N° d'ordre : 4181





MODÉLISATION CONJOINTE LOGICIEL/MATÉRIEL DE SYSTÈMES TEMPS RÉEL

DOCTORAT EN INFORMATIQUE

par

Safouan TAHA

Soutenue le 5 mai 2008

Eric RUTTEN,	INRIA Rhônes-Alpes,	Rapporteur
Jean-Philippe BABAU	INSA Lyon,	Rapporteur
Francois TERRIER,	CEA LIST,	Examinateur
Pierre BOULET,	INRIA Lille,	Examinateur
Jean-Luc DEKEYSER,	INRIA Lille,	Directeur
Ansgar RADERMACHER,	CEA LIST,	Encadrant
Sebastien GERARD,	CEA LIST,	Encadrant

Université des Sciences et Technologies de Lille — Ecole doctorale SPI

À ma chère famille,

مَن لم يذُقْ مُرَّ التعلُّمِ ساعةً تجَرَّعَ ذُلَّ الجهْلِ طولَ حياته

الإمام الشافعي [1]

[1] Vers de L'imâm ach-Châfiî : *Celui qui n'a pas goûté à la dureté de l'apprentissage une heure, s'empoisonnera de son ignorance toute sa vie.*

Table des matières

CHAPITRE 1

Introduction

1 Contexte et problématique

Tout travail de thèse traitant de systèmes embarqués ne peut s'empêcher de citer la loi de Moore concernant la densité matérielle, puisqu'elle impose une cadence exponentielle que les techniques de conception actuelles ne peuvent maintenir. Nous n'allons point rompre avec cet usage mais nous allons citer une version généralisée et intéressante de la loi de Moore, avancée par Ian Phillips (ARM) [1] et que nous illustrons par la figure 1.1.

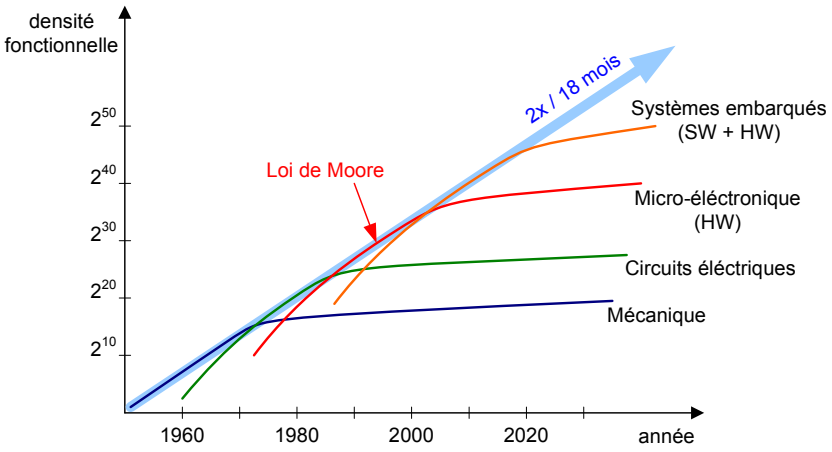


FIG. 1.1 – Loi de Moore généralisée

Avec un certain optimisme, Ian Phillips nous avance un constat, un aveu et une prévision :

- Il constate que la loi de Moore n'est pas spécifique à la micro-électronique. Auparavant, plusieurs disciplines scientifiques telle la mécanique ou l'électronique, ont toutes connu séparément et pendant les années où les avancées scientifiques leur ont été consacrées, une croissance dans la densité fonctionnelle de leurs systèmes équivalente à celle de la loi de Moore. En effet, on a doublé tous les 18 mois et depuis toujours, le nombre de fonctionnalités dans un même volume.
- Ian Phillips confirme également que les limites technologiques de l'industrie des semi-conducteurs (taille de gravure) sont déjà atteintes et que la loi de Moore n'est désormais plus vérifiée. La micro-éléctronique connaîtra donc un ralentissement dans la croissance de la densité de ses systèmes. L'histoire se répète.
- Finalement, Il prévoit que les systèmes embarqués hétérogènes (logiciel/matériel) pourront dans les années à venir maintenir le rythme de la loi de Moore en terme de densité fonctionnelle. Comme la densité matérielle enregistre un ralentissement, nous en déduisons que l'efficacité dans la conception de la composante logicielle avec une prise en charge de plus en plus amont de l'architecture matérielle, est seul garant d'une telle évolution dans la durée.

Depuis les années 80, les développeurs du logiciel profitaient confortablement de l'augmentation régulière de la performance des processeurs. Cette situation décrite par la loi de Moore, fût maintenue grâce au progrès continu dans l'industrie du semi-conducteur. On pouvait assembler plus de transistors dans moins d'espace et cadencer le tout à des fréquences plus élevées. Non seulement ce redoublement de performance matérielle ne remettait pas en cause la structure du logiciel qui demeurait inchangée, mais en plus, il

se transformait en redoublement de performance logicielle, puisque les applications étaient deux fois plus performantes. En effet, le développement des mono-processeurs avait pour seul but, l'exécution de plus en plus rapide de programmes, en augmentant la fréquence des processeurs, en multipliant et optimisant leurs pipelines, en améliorant les politiques de branchement, et en agrandissant les caches. Aujourd'hui, il est évident que la performance des mono-processeurs se stabilise et cela est principalement dû aux limites physiques et technologiques freinant la diminution de la taille de gravure. Toute l'industrie du matériel a pris une nouvelle direction pour maintenir son rythme de croissance des performances, elle s'oriente désormais vers des processeurs multi-coeurs et des plateformes multi-processeurs. La performance de ces multi-ressources de calcul ne correspond plus au débit d'instructions programme (comme c'était le cas des mono-processeurs), mais elle est mesurée dans l'absolu par la somme des débits d'instructions des différents coeurs et processeurs. Si on veut que la loi de Moore demeure vérifiée du moins fonctionnellement (densité des fonctionnalités du système), la composante logicielle des systèmes embarqués ne peut plus rester inchangée. Une adéquation logiciel/matériel est impérative.

Afin de concevoir des systèmes embarqués exponentiellement performants dans la durée, il nous faudra répondre à trois problématiques :

1. **Conception du logiciel** : Comment concevoir des applications de plus en plus modulaires et parallèles ?
2. **Conception du matériel** : Comment décrire, concevoir, simuler et implanter efficacement des plateformes matérielles distribuées et multi-processeurs ?
3. **Prise en charge du matériel** : Comment communiquer l'architecture matérielle au flot de conception du logiciel ? comment adapter et allouer le logiciel à cette architecture ?

2 Contribution

Le développement logiciel bénéficie globalement des avancées en matière de modélisation orientée-objet et composants. La méthodologie Accord/UML développée au sein de notre équipe apporte un certain nombre de solutions quant à la modélisation d'applications temps-réel. Elle prend en compte la modularité au niveau de la structure et le parallélisme au niveau des services. La méthodologie Accord/UML répond ainsi à notre première problématique (voir ci-dessus). L'objectif de cette thèse sera de répondre aux deux problématiques restantes.

I. Adoption de l'ingénierie dirigée par les modèles dans la conception, simulation et implantation de la plateforme matérielle, afin d'améliorer et de faciliter son développement et de communiquer les décisions architecturales :
    1. Définition d'un langage de modélisation HRM (Hardware Resource Model) pour la description de plateformes matérielles sous différentes vues et à différents niveaux de détail. HRM fait partie du profil MARTE récemment standardisé par l'OMG.
    2. Conception d'une méthodologie de modélisation du matériel en HRM pour garantir une construction de modèles consistants.

3. Développement d'un outillage complet et automatisé pour la simulation des plateformes matérielles ainsi modélisées.
4. Unification entre HRM et les standards d'implantation du matériel dans un même processus de développement.

II. Prise en charge du matériel dans le développemment des systèmes embarqués temps-réel :

1. Réalisation d'un métamodèle détaillant la structure des applications temps-réel modélisées par la méthodologie ACCORD/UML et permettant d'identifier, une à une, leurs entités logicielles.
2. Spécification des règles et des contraintes d'allocation qui régissent les placements des entités logicielles sur les ressources matérielles. Puis développement de mécanismes d'adéquation pour adapter des configurations à priori inadéquates.
3. Description du processus de validation des configurations d'allocation avec prise en charge des propriétés non-fonctionnelles.
4. Développement d'une chenille de robots comme cas d'étude illustrant dans un même flot de conception, toutes les contributions de ce travail de thèse.

3 Plan

Dans le chapitre 2 suivant, nous commencerons par introduire l'ingénierie dirigée par les modèles et nous nous attarderons particulièrement sur le langage de modélisation UML. Ensuite, nous établirons un état de l'art couvrant toutes les étapes du processus de développement des systèmes embarqués temps-réel depuis la spécification jusqu'à l'intégration et la validation, en passant par le développement conjoint de l'application temps-réel d'un côté et du matériel embarqué de l'autre. Nous finirons par présenter quelques bases de la robotique mobile.

Nous consacrerons respectivement les deux chapitres 3 et 4 aux deux axes de contribution énumérés précédemment. Chaque chapitre comprendra quatre sections correspondant aux quatre contributions de l'axe en question. Finalement en chapitre 5, nous conclurons ce travail de thèse et nous en expliciterons d'éventuelles perspectives.

Ce document contient d'innombrables figures, réalisées avec grand soin. Ces figures font partie intégrante du texte et manifestent une clarté qui nous dispensera de s'étendre longuement sur leur description. Nous invitons donc le lecteur à prendre le temps de les assimiler indépendamment du texte, lequel se contente parfois de n'en souligner que quelques aspects. Un effort particulier fût également apporté dans l'illustration de nos travaux par de nombreux exemples et divers cas d'études. Nous les déploierons au fur et à mesure des contributions avancées.

CHAPITRE 2

Positionnement

En plaçant les modèles au centre du cycle de développement, l'Ingénierie Dirigée par les Modèles (IDM) vient répondre à la complexité croissante des systèmes, en haussant le niveau d'abstraction, en séparant les préoccupations, puis en améliorant l'expressivité, la lisibilité et la réutilisation. Cette efficacité se traduit par des gains en temps et en argent.

Les systèmes embarqués temps-réel sont des systèmes complexes dont le développement doit tenir compte de contraintes de temps et de ressources limitées en termes de puissance de calcul, de mémoire et de consommation d'énergie. Le processus de développement de ces systèmes hybrides (logiciel/matériel) se compose généralement de plusieurs flots de conception qui évoluent parallèlement en gardant une forte adéquation entre eux.

Le nouveau standard MARTE a pour but d'adopter l'IDM dans le développement des systèmes embarqués temps-réel. Le profil MARTE définit un grand nombre de concepts nécessaires à la modélisation détaillée et à l'analyse des systèmes embarqués temps-réel. En appliquant MARTE, l'utilisateur profitera d'une riche base d'annotations qui l'accompagnera pendant le processus de développement depuis la spécification fonctionnelle jusqu'à l'allocation, l'analyse et la validation.

