Régis Chevrel, Frédéric Jouault, and Ivan Kurtev. Bridging the generic modeling environment (gme) and the eclipse modeling framework. In *In Proceedings of the OOPSLA Workshop on Best Practices for Model Driven Software Development*, 2005.

[20] H. Rebecca Callison. A time-sensitive object model for real-time systems. *ACM Trans. Softw. Eng. Methodol.*, 4 :287–317, July 1995.

[21] H. Chetto, M. Silly, and T. Bouchentouf. Dynamic scheduling of real-time tasks under precedence constraints. *Real-Time Syst.*, 2(3) :181–194, September 1990.

[22] Hyeonjoong Cho, Binoy Ravindran, and E. Douglas Jensen. Space-optimal, wait-free real-time synchronization. *IEEE Trans. Comput.*, 56(3) :373–384, March 2007.

[23] G. Csertdn, G. Huszerl, I. Majzik, Z. Pap, A. Pataricza, and D. Varro. Viatra - visual automated transformations for formal verification and validation of uml models. In *Automated Software Engineering, 2002. Proceedings. ASE 2002. 17th IEEE International Conference on*, pages 267–270, 2002.

[24] Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003.

[25] Vincent David, Christophe Aussaguès, Stéphane Louise, Philippe Hilsenkopf, Bertrand Ortolo, and Christophe Hessler. The oasis based qualified display system. *Fourth American Nuclear Society International Topical Meeting on Nuclear Plant Instrumentation, Controls and Human-Machine Interface Technologies (NPIC&HMIT 2004), Columbus, Ohio, USA*, page 11, 2004.

[26] J. Delange, L. Pautet, and F. Kordon. Code Generation Strategies for Partitioned Systems. In *29th IEEE Real-Time Systems Symposium (RTSS'08)*, pages 53–56, Barcelona, Spain, December 2008. IEEE Computer Society.

[27] Julien Delange and Laurent Lec. Pok, an arinc653-compliant operating system released under the bsd license. *13th Real-Time Linux Workshop*, 10 2011.

[28] Rachid Djenidi, C. Lavarenne, Ramine Nikoukhah, Y. Sorel, and S. Steer. From hybrid system simulation to real-time implementation. In *Proceedings of 11th European Simulation Symposium and Exhibition, ESS'99*, Erlangen-Nuremberg, Germany, October 1999.

[29] Airlines Electronic Engineering. Avionics application software standard interface. Technical report, Aeronautical Radio, INC, 1997.

[30] Horn F. and Delpiano F. The kilim configuration framework. homepage http ://kilim.ow2.org/, 2002.

[31] KORDON Fabrice, HUGUES Jérôme, CANALS Agusti, and DOHET Alain. *Modélisation et analyse de systèmes embarqués*. Lavoisier, 2013.

[32] Peter Feiler. Open source aadl tool environment (osate). In *AADL Workshop, Paris*, 2004.

[33] Sanford Friedenthal, Alan Moore, and Rick Steiner. *A practical guide to SysML : the systems modeling language*. Access Online via Elsevier, 2011.

[34] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns : elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[35] Ning Ge and Marc Pantel. Time properties verification framework for uml-marte safety critical real-time systems. In *ECMFA*, pages 352–367, 2012.

[36] Anna Gerber, Michael Lawley, Kerry Raymond, Jim Steel, and Andrew Wood. Transformation : The missing link of mda. In *Proceedings of the First International Conference on Graph Transformation*, ICGT '02, pages 90–105, London, UK, UK, 2002. Springer-Verlag.

[37] Olivier Gilles and Jérôme Hugues. Applying wcet analysis at architectural level. In Raimund Kirner, editor, *WCET*, volume 8 of *OASICS*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2008.

[38] Olivier Gilles and Jérôme Hugues. Expressing and enforcing user-defined constraints of aadl models. In Radu Calinescu, Richard F. Paige, and Marta Z. Kwiatkowska, editors, *ICECCS*, pages 337–342. IEEE Computer Society, 2010.

[39] T. Grandpierre, C. Lavarenne, and Y. Sorel. Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors. In *Proceedings of 7th International Workshop on Hardware/Software Co-Design, CODES'99*, Rome, Italy, May 1999.

[40] T. Grandpierre and Y. Sorel. From algorithm and architecture specification to automatic generation of distributed real-time executives : a seamless flow of graphs transformations. In *Proceedings of First ACM and IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE'03*, Mont Saint-Michel, France, June 2003.

[41] Alexandre Hamez, Lom Hillah, Fabrice Kordon, Alban Linard, Emmanuel Paviot-Adet, Xavier Renault, and Yann Thierry-Mieg. New features in cpn-ami 3 : focusing on the analysis of complex distributed systems. In *ACSD*, pages 273–275. IEEE Computer Society, 2006.

[42] M. González Harbour, J. J. Gutiérrez García, J. C. Palencia Gutiérrez, and J. M. Drake Moyano. Mast : Modeling and analysis suite for real time applications. In *In 13th Euromicro Conference on Real-Time Systems*, page 125, 2001.

