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.

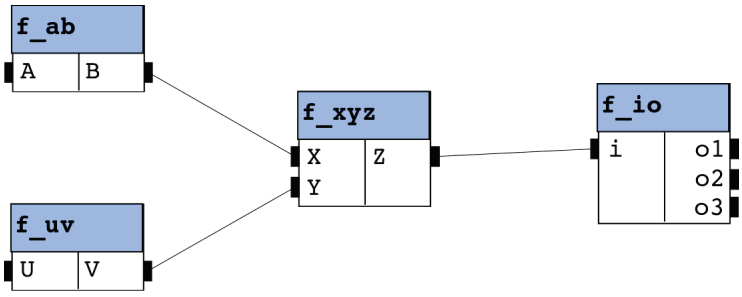

FIGURE 2.1 – Représentation du flot d'exécution avec SynDEx

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 : ψC et ψAda [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_Ω. 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_Ω. Les transitions sont annotées des instructions réalisées, comme le montre la figure 2.2.

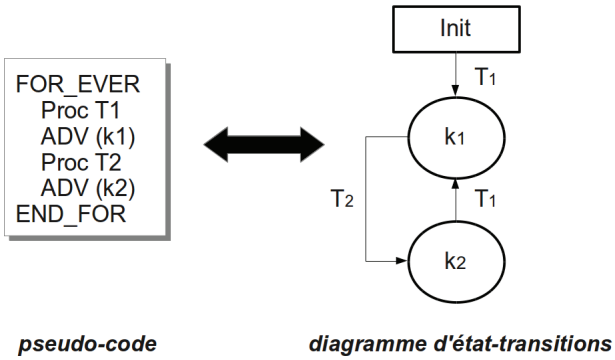

FIGURE 2.2 – Représentation d'une tâche définie avec OASIS

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).

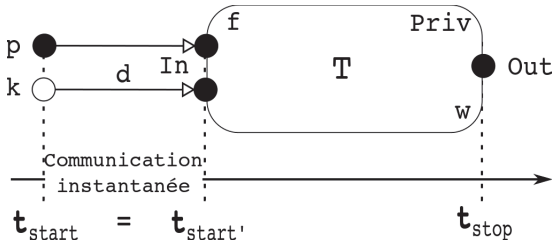

FIGURE 2.3 – Représentation d'une tâche définie avec Giotto

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.

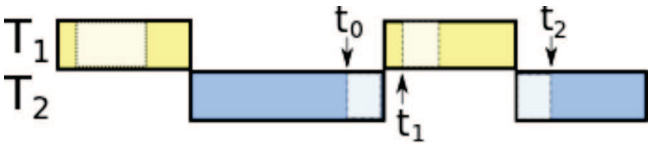

FIGURE 2.4 – Time-Sensitive Object : absence de blocage entre lecteur et écrivain

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).

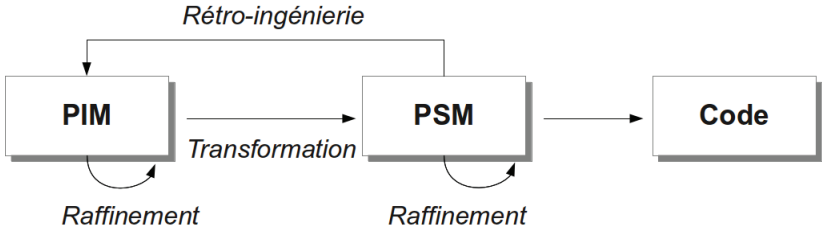

FIGURE 2.5 – Représentation des différents types de transformation

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.