La robotique mobile s'intéresse particulièrement à la conception des robots à roues. Globalement, La robotique nécessite un grand nombre de compétences dans différentes disciplines. Bien avant la réalisation mécanique d'un robot mobile et la conception de son architecture informatique embarquée (logiciel/matériel), une étape primordiale consiste à établir son modèle cinématique et à étudier sa commandabilité. Nous nous focaliserons sur les robots mobiles, non-holonomes et plats.

1 Ingénierie dirigée par les modèles

Toute personne ayant déjà tenté de lire et de comprendre du code afin de le réutiliser, de le modifier ou de le corriger en cas de dysfonctionnements, sait à quel point cette tâche est longue et difficile quel que soit le langage de programmation et malgré les indispensables lignes de commentaires. Selon Bran Selic [2], cela est principalement dû à la rupture sémantique qui existe entre les concepts spécifiques au domaine où évolue l'application et le vocabulaire du langage de programmation. Tel l'exemple des machines à états qui une fois transformées en code[1], perdent définitivement leur aspect graphique intuitif. La solution est donc de combler cette rupture sémantique par des abstractions qui nous permettront de spécifier notre programme dans un langage métier proche du domaine en question (DSL, Domain Specific Language). De plus, la spécification de l'application à un tel niveau d'abstraction nous évitera de la polluer par les détails et techniques d'implémentation.

L'Ingénierie Dirigée par les Modèles (IDM) vient répondre à ce besoin d'abstraction en plaçant les modèles au centre du développement logiciel. Via des modèles, on peut décrire dans notre langage métier plusieurs facettes de notre application loin des détails d'implémentation. L'IDM permet ainsi de répondre à la complexité croissante des systèmes, en améliorant l'expressivité, la portabilité, la lisibilité et la communication des intentions entre flots de conception, elle autorise également la séparation des préoccupations et la réutilisation. Forte de ces avantages, l'IDM recherche l'efficacité dans le processus de développement et se traduit par des gains en temps et en argent immédiats et appréciables.

Longtemps les langages de modélisation avaient une sémantique peu précise contenant beaucoup de points de variation. Cela freinait l'automatisation de la génération de code depuis les modèles qui restait manuelle et source d'erreurs. En effet, l'usage des modèles se justifiait dans les premières étapes du cycle de développement, car leur clarté facilite la conception et la communication entre flots, mais au moment de l'implémentaion, on traduisait manuellement ces modèles en code sans garder aucun lien formel. Le développeur se retrouvait dès lors avec deux représentations indépendantes de son application, l'une en modèle et l'autre en code. Lors des modifications du code, il abandonnait rapidement le maintien des modèles qu'il considérait comme une charge de travail supplémentaire. Par conséquent, on a longtemps cantonné les modèles au simple rôle de documentation, et en général, l'IDM avait du mal à conquérir un public.

Aujourd'hui les sémantiques des DSLs sont devenues non-ambigues, les modèles construits grâce à ces DSLs ont une représentation équivalente en XML [3], et les technologies autour de XML ont permis le développement d'une multitude d'outils de manipulation de modèles pour extraction de données, transformation ou génération de code. L'IDM est dorénavant gage d'efficacité dans le monde du logiciel. Pour plus d'informations, nous invitons le lecteur à lire les chapitres 1 et 2 du livre *«L'ingénierie dirigée par les modèles, au-delà du MDA»* [4], il y trouvera une excellente synthèse de l'IDM.

Dans la suite de cette section, nous allons introduire l'approche MDA qui est au centre de l'IDM et qu'on adoptera le long de notre travail de thèse. Nous expliquerons aussi la notion de métamodèle. Ensuite nous nous attarderons sur le langage unifié de modélisation UML, nous rappellerons sa structure, les moyens de l'étendre, les techniques de manipulation des

[1] On code souvent les machines à états sous forme de commandes conditionnelles imbriquées, la génération de code est automatisable.

modèles et l'outillage nécessaire pour ce faire.

1.1 Approche MDA

La variété et l'évolution rapide des plateformes d'exécution accentuent le besoin de réutilisation, de portabilité et d'interopérabilité des modèles. En novembre 2000, l'OMG (Object Managment Group) [5] introduit la séparation architecturale des préoccupations avec son initiative MDA (Model Driven Architecture) [6]. L'approche MDA conjugue, entre autres, trois modèles, le PIM (Platform Independent Model), le PDM (Plateform Description Model) et le PSM (Platform Specific Model). Le PIM décrit la structure de l'application indépendamment de la plateforme, il s'agit d'une spécification de l'application dans le langage métier qui s'abstrait des détails d'implémentation. Le PDM est quant à lui un modèle de description de la plateforme d'exécution cible ou du moins celui de son API (Application Programming Interface) et de son comportement. Enfin, le PSM est, comme son nom l'indique, un modèle d'application spécifique à une plateforme ou une technologie particulière.

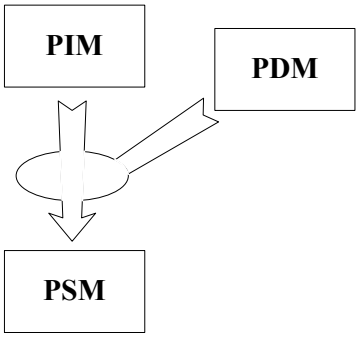


FIG. 2.1 – Approche MDA

Comme le montre la figure 2.1, l'approche MDA préconise que le PIM et le PDM doivent mener à la génération du PSM. Elle recommande ensuite que ce processus s'opère automatiquement par une succession de transformations et de raffinements du PIM avec des détails du PDM. Ainsi, un PIM peut être utilisé pour fournir différents PSMs pour des plateformes d'exécution différentes. En effet, il suffit de garder le même modèle PIM et de varier les PDMs pour générer différents PSMs.

L'approche MDA utilise massivement le concept de plateforme sans en donner une définition précise. Le long de ce document, on dénotera par plateforme d'exécution, un support complet de l'application, dans le sens où il offre toutes les ressources nécessaires à la réalisation des fonctionnalités de l'application [7]. Signalons que *application* et *plateforme* restent des rôles joués par un système dans un environnement donné (ex : un système d'exploitation joue le rôle d'une plateforme pour un programme utilisateur et le rôle d'une application vis-à-vis de la plateforme matérielle). Le MDA a également introduit la séparation des propriétés fonctionnelles de celles qu'on dit non-fonctionnelles. Si les premières sont annotées au niveau du PIM, les propriétés non-fonctionnelles sont relatives à la plateforme d'exécution et annotées uniquement au niveau du PSM (ex : la performance est une propriété non-fonctionnelle car elle dépend à la fois de l'application et de son support d'exécution).

Pour beaucoup, le MDA est à l'origine de l'IDM [4], cependant l'IDM se veut une vision plus générale qui intègre tout ce qui gravite autour des modèles et ne se résume point à une approche particulière. Comme l'IDM, le MDA a ses détracteurs qui jugent l'approche très idéaliste, et qui considèrent qu'il est pratiquement impossible de spécifier un PIM en ignorant complètement le PDM. D'autres moins catégoriques, insistent sur la nécessité d'avoir au minimum un modèle abstrait qui factorise les concepts des différentes plateformes cibles (PDMs) et qui doit être pris en compte lors de la réalisation du PIM, dans le but de générer ensuite les différents PSMs correspondant aux différents PDMs [8].

1.2 Les niveaux de modélisation

Un modèle d'un système est une vue abstraite de ce dernier qui en décrit un ou plusieurs aspects en occultant d'autres. On peut donc avoir plusieurs modèles d'un même système qui diffèrent selon les aspects considérés ou selon le niveau d'abstraction appliqué. Si tout modèle est compréhensible par l'homme qui l'a créé, il doit aussi être interprétable par la machine qui va l'exécuter. Il faut donc que le modèle soit conforme à un langage clairement défini pour qu'on puisse automatiser son interprétation. **Ce langage de modélisation est dit métamodèle**. Un métamodèle est à son tour un modèle conforme à un méta-métamodèle et ainsi de suite. Pour arrêter ce schéma infiniment récursif, il nous faut un méta-métamodèle qui s'auto-définit. MOF (Meta-Object Facility) [9] standardisé par l'OMG est un tel méta-métamodèle, il définit des concepts de base comme classe et relation qui peuvent tout définir y compris eux-mêmes. A ce stade, on ne peut s'empêcher d'établir une analogie avec les langages de programmation, où tout programme (modèle) est conforme à une grammaire (métamodèle) spécifiée en EBNF (méta-métamodèle) sachant que la grammaire du EBNF peut se décrire elle-même.

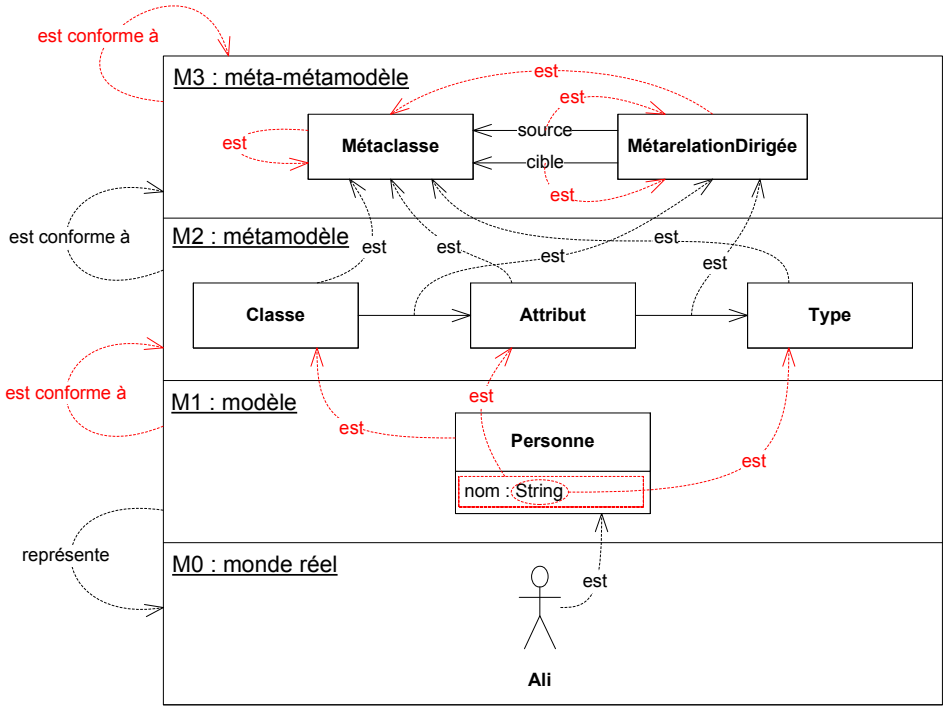


FIG. 2.2 – Les niveaux de modélisation

La figure 2.2 illustre la relation de conformité qui lie un modèle à son métamodèle. On y distingue quatre niveaux de modélisation, M0 correspond au monde réel, qu'on représente en M1 par un modèle contenant une classe *Personne* avec un attribut *nom* de type *String*. Ce modèle est donc conforme au métamodèle du niveau M2 qui à son tour est conforme au méta-métamodèle du niveau M3. Ce dernier définit le concept de *Métaclasse*, de *MétarelationDirigée* et les relations *source* et *cible* qui les lient. M3 a donc cette capacité de s'auto-définir puisque *Métaclasse* et *MétarelationDirigée* sont des *Métaclasse*(s) et les relations *source* et *cible* sont des *MétarelationDirigée*(s).