[43] Zef Hemel, Lennart C. L. Kats, and Eelco Visser. Code generation by model transformation. In *Proceedings of the 1st international conference on Theory and Practice of Model Transformations*, ICMT '08, pages 183–198, Berlin, Heidelberg, 2008. Springer-Verlag.

[44] Thomas A. Henzinger and Christoph M. Kirsch. The embedded machine : Predictable, portable real-time code. *ACM Trans. Program. Lang. Syst.*, 29(6), 2007.

[45] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Trans. Program. Lang. Syst.*, 15(5) :745–770, November 1993.

[46] Benjamin Horowitz. *Giotto : a time-triggered language for embedded programming*. PhD thesis, 2003. AAI3121523.

[47] Hai Huang, Padmanabhan Pillai, and Kang G. Shin. Improving wait-free algorithms for interprocess communication in embedded real-time systems. In *Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*, ATEC '02, pages 303–316, Berkeley, CA, USA, 2002. USENIX Association.

[48] J. Hugues and B. Zalila. PolyORB High Integrity User's Guide. Technical report, École Nationale Supérieure des Télécommunications, jan 2007.

[49] Jerome Hugues, Bechir Zalila, Laurent Pautet, and Fabrice Kordon. From the prototype to the final embedded system using the ocarina aadl tool suite. *ACM Trans. Embed. Comput. Syst.*, 7 :42 :1–42 :25, August 2008.

[50] Jean-Marc Jézéquel, Olivier Barais, and Franck Fleurey. Model driven language engineering with kermeta. In *Proceedings of the 3rd International Summer School Conference on Generative and Transformational Techniques in Software Engineering III*, GTTSE'09, pages 201–221, Berlin, Heidelberg, 2011. Springer-Verlag.

[51] Frédéric Jouault and Ivan Kurtev. Transforming models with atl. In *Proceedings of the 2005 International Conference on Satellite Events at the MoDELS*, MoDELS'05, pages 128–138, Berlin, Heidelberg, 2006. Springer-Verlag.

[52] Frédéric Jouault. Loosely coupled traceability for atl. In *In Proceedings of the European Conference on Model Driven Architecture (ECMDA) workshop on traceability*, pages 29–37, 2005.

[53] Christoph M. Kirsch, Marco A. A. Sanvido, Thomas A. Henzinger, and Wolfgang Pree. A giotto-based helicopter control system. In Alberto L. Sangiovanni-Vincentelli and Joseph Sifakis, editors, *EMSOFT*, volume 2491 of *Lecture Notes in Computer Science*, pages 46–60. Springer, 2002.

[54] Mark H. Klein, Thomas Ralya, Bill Pollak, Ray Obenza, and Michael González Harbour. *A practitioner's handbook for real-time analysis*. Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[55] H. Kopetz and G. Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1) :112–126, 2003.

[56] Hermann Kopetz. Event-triggered versus time-triggered real-time systems. In *Proceedings of the International Workshop on Operating Systems of the 90s and Beyond*, pages 87–101, London, UK, UK, 1991. Springer-Verlag.

[57] Jernej Kovse and Theo Härder. Generic xmi-based uml model transformations. In *Proceedings of the 8th International Conference on Object-Oriented. Information Systems*, OOIS '02, pages 192–198, London, UK, UK, 2002. Springer-Verlag.

[58] Jernej Kovse and Theo Härder. Mt-flow–an environment for workflow-supported model transformations in mda. In *Advanced Information Systems Engineering*, pages 160–174. Springer, 2004.

[59] G. Lasnier, L. Pautet, and J. Hugues. A model-based transformation process to validate and implement high-integrity systems. In *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2011 14th IEEE International Symposium on*, pages 67–74, 2011.

[60] Gilles Lasnier. *Une Approche Intégrée pour la Validation et la Génération de Systèmes Critiques par Raffinement Incrémental de Modèles Architecturaux*. PhD thesis, Télécom ParisTech, 2012.

[61] Michael Lawley and Jim Steel. Practical declarative model transformation with tefkat. In Jean-Michel Bruel, editor, *Satellite Events at the MoDELS 2005 Conference*, volume 3844 of *Lecture Notes in Computer Science*, pages 139–150. Springer Berlin Heidelberg, 2006.

[62] Stephane Louise, Matthieu Lemerre, Christophe Aussagues, and Vincent David. The oasis kernel : A framework for high dependability real-time systems. In *Proceedings of the 2011 IEEE 13th International Symposium on High-Assurance Systems Engineering*, HASE '11, pages 95–103, Washington, DC, USA, 2011. IEEE Computer Society.

[63] F. Mallet, C. Andre, and J. DeAntoni. Executing aadl models with uml/marte. In *Engineering of Complex Computer Systems, 2009 14th IEEE International Conference on*, pages 371–376, June 2009.

[64] Frédéric Mallet. Clock constraint specification language : specifying clock constraints with uml/marte. *Innovations in Systems and Software Engineering*, 4(3) :309–314, 2008.

[65] Raphaël Marvie. A transformation composition framework for model driven engineering. 2004.