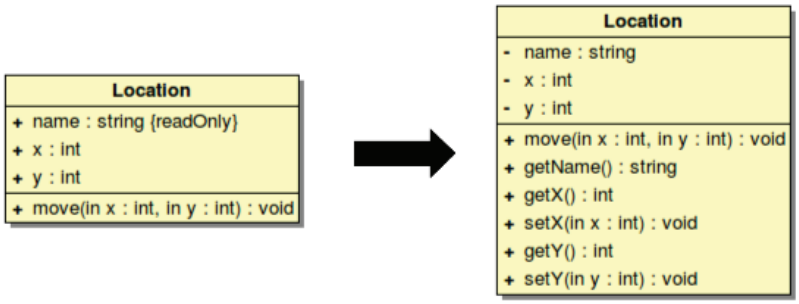

FIGURE 2.6 – Exemple de raffinement d'une classe UML : traduction des attributs publiques en privés

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.

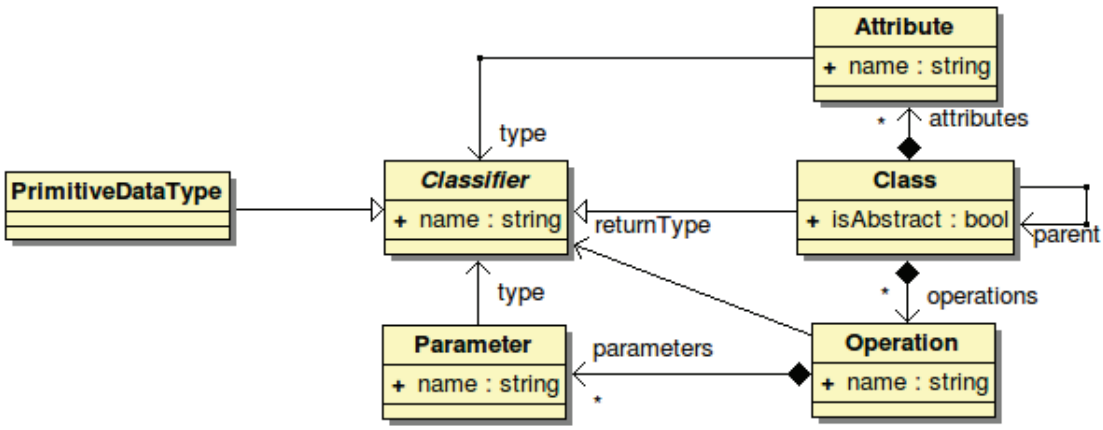

FIGURE 2.7 – Extrait d'un méta-modèle définissant le langage UML

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*).

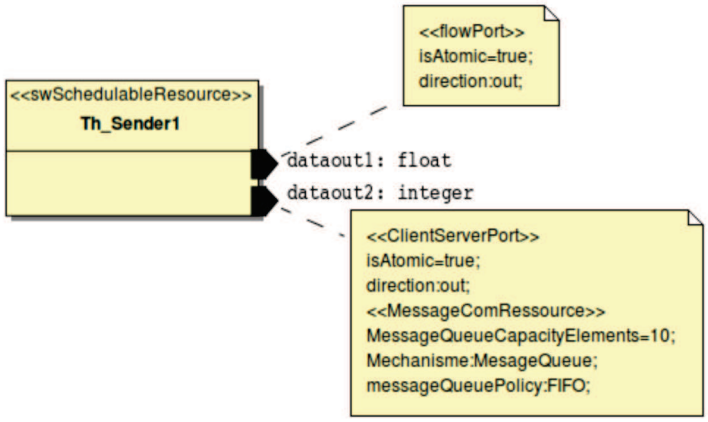

FIGURE 2.8 – Définition d'une tâche en MARTE : spécification de son interface

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*.

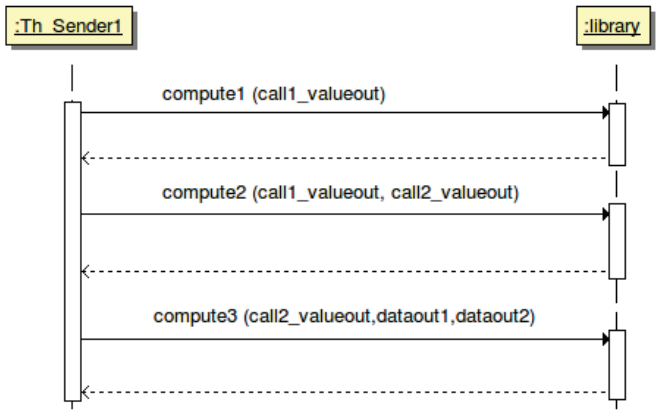

FIGURE 2.9 – Définition d'une tâche en MARTE : spécification du comportement

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.