1.3 Le langage de modélisation UML

Un objet (une classe) est une entité constituée d'attributs représentant des données ainsi que d'opérations manipulant ces attributs. Bien que le paradigme objet fût introduit en 1967 grâce à Simula [10], il n'a connu son grand essor qu'à partir des années 80 avec l'introduction de mécanismes tels l'héritage et l'encapsulation(accessibilité), l'apparition de langages de programmation comme C++, et la multiplication de méthodologies de développement orienté objet. On peut dénombrer plusieurs dizaines de méthodologies publiées avant 1995, trois parmi elles ont rencontré plus de succès dans la communauté du logiciel, il s'agit de la méthode Booch [11] développée par Grady Booch, OMT [12] par James Rumbaugh et OOSE [13] par Ivar Jacobson. Ce grand nombre de méthodologies a considérablement freiné leur usage parmi les industriels. Ces derniers se perdaient dans la multitude d'approches et d'outils de développement objet. Face à cette situation, en 1994, Rational engagea Booch, Rumbaugh et Jacobson pour reprendre leurs trois méthodes de développement et les unir dans une seule et nouvelle méthode unifiée. Confrontés à l'impossibilité d'unifier toutes les méthodologies compte tenu de la diversité de leurs domaines d'application, ils décidèrent d'unifier au moins les différents langages de modélisation adoptés par ces méthodologies. En Juin 1996, la version 0.9 du langage de modélisation unifié UML (Unified Modeling Language) fût éditée pour être un support commun à toutes les méthodologies développées jusqu'ici. Rapidement, UML a été standardisé par l'OMG et son consortium international regroupant plusieurs organismes industriels et centres académiques. En constante évolution, UML2.1 est aujourd'hui le langage standard de la modélisation orientée objet, il s'agit du métamodèle le plus utilisé dans le monde. La figure 2.3 résume cet historique.

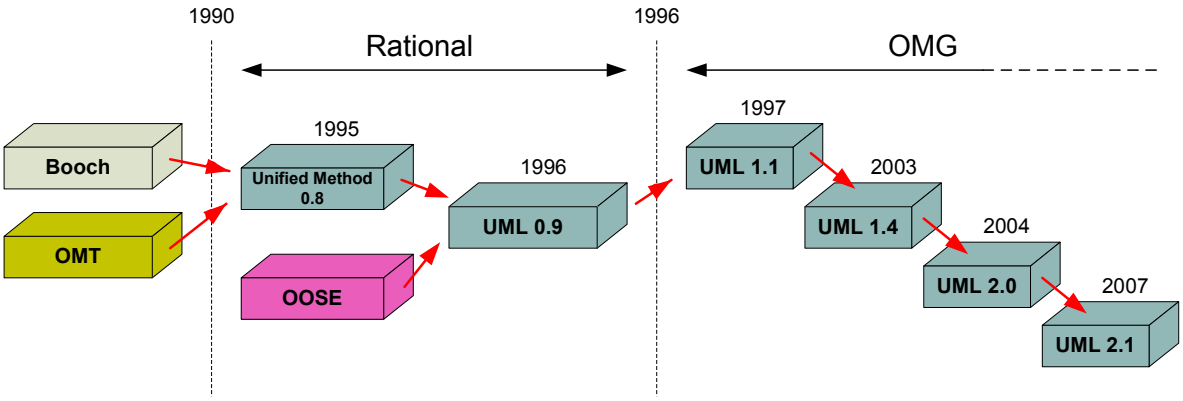


FIG. 2.3 – Historique de UML

UML est riche en notations, il définit treize diagrammes organisés en trois catégories. Six diagrammes de structure dont le diagramme de classe, le diagramme de déploiement et le diagramme de structure composite, les diagrammes de structure décrivent la structure statique du système. Trois diagrammes de comportement dont le diagramme d'activité et le diagramme de machine à états. Et finalement, quatre diagrammes d'interaction dont le diagramme de séquence et le diagramme de communication, les diagrammes d'interaction peuvent dériver des diagrammes de comportement. Le langage UML est défini par une norme en deux parties, l'*infrastructure* [14] spécifie le noyau UML commun à tous les diagrammes pendant que la *superstrusture* [15] est plus longue et spécifie le détail de chaque diagramme. La lecture et la compréhension de ce document de thèse nécessitent une fine connaissance du diagramme de classe et du diagramme de structure composite.

1.4 Mécanismes d'extension d'UML

UML est un langage de modélisation généraliste adapté à de nombreux domaines d'application. Il existe toutefois des situations et des domaines spécifiques auxquels ce langage n'est pas entièrement approprié. C'est le cas, par exemple, lorsque la syntaxe ou la sémantique des éléments UML ne peut pas exprimer certains concepts spécifiques au domaine, ou lorsque l'on veut limiter ou personnaliser certains des éléments UML dont la généricité ou les points de variation sont sources d'ambiguité. Consciente de cette déficience, l'OMG propose deux approches pour définir un DSL. La première est fondée sur la spécification d'un nouveau métamodèle, indépendamment d'UML, en utilisant le méta-métamodèle standardisé MOF. La deuxième solution est une extension d'UML, dans laquelle certains des éléments d'UML sont spécialisés et regroupés dans un profil, tout en laissant intact la structure et la sémantique du métamodèle UML. Chaque variante a ses avantages et ses inconvénients [16], et le choix d'une approche plutôt que l'autre se fait au cas par cas. Toutefois, des passerelles ont été réalisées pour passer d'un DSL à une extension UML et vice versa [17].

Nous préférons clairement dans le cadre de ce travail de thèse le mécanisme d'extension d'UML à la création, de toute pièce, d'un nouveau métamodèle. Cela nous évitera des redéfinitions inutiles de la structure, de la sémantique ou des notations du grand nombre de concepts que nous réutilisons et qui sont déjà exhaustivement définis dans UML. On pourra également utiliser de la sorte les outils UML pour le dessin des diagrammes, la transformation de modèles, la génération de code, la rétro-ingénierie, etc. Nous restons cependant conscients des limites des extensions d'UML via des profils, puisque celles-ci ne donnent pas accès au métamodèle UML. Il s'agit certainement d'une stratégie voulue par l'OMG pour protéger le standard UML de toute modification. Nous allons donc présenter deux alternatives à ce mécanisme d'extension d'UML, qui accèdent au métamodèle et qu'on nommera *lightweight extension* et *heavyweight extension*. Signalons, tout de même, que ces deux alternatives ne sont pas standardisées par l'OMG.

1.4.1 Profils UML

Ce mécanisme simple et efficace pour étendre UML, a permis de définir plusieurs profils adaptant UML à la grande majorité des domaines d'application. Plus d'une dizaine de

profils a même été standardisée par l'OMG. L'extension d'UML via profil, s'opère par trois dispositifs complémentaires :

➢ ***Stereotype*** : Un stéréotype étend une métaclasse existante du métamodèle UML pour en changer la sémantique. Un stéréotype peut posséder des propriétés (`tag definition`) et être sujet à des contraintes. UML permet également d'associer à chaque stéréotype sa propre notation sous forme d'une icône et/ou un cadre (`shape`) pour le distinguer graphiquement du concept qu'il étend.

➢ ***Tag definition*** : Les propriétés de stéréotype sont un moyen d'annoter des caractéristiques supplémentaires au concept UML étendu. Elles ont impérativement un type et ce type doit appartenir à UML, sinon, il doit être défini et annexé au profil.

➢ ***Constraint*** : On peut spécifier des règles OCL (Object Constraint Language) [18] qui viennent contraindre le métamodèle UML afin de le spécialiser.

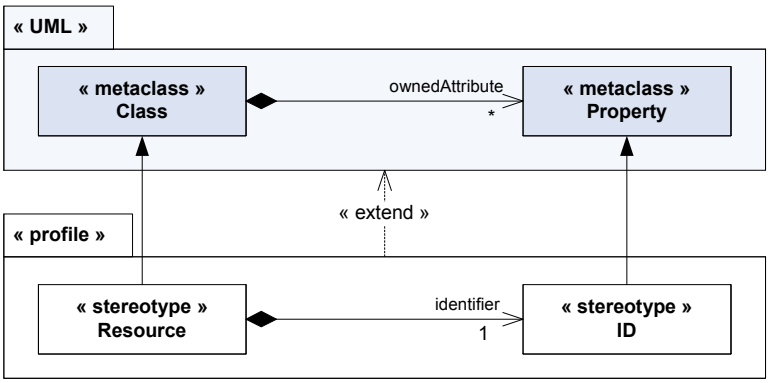


FIG. 2.4 – Exemple de profil UML

Comme exemple, prenons un domaine d'application basé sur le concept de ressource qui n'est autre qu'une classe UML avec une sémantique d'exécution particulière. Puis, imaginons que chaque ressource doit avoir un attribut qui joue le rôle d'identifiant. Pour commencer, il suffit d'étendre UML avec un profil formé de deux stéréotypes `Resource` et `ID` qui étendent respectivement les métaclasses `Class` et `Property` (voir figure 2.4). Maintenant, si on veut spécifier que chaque ressource doit avoir un attribut comme identifiant, seule une règle OCL (voir ci-dessous) nous le permettra. En effet, le mécanisme d'extension d'UML via profil habilite à étendre la sémantique des métaclasses UML par des stéréotypes, mais il ne prend point en charge les associations qui existent entre stéréotypes susceptibles de concerner et d'affecter les méta-associations entre les métaclasses étendues.

```
context Resource
 inv: self.base_class.ownedAttribute->exists(p:uml::Property |
        p.getAppliedStereotypes()->exists(name = 'ID'))
```

1.4.2 Import du métamodèle UML

Le mécanisme d'extension d'UML qu'on dit *lightweight extension*, se base sur un import (en lecture seule) du métamodèle UML ou du moins d'une partie du métamodèle UML.