[66] Stephen J. Mellor and Marc Balcer. *Executable UML : A Foundation for Model-Driven Architectures*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[67] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. *Electron. Notes Theor. Comput. Sci.*, 152 :125–142, March 2006.

[68] Joaquin Miller and Jishnu Mukerji. Model driven architecture (MDA). Draft ormsc/2001-07-01, Architecture Board ORMSC, July 2001.

[69] Pierre-Alain Muller, Franck Fleurey, Didier Vojtisek, Zoé Drey, Damien Pollet, Frédéric Fondement, Philippe Studer, and Jean-Marc Jézéquel. On Executable Meta-Languages applied to Model Transformations. In *Model Transformations In Practice Workshop*, Montego Bay, Jamaïque, October 2005.

[70] OMG. *White Paper on the Profile Mechanism*. Object Management Group, 1999.

[71] OMG. *Meta Object Facility (MOF) Specification*. Object Management Group, 2000.

[72] OMG. UML profile for EJB, 2001.

[73] OMG. *Unified Modeling Language Specification : Version 1.4*. OMG, Needham, 2001.

[74] OMG. UML profile for CORBA specification, 2002.

[75] OMG. Uml profile for schedulability, performance, and time, 2005.

[76] OMG. *XML Metadata Interchange (XMI)*. OMG, 2007.

[77] OMG. Uml profile for marte : Modeling and analysis of real-time embedded systems, 2009.

[78] OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.1, January 2011.

[79] Richard F. Paige, Alan Hartman, and Arend Rensink, editors. *Model Driven Architecture - Foundations and Applications, 5th European Conference, ECMDA-FA 2009, Enschede, The Netherlands, June 23-26, 2009. Proceedings*, volume 5562 of *Lecture Notes in Computer Science*. Springer, 2009.

[80] Q. Pan, T. Gautier, L. Besnard, and Y. Sorel. Signal to syndex : Translation between synchronous formalisms. internal report, 2003. Internal report, INRIA, Rocquencourt, France, 2003.

[81] Renaud Pawlak, Laurence Duchien, Gérard Florin, and Lionel Seinturier. Jac : A flexible solution for aspect-oriented programming in java. In *Metalevel architectures and separation of crosscutting concerns*, pages 1–24. Springer, 2001.

[82] José E Rivera, Daniel Ruiz-Gonzalez, Fernando Lopez-Romero, José Bautista, and Antonio Vallecillo. Orchestrating atl model transformations. *Proc. of MtATL*, pages 34–46, 2009.

[83] As-2 Embedded Computing Systems Committee SAE. Architecture Analysis & Design Language (AADL). SAE Standards n$^o$ AS5506, November 2004.

[84] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols : An approach to real-time synchronization. *IEEE Trans. Comput.*, 39(9) :1175–1185, September 1990.

[85] F. Singhoff, J. Legrand, L. Nana, and L. Marcé. Cheddar : a flexible real time scheduling framework. In *Proceedings of the 2004 annual ACM SIGAda international conference on Ada : The engineering of correct and reliable software for real-time & distributed systems using Ada and related technologies*, SIGAda '04, pages 1–8, New York, NY, USA, 2004. ACM.

[86] Frank Singhoff, Alain Plantec, and Pierre Dissaux. Can we increase the usability of real time scheduling theory ? the cheddar project. In *Reliable Software Technologies–Ada-Europe 2008*, pages 240–253. Springer, 2008.

[87] C. U. Smith and L. G. Williams. *Performance Solutions : a practical guide to creating responsive, scalable software*. Addison-Wesley, 2002.

[88] David B Stewart. Measuring execution time and real-time performance. In *Embedded Systems Conference (ESC)*, 2001.

[89] Gabriel Tamura and Anthony Cleve. A Comparison of Taxonomies for Model Transformation Languages. *Paradigma*, 4(1) :1–14, March 2010.

[90] J. Delcoigne C. Aussaguès C. Cordonnier V. David, M. Aji. Le modèle de conception oasis/yc pour les systèmes temps-réel complexes critiques. In *Real-Time & Embedded Systems Conference*, 1996.

[91] Bert Vanhooff, Dhouha Ayed, Stefan Van Baelen, Wouter Joosen, and Yolande Berbers. Uniti : A unified transformation infrastructure. In *Model Driven Engineering Languages and Systems*, pages 31–45. Springer, 2007.

[92] A. Vicard and Y. Sorel. Formalization and static optimization for parallel implementations. In *Proceedings of Workshop on Distributed and Parallel Systems, DAPSYS'98*, Budapest, Hungary, September 1998.

[93] Didier Vojtisek and Jean-Marc Jézéquel. MTL and Umlaut NG - Engine and Framework for Model Transformation. *ERCIM News 58*, 58, 2004.

[94] Dennis Wagelaar, Massimo Tisi, Jordi Cabot, and Frédéric Jouault. Towards a general composition semantics for rule-based model transformation. In *MoDELS*, pages 623–637, 2011.

[95] Edward D Willink. Umlx : A graphical transformation language for mda. In *A. Rensink (Ed.) Proceedings of the Workshop on Model Driven Architecture : Foundations and Applications*, pages 13–24, 2003.