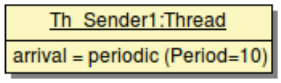

FIGURE 2.10 – Définition d'une tâche en MARTE : ajout de contraintes temporelles

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.

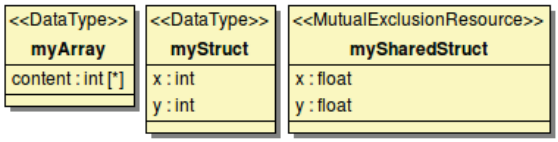

FIGURE 2.11 – Modélisation des types de données en MARTE

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*.

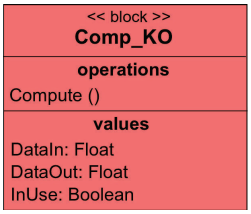

FIGURE 2.12 – Définition d'un bloc en SysML

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.

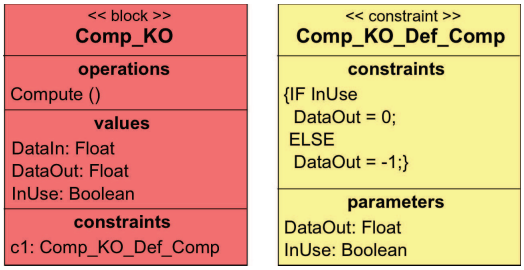

FIGURE 2.13 – Définition d'un bloc en SysML : ajout de contraintes

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.

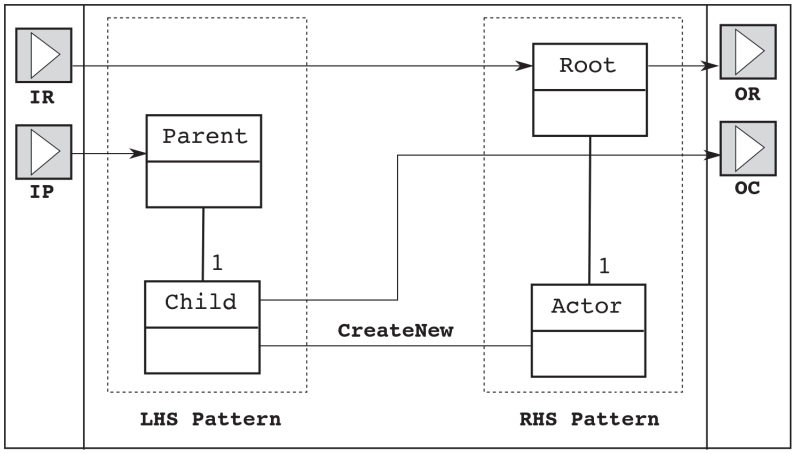

FIGURE 2.14 – Langage de transformation GReAT : modélisation d'une règle

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*).