Pour ce faire, on se place au niveau métamodèle (M2) et on utilise l'import du MOF (M3). Ce mécanisme d'extension s'opère grâce à trois dispositifs :

➢ ***Spécialisation des métaclasses*** : Par de simples héritages (au niveau M2), on peut étendre des métaclasses importées d'UML.

➢ ***Spécialisation des méta-associations*** : Grâce au dispositif `subsets` du MOF, on peut spécialiser des méta-associations d'UML.

➢ ***Surcharge des méta-associations*** : Grâce au dispositif `redefines` du MOF on peut redéfinir des méta-associations d'UML.

Reprenons l'exemple précédent en adoptant la *lightweight extension*. Cette fois, on importe le métamodèle UML, les métaclasses du domaine `Resource` et `ID` héritent respectivement des métaclasses d'UML `Class` et `Property`, puis on spécialise le rôle `ownedAttribute` de la méta-association d'UML, qui lie `Class` à `Property`, par celui de la méta-association du domaine. On n'a plus besoin de la règle OCL.

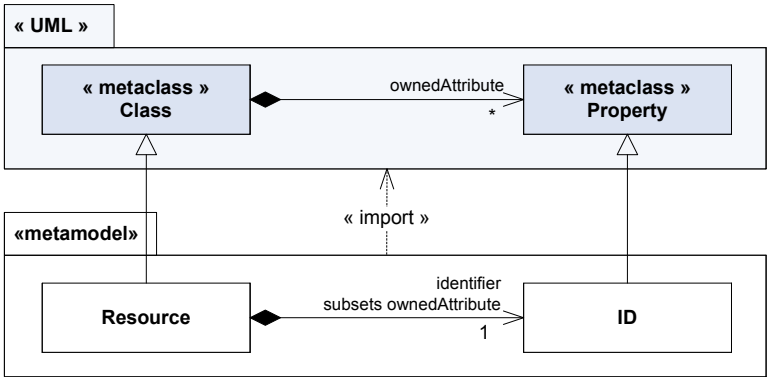


FIG. 2.5 – Exemple de *lightweight* extension

1.4.3 Modification du métamodèle UML

Le mécanisme d'extension qu'on dit *heavyweight extension* est catégorique, il se base sur une copie du métamodèle d'UML ou du moins d'une partie. Pour cela, on se place au niveau métamodèle (M2) et on utilise le dispositif « `merge` » du MOF (M3). On peut ensuite opérer toutes les modifications nécessaires directement sur notre copie de la métaclasse `Class`. Dans l'exemple précédent, si on considère qu'en général toute classe a un attribut identifiant, il nous suffit de modifier directement notre copie du métamodèle UML pour obtenir le métamodèle de la figure 2.6.

A noter que cette classification des mécanismes d'extension d'UML est similaire aux travaux de James Bruck and Kenn Hussey (IBM) dans [19], et que seuls le mécanisme d'extension via profil et le mécanisme *lightweight extension* seront utilisés dans la suite.

1.5 Manipulation de modèles

Une fois le modèle du système défini et conforme au métamodèle du domaine, on aimerait entamer diverses manipulations de ce modèle afin de l'analyser, le simuler et l'implémenter. Mis à part le parcours du modèle pour extraction de données, deux

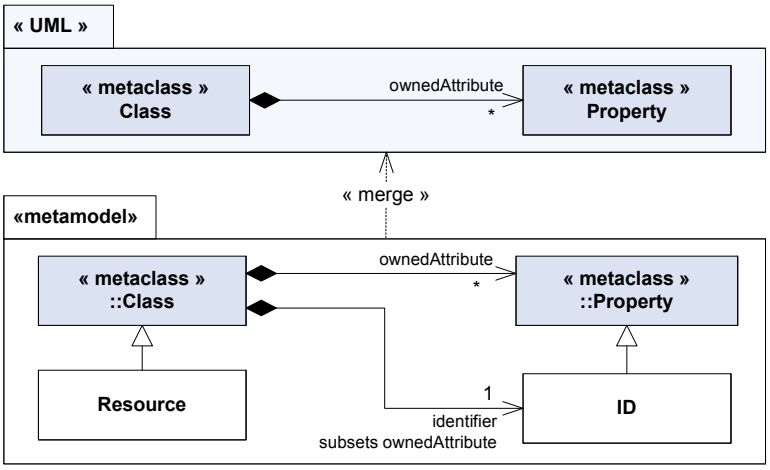


FIG. 2.6 – Exemple de *heavyweight* extension

manipulations majeures s'offrent à nous : la transformation de modèles et la génération de code.

1.5.1 Transformation de modèles

La transformation de modèles est un processus qui convertit un ensemble de modèles d'un système donné à un autre ensemble de modèles du même système. La figure 2.7 illustre ce processus. Si les modèles en entrée et en sortie de la transformation sont conformes au même métamodèle, on dit que la transformation est endomorphe ou endogène. Dans le cas contraire, quand le métamodèle source et le métamodèle cible sont différents, on dit que la transformation est exomorphe ou exogène.

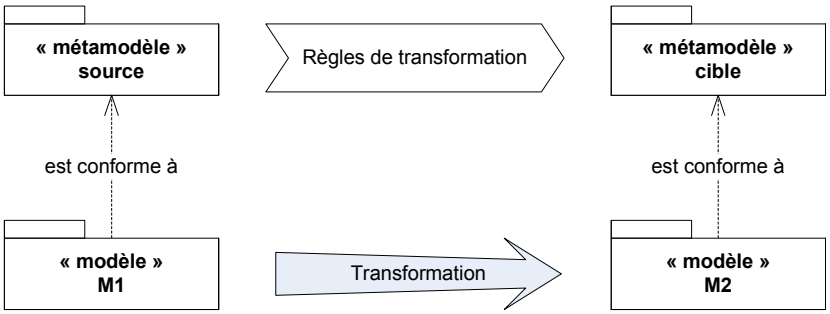


FIG. 2.7 – Transformation de modèles

Une transformation de modèles est spécifiée par un ensemble de règles de transformation. Même si ces règles de transformation sont appliquées au niveau modèle (M1), elles sont cependant décrites au niveau métamodèle (M2) et nécessitent un environnement et un langage permettant le parcours de métamodèles. Pour répondre à la multiplication de tels environnements [20, 21], l'OMG a récemment standardisé un langage de transformation de modèles : Query / Views / Transformations (QVT) [22].

1.5.2 Génération de code

La transformation de modèles garantit l'interopérabilité entre langages de modélisation, elle est souvent utilisée afin de générer des vues du système qui servent d'entrées à des analyses particulières. Cependant, tant que les plateformes d'exécution ne prennent pas directement en entrée des modèles qu'elles interprètent puis exécutent, il va falloir transformer le modèle du système, non pas en un nouveau modèle, mais en un programme du langage machine. Cette transformation d'un langage de modélisation vers un langage machine est dite la génération de code.

La génération de code se fait en deux étapes. La première étape consiste à traduire le modèle du système en un langage de programmation haut niveau (C, java, C++, SystemC). La seconde étape consiste à compiler ce programme pour produire un binaire exécutable par la machine. Grâce à cette séparation en deux étapes, la portabilité des modèles vers une plateforme particulière dépend uniquement de l'existence d'un compilateur du langage de programmation pour la plateforme cible. Aussi, les évolutions du langage de modélisation se répercutent uniquement sur les règles de transformation des modèles en langage de programmation, l'étape de compilation reste inchangée.

1.6 Outillage

Depuis sa standardisation par l'OMG en 1996, le langage unifié de modélisation UML est largement adopté par la communauté du logiciel. Plusieurs outils de modélisation en UML et de manipulation de modèles UML ont vu le jour. Cependant chacun de ces outils avait sa propre représentation des modèles UML, c-à-d son propre format de stockage des modèles, cela freinait l'interopérabilité entre outils et rendait la manipulation des modèles dépendante de l'outil de modélisation utilisé et de son format de stockage. L'OMG a donc intervenu une nouvelle fois pour standardiser un format de stockage particulier : XMI (XML Metadata Interchange) [23]. Il s'agit d'un schéma XML adapté à la représentation des modèles qui sont conformes à des métamodèles spécifiés en MOF, les modèles UML en font partie. Il est dorénavant possible d'échanger des modèles entre différents outils de modélisation supportant le schéma XMI. De plus, les outils de manipulation de modèles sont devenus indépendants des outils de modélisation. Une troisième conséquence de la standardisation du XMI a été de rassembler les efforts autour d'outils libres de modélisation et de manipulation de modèles. Dans le cadre de ce travail de thèse, nous utiliserons Papyrus [24] comme outil de modélisation et Acceleo [25] comme outil de génération de code, ces deux outils sont libres et à code ouvert (Open Source).

1.6.1 Papyrus

Développé au sein de notre laboratoire CEA LIST [26], Papyrus est un outil d'édition graphique de modèles UML2.1, rigoureusement conforme à la norme UML. D'un côté, il est conforme au format de sauvegarde (XMI) et de l'autre il est conforme au standard graphique DI (Diagram Interchange) [27]. Décrit par l'OMG, DI complète le format XMI afin de pouvoir échanger les données graphiques. Papyrus est basé sur l'environnement Eclipse EMF [28]. Il est ainsi facilement extensible et il fusionne parfaitement avec d'autres *plugins* Eclipse autour

de l'IDM, tels le *plugin* OCL [29] pour la spécification des contraintes sur les modèles, ATL [20] pour la transformation de modèles, et JET [30] et Acceleo pour la génération de code.

Papyrus permet l'édition de la plupart des diagrammes UML2.1, on utilisera particulièrement le diagramme de classe et le diagramme de composite structure. Il permet aussi la définition de profils, on peut notamment définir des types complexes pour les `tag definition`(s) et associer des icônes aux stéréotypes. Ensuite, Papyrus nous habilite à appliquer à la volée ces profils sur nos modèles, on peut effectivement modifier le profil même quand il est déjà appliqué sur le modèle du système en cours de développement, Papyrus supporte la réapplication de profils.

1.6.2 Acceleo

Acceleo permet d'exploiter les données contenues dans un modèle pour générer le code de l'application. Acceleo est nativement intégré à Eclipse et supporte les standards de modélisation UML, EMF, XMI, MOF. Cet outil encourage la mise en oeuvre de l'approche MDA en permettant le passage de modèles de haut niveau vers différentes cibles technologiques. En effet, il suffit de définir un module Acceleo pour chaque plateforme cible.

La syntaxe mise au point par Acceleo est intuitive, extensible et dédiée à la génération de code. Un modèle en entrée d'Acceleo est considéré comme un arbre de données qu'on parcourt efficacement avec l'éditeur Acceleo, puis on spécifie le code à générer grâce éventuellement à des éléments de contrôle simples tels que des conditions et des boucles.

2 Développement de systèmes embarqués temps-réel

Par souci de coût, de poids et d'encombrement, on désigne par système embarqué tout système hybride (logiciel/matériel) dont la composante matérielle est contrainte à des ressources limitées en termes de puissance de calcul, de mémoire et de consommation d'énergie. Nous ajoutons à cela des contraintes de temps puisque nous nous intéressons particulièrement aux systèmes embarqués temps-réel. Ces contraintes de temps sont souvent critiques car tout dépassement des délais d'exécution peut avoir des conséquences catastrophiques (ex : perte de vies humaines dans l'avionique).

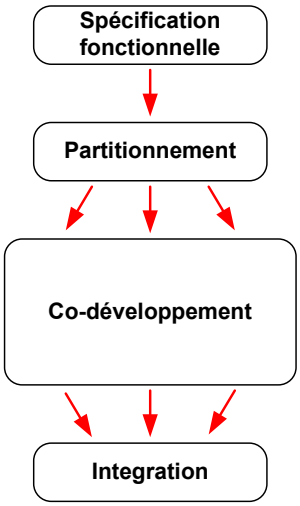


FIG. 2.8 – Processus de conception des systèmes hybrides

La figure 2.8 représente le processus de conception des systèmes hybrides, on y distingue quatre étapes majeures, la spécification fonctionnelle, le partitionnement, le co-développement et l'intégration. L'étape de co-développement est une étape centrale, elle se compose de plusieurs flots de conception qui évoluent parallèlement et simultanément tout en gardant une forte communication entre eux. Plus il y a de flots de conception, plus le temps de développement est réduit et plus le besoin de synchronisation entre flots est important. Dans la suite, nous expliquerons chacune des quatre étapes et nous détaillerons particulièrement l'étape de co-développement qui concentre nos contributions.

2.1 Spécification fonctionnelle

La spécification fonctionnelle est la première étape du processus de développement, elle prend en entrée le cahier des charges du système, pour en extraire un découpage structurel séparant les fonctionnalités du système. La spécification fonctionnelle est considérée comme discipline de l'ingénierie système [31]. Le standard SysML [32] qui applique l'IDM à l'ingénierie système, définit deux diagrammes de blocs (Block Definition Diagram & Internal Block Diagram) parfaitement adaptés à la spécification fonctionnelle. Les diagrammes de blocs de SysML sont des versions modifiées des diagrammes de structure d'UML. En effet, le concept de bloc dans SysML et le concept de composant dans UML ont une sémantique similaire, il s'agit d'une entité fonctionnelle (logicielle ou matérielle) qui communique avec les autres entités via des ports et des interfaces.

Dans sa phase préliminaire, la méthodologie Accord/UML [33] aide le développeur à mettre en oeuvre un dictionnaire depuis le cahier des charges, et à établir ensuite un diagramme des cas d'utilisation qui identifie grossièrement les fonctionnalités du système. Par des raffinements successifs, le développeur devra aboutir à un modèle du système qui identifie, un à un, tous les composants (blocs).

2.2 Partitionnement

L'étape de partitionnement consiste à figer un choix d'implémentation pour chaque composant identifié lors de l'étape de spécification fonctionnelle précédente. Comme illustré en figure 2.9, ce choix s'opère parmi les trois flots de conception de l'étape co-développement :

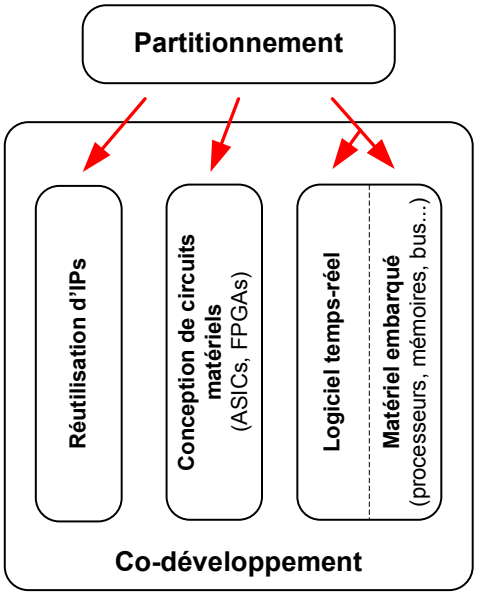


FIG. 2.9 – Flots de conception

Réutilisation d'IPs : propre aux systèmes sur puce, il s'agit de réutiliser des composants (IP, Intellectual Property) déjà conçus, réalisés et validés pour gagner du temps et minimiser les coûts de conception.

Conception de circuits matériels : depuis la modélisation du circuit jusqu'au routage, ce flot est souvent long et coûteux. Il réunit volontairement les circuits imprimés et les circuits configurables, car le processus de conception est pratiquement le même.

Conception de logiciels : Il s'agit d'un flot composite, il réunit deux flots fortement couplés, d'un côté le logiciel embarqué et de l'autre sa plateforme matérielle d'exécution (*computer-based architecture*).

Le rôle du partitionnement est de trouver le meilleur compromis entre ces différents flots de conception tout en minimisant les coûts et en maximisant les performances. Ce problème d'optimisation n'a jamais été mathématiquement résolu, car il dépend de plusieurs paramètres difficilement modélisables. Toutefois, beaucoup de méthodes ont été proposées [34, 35]. De manière générale, les architectes système ramènent ce problème à un

compromis flexibilité/performance en constatant qu'un matériel dédié (ASIC, Application Specialized Integrated Circuit) est performant mais non flexible, et qu'un processeur est flexible mais peu performant. Comme le montre la figure 2.10, des solutions intermédiaires existent aussi, il s'agit des circuits configurables (ex : FPGA) et des processeurs dédiés (ex : DSP). Enfin, les architectes système prennent également en compte le critère du temps de développement qui est en faveur du logiciel et de la réutilisation d'IPs.

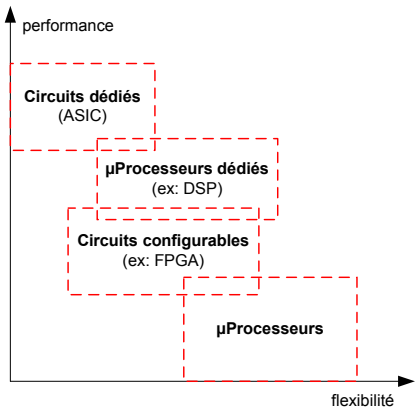


FIG. 2.10 – Rapport performance/flexibilité des composants matériels

La réutilisation d'IPs reste limitée par le manque de souplesse des composants, et les stratégies de confidentialité entre entreprises qui limitent les échanges d'IPs. Aussi, la conception de circuits matériels est très coûteuse en temps dans un contexte où le *time-to-market* est de plus en plus court. Par conséquent, nous ne considérons dans le cadre de ces travaux que le troisième flot de conception, c-à-d que nous implémentons les fonctionnalités du système exclusivement en logiciel. Nous obtenons ainsi des systèmes hybrides dont la composante matérielle est une plateforme d'exécution du logiciel, elle est typiquement composée de processeurs, de mémoires, de périphériques… Le processus de développement qu'on a décrit précédemment dans la figure 2.8, se retrouve simplifié en un processus losange illustré dans la figure 2.11 suivante.

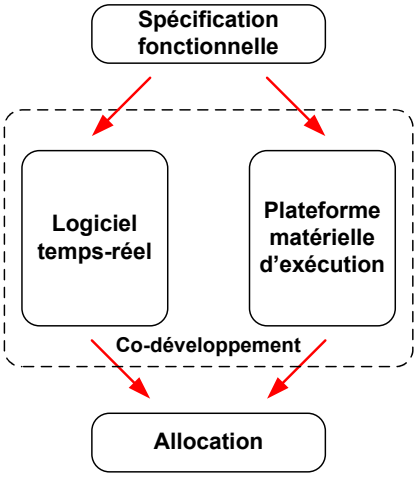


FIG. 2.11 – Processus de conception simplifié

Dans cette forme simplifiée du processus de conception, l'étape de partitionnement devient obsolète, puisqu'on implémente toutes les fonctionnalités du système en logiciel. L'étape de co-développement se limite à deux flots parallèles, l'un pour la conception du logiciel et l'autre pour la réalisation du matériel embarqué qui lui sert de plateforme d'exécution. L'étape d'intégration devient une étape d'allocation, car si l'intégration signifie l'interfaçage entre IPs, circuits matériels, logiciel embarqué... provenant des différents flots (voir figure 2.9), l'allocation est quant à elle plus spécifique et consiste à associer à chaque composant logiciel un noeud matériel pour supporter son exécution. Dans la section 2.3 suivante, nous établirons l'état de l'art du développement du matériel embarqué de la modélisation jusqu'à la simulation. Dans la section 2.4 nous établirons l'état de l'art de la conception d'applications temps-réel. Finalement, dans la section 3.2 nous discuterons de l'allocation.

2.3 Développement du matériel embarqué

L'usage de modèles haut niveau pour la spécification de systèmes complexes est une tradition dans l'ingénierie. La modélisation permet d'abstraire les aspects les plus importants pour les communiquer, les analyser et les valider avant toute implémentation. Pour le développement du logiciel comme pour le développement du matériel, le recours à la modélisation est une pratique courante. UML a mis le paradigme objet et l'approche composant au service de la modélisation, cela a révolutionné le développement logiciel grâce notamment à des mécanismes comme la généralisation, la composition, l'encapsulation, le raffinement, etc. La question qui se pose maintenant, est ce que le paradigme objet et l'approche composant sont adaptés à la conception du matériel ? surtout que la conception du matériel est de plus en plus un exercice de programmation. Dans ce cas là, il nous suffira donc d'étendre UML pour prendre en charge les concepts du matériel.

Le paradigme objet ainsi que l'approche composant sont parfaitement adaptés au développement matériel, ils se prêtent même mieux au matériel qu'au logiciel pour lequel ils ont été conçus. En effet, une ressource matérielle peut être intuitivement représentée comme une classe ayant des propriétés (caractéristiques) et des opérations (services), comme elle peut être considérée comme un composant avec des ports échangeant des services via des interfaces. Les mécanismes offerts par ces deux approches sont également bien adaptés à la conception du matériel [36]. Par exemple, la généralisation sert efficacement à la classification des ressources matérielles. En effet, le matériel est extraordinairement varié mais facilement classifiable, on distingue par exemple parmi les processeurs plusieurs architectures, parmi ces architectures plusieurs générations et ainsi de suite.

UML est par conséquent un langage de modélisation qu'on pourra étendre au matériel.