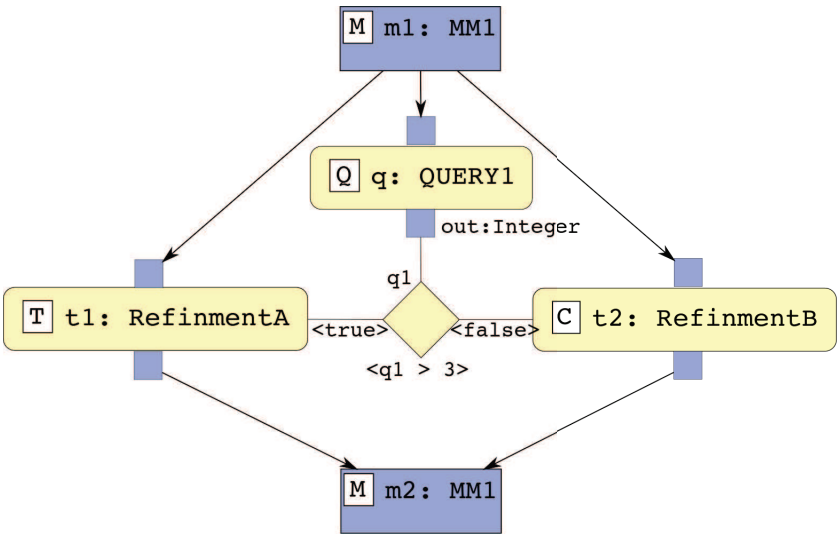

FIGURE 2.15 – Langage Wires* : modélisation d'un processus de transformation

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.

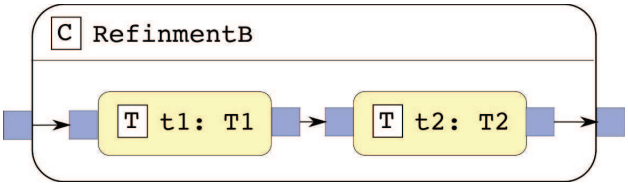

FIGURE 2.16 – Langage Wires* : modélisation d'une étape composite

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.

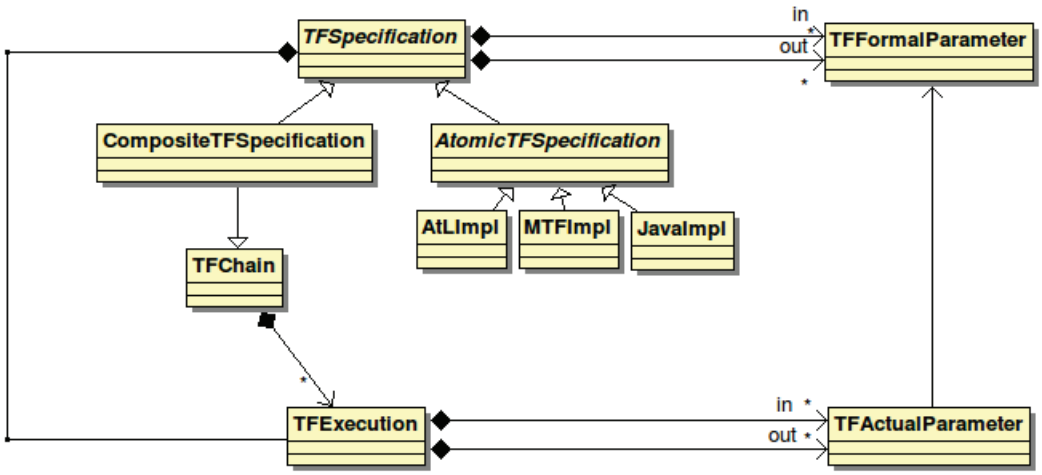

FIGURE 2.17 – Langage UniTI : extrait du méta-modèle UTR

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

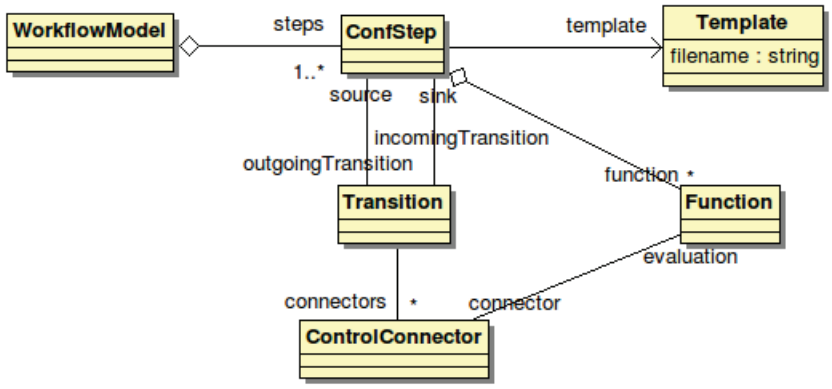

FIGURE 2.18 – Langage MT-Flow : extrait du méta-modèle

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.