2.3.1 Comparatif des langages de modélisaton du matériel

Standardisé par l'OMG, le mécanisme d'extension d'UML par des profils a encouragé l'adoption d'UML comme langage de modélisation du matériel. Plusieurs profils ont vu ainsi le jour et bon nombre parmi eux, a même été standardisé. On distingue clairement deux grandes catégories. Une première catégorie regroupant des profils orientés implémentation qui s'intéressent plus aux circuits et à la réalisation des composants. Ces profils ont pour but

de générer du code et utilisent de ce fait UML comme HDL (Hardware Design Language) c-à-d comme langage d'implémentation du matériel. UML for SoC [37] et UML for SystemC [38] sont de tels profils, comme le montre la table ci-dessous, ils donnent tous les deux une transcription un à un des concepts de SystemC [39] en UML et garantissent ainsi la génération automatique du code SystemC ou du moins du squelette du code. Ces profils sont par conséquent peu abstraits, trop proches de l'implémentation et ont pour effet d'étendre UML avec les détails et la sémantique du HDL généré (en l'occurrence SystemC).

Profils UML	*Principaux concepts*
UML for SoC	SoCModule, SoCChannel, SoCConnector, SoCPort...
UML for SystemC	sc_module, sc_channel, sc_port...

La seconde catégorie est celle regroupant des profils qui modélisent le matériel du point de vue fonctionnel. Parallèlement à l'implémentation, il est nécessaire de spécifier des modèles fonctionnels, abstraits et compréhensibles du matériel afin de communiquer les intentions de conception, influencer le développement du logiciel, réaliser différentes analyses et procéder à l'allocation. Par exemple, une analyse d'ordonnancement requiert une vue du matériel qui met en valeur l'architecture en termes de nombre de processeurs, de taille mémoire, etc. Plusieurs profils UML permettent une telle description, on peut en citer SPT (Schedulability, Performance and Time) [40], AADL (Avionics Architecture Description Language) [41], AUTOSAR (AUTomotive Open System ARchitecture) [42], GASPARD (Graphical Array Specification for Parallel and Distributed Computing) [43] et ACOTRIS (Analyse et Conception à Objets Temps-Réel pour Implantation asynchrone/Synchrone) [44]. Comme le montre la table ci-dessous, tous permettent d'annoter la fonctionnalité d'un composant matériel selon qu'il soit une ressource de calcul, de stockage, de communication, etc. Cependant le niveau de détail qu'ils procurent reste insuffisant pour permettre des analyses précises ou la simulation du matériel. Par exemple, si on veut calculer le WCET (Worst Case Execution Time), il nous faut prendre en compte la micro-architecture du processeur, chose qu'aucun profil précédemment cité ne peut décrire. Néanmoins, ces profils pourront servir de base à des métamodèles plus détaillés.

Profils UML	*Principaux concepts*
SPT	Device, Processor, CommunicationResource.
AADL	Processor, Memory, Bus, Device.
AUTOSAR	HWElement, ProcessingUnit, SensorActuator, HWPort, DigitalIO...
Gaspard	Processor, Communication, Memory, Asic, Fpga, Ram, Bus, Cache...
ACOTRIS	ECU, ECUGate, Channel.

En dehors de la sphère UML, MILAN (Model-based Integarted simuLAtioN) [45] est un atelier qui définit un métamodèle pour la modélisation précise et la simulation du matériel. On reproche à MILAN d'être fortement dépendant des outils d'analyse et de simulation qu'il cible. En effet, le métamodèle de MILAN est une simple représentation de la grammaire de SimpleScalar [46] qui est un outil de simulation largement utilisé par la communauté du matériel. Dans le cadre de ce travail de thèse, on va proposer un profil UML de modélisation des aspects fonctionnels du matériel, on se basera sur l'existant et en couvrera plusieurs niveaux de détail afin d'habiliter toute sorte d'analyse et de simulation. On va également

associer notre langage avec les profils d'implémentation, le développement matériel sera ainsi entièrement basé sur les modèles.

2.3.2 Techniques de simulation du matériel

Dès les premières étapes du cycle de développement, la simulation du matériel est un formidable moyen de test et de debuggage du logiciel puisqu'elle dispense de la disponibilité physique du matériel. Les avancées techniques dans la simulation sont en train de secouer tout le processus de développement des systèmes hétérogènes. Grâce à la simulation, on peut explorer plusieurs architectures matérielles, y compris de nouvelles configurations sans attendre leur implémentation. On peut même tester la tolérance du logiciel en simulant des pannes matérielles. Si par le passé, il n'existait que des simulateurs de jeu d'instructions (ISS) qui ne simulaient qu'un processeur exécutant un code assembleur, aujourd'hui, bon nombre d'outils permettent de simuler une plateforme matérielle entière (processeurs, mémoires, bus, périphériques...) démarrant un système d'exploitation.

On distingue clairement deux types de simulation : la simulation fonctionnelle et la simulation de performance. La simulation fonctionnelle qu'on dit aussi transactionnelle se limite à la simulation des fonctionnalités du matériel sans prendre en compte les propriétés non-fonctionnelles comme les temps d'exécution ou l'usage mémoire. Elle est plus facile à réaliser et plus rapide à exécuter, puisqu'elle ne prend pas en charge la microarchitecture des processeurs, les bandes passantes des bus, les timings mémoire, etc. La simulation de performance implémente quant à elle toutes les propriétés du matériel, elle est par conséquent plus lente mais plus précise. La simulation des systèmes temps-réel exige la simulation de performance puisque les contraintes de temps font partie intégrante des fonctionnalités.

On trouve dans [47] une étude et un comparatif des outils de simulation existants. Dans le cadre de ce travail de thèse, nous avons opté pour l'outil Simics [48] de Virtutech. Simics offre la plus riche librairie de composants matériels, il permet d'exécuter l'application, le système d'exploitation et les pilotes de manière identique à la plateforme matérielle réelle. Simics couvre également les deux types de simulation. Notons que Simics est libre pour le monde académique.

Simics

L'outil de simulation Simics est capable de simuler entièrement des systèmes embarqués distribués. L'approche utilisée est la simulation complète du système où le processeur, la mémoire, les périphériques et l'environnement du système sont simulés à un tel niveau de détail que le logiciel ne peut voir la différence avec la plateforme réelle. En effet, aucune modification du binaire n'est requise. Simics est capable de simuler un grand nombre de composants du matériel dont les processeurs PowerPC, ARM, SPARC, x86, les mémoires SRAM, DDR, Flash, les bus I2C, PCI, les ports série, les Timers, etc. En plus, la simulation est très rapide, il arrive même qu'elle soit plus rapide que la réalité quand les ressources simulées sont moins puissantes que la machine hôte. Dans le même registre, on peut aussi simuler une plateforme 64-bit sur une machine hôte 32-bit. Simics est riche en dispositifs [49] :

- La simulation est déterministe, le comportement du matériel simulé est toujours le même.
- On dispose de points d'arrêt pour suspendre la simulation à tout moment, sauvegarder l'état courant du système (logiciel et matériel) et reprendre la simulation plus tard.
- Un langage script nous permet de paramétrer le matériel avant le lancement, et d'accéder à l'état du système pendant l'exécution (ex : on peut brancher/débrancher un câble Ethernet à chaud).
- Simics offre un système de trace efficace qui permet de filtrer les informations requises.
- Simics permet notamment de créer de nouveaux composants qui ne font pas partie de sa librairie grâce au langage de modélisation DML (Device Modeling Language). On trouve dans [50] une comparaison entre DML et SystemC.

Nous avons réussi à porter facilement, dans Simics, l'infrasructure ACCORD que nous présenterons dans la section suivante.

2.4 Conception d'application temps réel

L'objectif de cette nouvelle section est de présenter l'atelier ACCORD (Atelier de Conception par Composants et Objets temps-Réel Distribués) [33, 51], qu'on adoptera pour la conception d'applications temps-réel distribuées. Cet atelier s'appuie largement sur les principes du MDA. Il est complètement basé sur UML et vise le développement d'applications temps-réel par des non-experts de ce domaine. En effet, l'objectif général de ACCORD est de masquer autant que possible les aspects d'implémentation autour d'une approche dirigée par les modèles (patrons de conception, raffinement automatique de modèle, génération de code, validation par les modèles…), afin de permettre aux développeurs de se concentrer sur les aspects métiers de ces systèmes (fonctionnalités, contraintes de performance…). L'atelier ACCORD s'inscrit dans le cadre d'une architecture logicielle à quatre couches illustrée en figure 2.12.

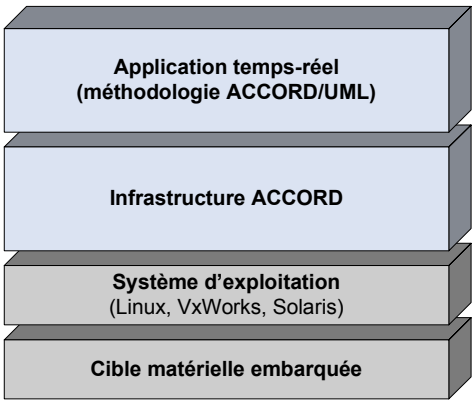


FIG. 2.12 – Architecture de l'atelier ACCORD

Il s'agit de trois couches logicielles et d'une couche matérielle. Directement au dessus de la cible matérielle, on trouve une couche système d'exploitation comme, par exemple, Linux ou VxWorks [52]. Ensuite on trouve l'infrastructure ACCORD qui implémente les mécanismes nécessaires au développement d'une application temps-réel multi-tâches. Elle fixe les politiques d'ordonnancement, de communication et de synchronisation. La dernière

couche de cette architecture est la couche application. Elle est entièrement basée sur des modèles UML réalisés grâce à la méthodologie de modélisation ACCORD/UML. Pour conclure, l'atelier ACCORD offre principalement une méthodologie ACCORD/UML pour la modélisation de l'application temps-réel puis la génération du code, et une infrastructure ACCORD pour le portage et l'exécution de cette application sur un système d'exploitation et une cible matérielle.

2.4.1 La méthodologie ACCORD/UML

Comme son nom l'indique, ACCORD/UML [33, 53] est une méthodologie basée sur UML, elle permet le développement d'applications temps-réel distribuées dans le langage métier du domaine temps-réel. Elle définit pour cela un ensemble de mécanismes abstraits permettant une expression des aspects qualitatifs (concurrence, comportement, communication…) et quantitatifs (contraintes temps-réel…) des applications à modéliser. ACCORD/UML a pour objectif d'adapter UML au temps-réel, elle décrit alors formellement des extensions à UML, ainsi que l'ensemble des règles de modélisation définissant les transformations de modèles, les choix pour les points de variation et autres points ambigus de la sémantique UML. Le schéma de la figure 2.13 présente une vue globale du processus de développement mis en oeuvre dans la méthodologie ACCORD/UML.

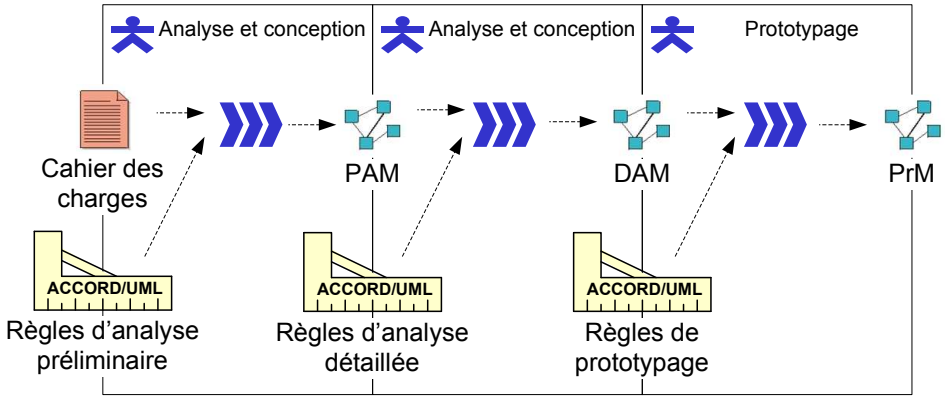


FIG. 2.13 – Vue globale du processus de développement ACCORD/UML

Le processus de développement ACCORD/UML s'articule autour de trois phases de raffinement de modèles. La première phase d'analyse permet d'obtenir le modèle de haut niveau PAM (Preliminary Analysis Model). Ensuite, la phase d'analyse détaillée construit le DAM (Detailed Analysis Model). Et finalement la phase de prototypage produit le PrM (Prototype Model) qui représente un modèle complet pour lancer automatiquement la génération de code fournie par l'atelier ACCORD. La transformation du PAM au DAM et la transformation du DAM au PrM sont automatiquement initiées par l'atelier ACCORD. Il reste ensuite à l'utilisateur d'enrichir les modèles ainsi obtenus selon les règles de conception énumérées par la méthodologie ACCORD/UML. Du point de vue MDA, le PAM est indépendant de toute plateforme d'exécution alors que le modèle PrM est clairement spécifique à une plateforme d'exécution particulière.

La méthodologie ACCORD/UML propose essentiellement deux concepts dédiés au temps-réel : *RealTimeObject* et *RealTimeFeature*.

RealTimeObject

ACCORD/UML introduit le concept d'objet temps-réel (RTO) comme une extension temps réel du paradigme d'objet actif. En effet, un objet temps-réel peut être considéré comme une entité autonome qui gère ses propres ressources de calcul et qui assure d'elle-même la réception des messages. Dans ce modèle, toute tâche est attachée à un certain objet temps-réel et correspond au traitement d'un message reçu. Une application multi-tâches est alors définie comme une interaction entre plusieurs objets temps-réel.

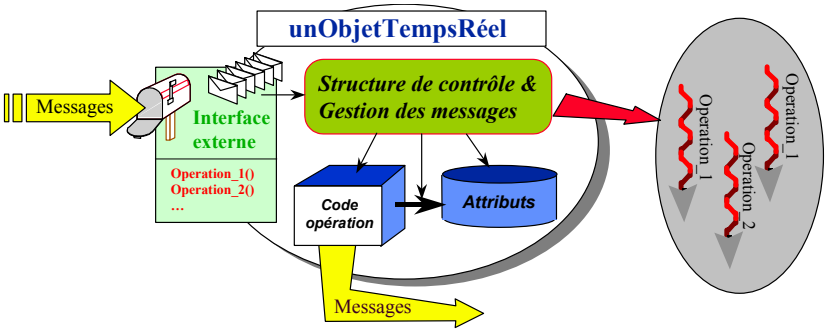


FIG. 2.14 – Structure d'un objet temps-réel

On voit dans la figure 2.14 que la structure d'un objet temps-réel est principalement composée de quatre éléments : un ensemble d'attributs, une série de méthodes, un contrôleur local et une boîte aux lettres. Un objet temps-réel reçoit et enregistre les messages dans sa boîte aux lettres. Le contrôleur local est en charge de la gestion de la boîte aux lettres, c'est-à-dire le lancement, la suspension ou le rejet du traitement de chaque message reçu, ainsi que la gestion de la concurrence. Notons que la boîte aux lettres contient à tout instant, tous les messages en attente d'exécution.

RealTimeFeature

Pour spécifier les contraintes de temps dans un modèle, la méthodologie ACCORD/UML introduit le dispositif temps-réel RTF. Ce dispositif est renseigné au niveau de l'action générant l'émission d'un message, il sera ensuite vérifié lors du traitement de l'opération correspondante par l'objet récepteur du message. Différentes valeurs peuvent être spécifiées au niveau d'une RTF permettant ainsi de modéliser des caractéristiques qualitatives temps-réel telles les échéances, périodes, temps-début…

La figure 2.15 suivante représente le dispositif RTF en étendant UML par le mécanisme *heavyweight extension*.

2.4.2 L'infrastructure ACCORD

L'infrastructure ACCORD [54, 51] est une couche logicielle (intergiciel) qui s'intercale entre l'application ACCORD/UML et le système d'exploitation. Elle a de ce fait deux rôles, d'un côté elle implémente le modèle d'exécution ACCORD et de l'autre elle abstrait l'application temps-réel ACCORD/UML de toute dépendance vers un système d'exploitation particulier. Le modèle d'exécution ACCORD respecte les contraintes de

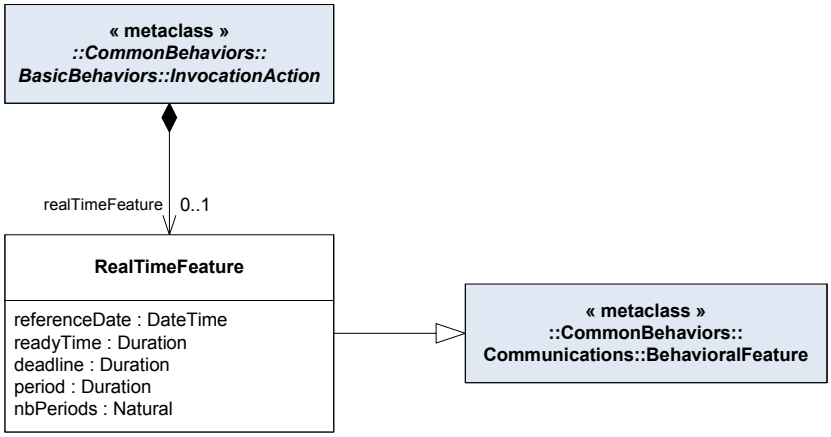


FIG. 2.15 – Le dispositif temps-réel RTF

concurrence (nReaders1Writer) et propose plusieurs mécanismes de synchronisation (synchrone, asynchrone, synchrone retardé) [55]. L'infrastructure ACCORD se doit donc de les implémenter.

L'infrastructure ACCORD est par ailleurs basée sur trois principes :

- Une requête est un appel de service entre objets temps-réel qui s'opère uniquement par le biais d'envoi de message.
- Chaque requête est localement traitée par l'objet temps-réel appelé qui vérifie les contraintes de concurrence et les contraintes de temps avec les autres requêtes en cours d'exécution ou en attente.
- Quand un objet temps-réel choisit une requête à exécuter, il l'insère dans un plan d'ordonnancement global partagé par tous les objets temps-réel de l'application. Cet ordonnancement global est EDF (Earliest Deadline First) mais RTC (Run To Completion), il lance l'exécution locale de la requête la plus urgente mais pas avant que le traitement de la requête (écrivain) précédente soit achevé.

Au lancement de l'application ACCORD, une pile de threads est créée statiquement, les contraintes de prédictibilité obligeant à limiter le nombre maximum de threads dans le système. Chaque réception de message est alors prise en charge par deux threads de cette pile : un thread de contrôle et un thread d'exécution. Le premier gère les contraintes et l'ordonnancement de la requête et le second reste bloqué jusqu'au lancement du traitement. A tout instant, il y a autant de threads de contrôle et autant de threads d'exécution que de requêtes (en cours d'exécution ou en attente) dans les boites aux lettres de tous les objets temps-réel de l'application. Cependant, seuls sont actifs les threads d'exécution des requêtes les plus urgentes et qui n'interfèrent pas entre elles. En général, l'infrastructure ACCORD protège efficacement des accès multiples en écriture par un ensemble de sémaphores et de zones critiques.

3 Le standard MARTE

Si dans les deux parties précédentes 1 et 2, on a présenté respectivement l'Ingénierie Dirigée par les Modèles (IDM) et le développement de Systèmes Embarqués Temps-Réel (SETRs), dans cette partie on présentera le nouveau standard OMG : MARTE (Modeling and Analysis of Real-Time and Embedded systems) qui a pour but d'adopter l'IDM dans le développement des SETRs. MARTE [56] est un profil UML venant remplacer le profil SPT [40]. Comme illustré en figure 2.16, MARTE définit les fondations de la modélisation des SETRs. Ces concepts de base sont ensuite spécialisés à la fois pour la modélisation détaillée et l'analyse des préoccupations. L'intention n'est pas de définir de nouvelles méthodologies de conception ou de nouvelles techniques d'analyse des SETRs, mais de les soutenir par une riche base d'annotations.

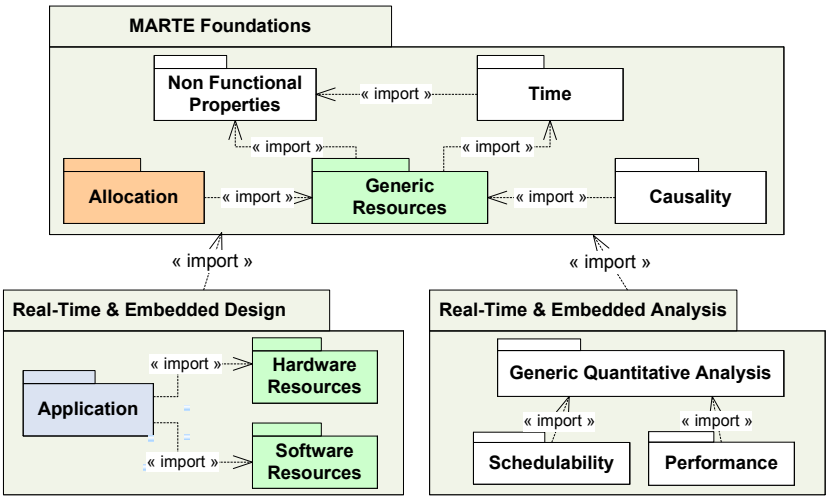


FIG. 2.16 – Architecture du standard MARTE

Un objectif majeur de MARTE est de proposer un langage de modélisation commun pour spécifier à la fois les aspects logiciels et matériels des SETRs. En effet, MARTE adopte le modèle en Y [57] grâce à trois profils identifiés par différentes couleurs dans la figure 2.16 :

- **Le modèle d'application** qui décrit les fonctionnalités du système.
- **Le modèle des ressources** qui décrit la plateforme d'exécution en prenant compte des propriétés non-fonctionnelles. Ce modèle se compose à son tour de trois modèles :
  - *Le modèle des ressources génériques* qui reprend les concepts définis dans SPT.
  - *Le modèle des ressources logicielles* qui décrit la plateforme d'exécution logicielle (ex : système d'exploitation).
  - *Le modèle des ressources matérielles* que nous allons spécifier par la suite.
- **Le modèle d'allocation** de l'application sur les ressources (voir section 3.2).

Nous détaillerons trois modèles du profil MARTE que nous utiliserons dans nos travaux, il s'agit de NFPs, Alloc et RSM.

3.1 Non Functional Properties Modeling (NFPs)

Les propriétés d'une application sont regroupées traditionnellement en deux catégories : celles propres aux fonctionnalités que doit remplir l'application (i.e services) et celles liées

à la qualification des fonctionnalités attendues (i.e. qualités de service). Les premières sont dites fonctionnelles, les secondes non-fonctionnelles (NFPs). Les NFPs fournissent des informations sur différentes caractéristiques telles que les délais d'exécution, l'utilisation de la mémoire, la consommation d'énergie, les bandes passantes etc.

Le paquetage NFPs [56, 58] de MARTE formalise un ensemble de concepts de modélisation permettant la description précise et complète des informations non-fonctionnelles. Ce paquetage vise donc à qualifier et à typer de manière standard les propriétés non-fonctionnelles. Pour cela il étend les types de données d'UML par les principaux types manipulés dans le domaine de l'embarqué et du temps-réel (ex : la fréquence, le débit, la consommation). Ces types standards sont fournis sous une forme de librairie BasicNFP_Types que l'utilisateur peut importer. Par ailleurs, l'utilisateur pourra également définir ses propres types tout en utilisant une notation standard, ce qui n'existait pas auparavant dans UML. Ensuite, MARTE permet au travers du langage VSL (Value Specification Language) [56] de définir une syntaxe concrète associée à chacun de ces types. VSL permet de décrire des constantes, des variables, des expressions complexes, et des expressions de temps.

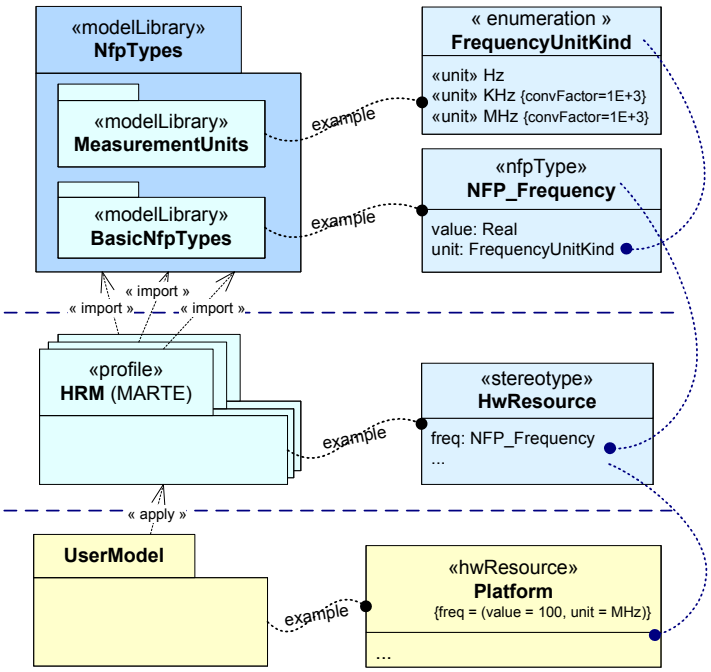


FIG. 2.17 – Usage de NFPs

La figure 2.17 montre un exemple d'utilisation de NFPs et de VSL. La librairie BasicNFP_Types fournie par MARTE définit le type NFP_Frequency On exprime une valeur donnée de ce type en VSL sous la forme (value = 100, unit = MHz).

3.2 Allocation Modeling (Alloc)

L'allocation [56] est l'opération qui consiste à placer des éléments fonctionnels de l'application sur des ressources de la plateforme d'exécution. Comme on l'a vu précédemment, l'allocation est une étape centrale dans le processus de développement

des systèmes embarqués temps-réel. MARTE permet de décrire séparément le modèle de l'application et le modèle de la plateforme, il s'agit maintenant d'associer ces deux modèles via un modèle d'allocation qu'on représente en partie dans la figure 2.18. L'allocation est le résultat d'une distribution spatiale ou d'un ordonnancement dans le temps. La distribution spatiale est le placement d'un calcul sur un processeur, de données sur une mémoire ou de dépendances de contrôle/données sur une ressource de communication. L'ordonnancement est cependant une organisation dans le temps de plusieurs activités allouées à une même ressource.

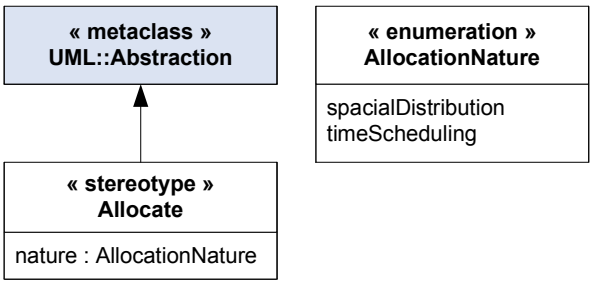


FIG. 2.18 – Modèle d'allocation

3.3 Repetitive structure Modeling (RSM)

Dans un contexte où le parallélisme est de plus en plus présent au coeur des systèmes embarqués pour des raisons que nous avons explicitées dans notre introduction, le profil RSM [56, 59] de MARTE permet de décrire de manière compacte les structures répétitives. Nous signifions par structure répétitive tout système composé d'une répétition des mêmes éléments structurels, liés entre eux par des connexions régulières. RSM introduit principalement deux extensions à UML :

- Des multiplicités multidimensionnelles pour décrire la répétition sous forme matricielle.
- La topologie d'ArrayOL [60] pour décrire les liens réguliers entre éléments répétés.

La figure 2.19 représente un exemple typique d'une grille de 4x4 processeurs où chaque processeur est connecté à ses voisins. Cette structure est efficacement modélisée grâce au stéréotype «interRepetition» de RSM.

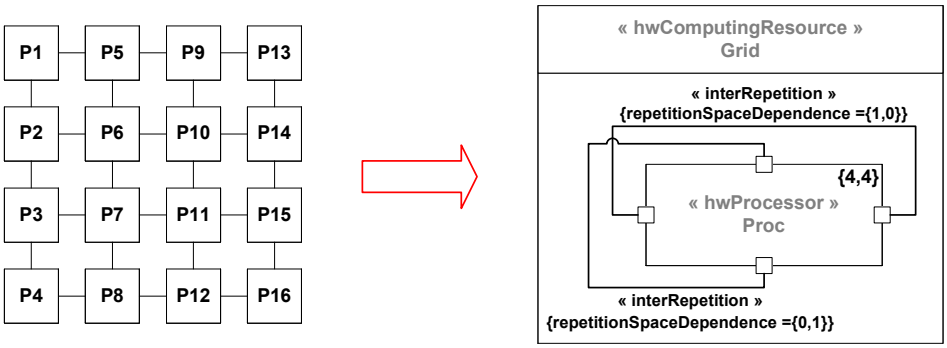


FIG. 2.19 – Modélisation d'une grille 4x4 processeurs

4 Robotique mobile

Dans le cadre de ce travail de thèse, nous développerons une chenille de robots unicycles comme un cas d'étude pour mettre en valeur nos contributions. En général, la robotique nécessite un grand nombre de compétences dans différentes disciplines (modélisation, localisation, perception, contrôle-commande, mécanique, autonomie, etc). La robotique mobile [61, 62] s'intéresse particulièrement à la conception des robots à roues, par opposition notamment aux robots manipulateurs, marcheurs ou sous-marins. Bien avant la réalisation mécanique d'un robot mobile et la conception de son architecture informatique embarquée (logiciel/matériel), une étape primordiale consiste à établir son modèle cinématique et à étudier sa commandabilité. Cette étape nécessite une bonne connaissance de la géométrie différentielle [63, 64] et de la commande non-linéaire [65, 66], nous en survolerons dans la suite quelques principes qui nous ont été nécessaires.

4.1 Modélisation cinématique

Rappelons quelques définitions de base :

Définition 1 *Soit Φ une application différentiable de $\mathbb{R}^n$ vers $\mathbb{R}^{n-p}$ ($0 \leq p < n$), s'il existe un x_0 t.q $\Phi(x_0) = 0$ et que l'application linéaire tangente $D\Phi(x)$ est de rang plein $(n-p)$ dans un voisinage V de x_0, alors on appelle variété différentiable de dimension p l'ensemble X t.q :*

$$X = \{x \in V |\ \Phi(x) = 0\}$$

Notons que l'application linéaire tangente $D\Phi(x)$ est la matrice jacobienne de taille $(n-p) \times n$ dont l'élément situé à la ligne i et à la colonne j est $\frac{\partial \Phi_i}{\partial x_j}(x)$.

Définition 2 *On appelle l'espace tangent à X au point $x \in X$, l'espace vectoriel T_xX de dimension $(n-p)$ t.q $T_xX = \ker D\Phi(x)$.*

Définition 3 *Un champ de vecteurs f sur X est une application analytique qui à tout $x \in X$ fait correspondre le vecteur $f(x) \in T_xX$.*

La modélisation cinématique consiste à décrire le mouvement du système et à traduire mathématiquement les lois géométriques et physiques (mécanique) qu'il subit, sous forme d'une équation différentielle : $\dot{x} = f(x)$, où f est un champ de vecteurs et x est le vecteur d'état du système. x est un ensemble de n coordonnées suffisants pour situer tous les points du système dans l'espace. Il existe une et une seule solution à cette équation et elle est appelée courbe intégrale de f.

Quand un système est sous l'action d'un nombre de variables appelées entrées ou commandes, on dit que le système est commandé et son équation différentielle s'écrit sous la forme :

$$\dot{x} = f(x, u)$$

le vecteur $u \in \mathbb{R}^m$ est le vecteur des entrées. Notons que x et u sont des fonctions du temps (c-à-d $t \mapsto x(t)$ et $t \mapsto u(t)$).

Exemple : Robot unicycle

Dans la figure 2.20, on représente un robot unicycle constitué de deux roues fixes de même axe et d'une roue folle centrée. Toutes les roues ont le même rayon r. Le mouvement est conféré au robot par les deux roues arrières qui sont motrices et indépendantes. Par conséquent, le vecteur de commande est $u = (\omega_d, \omega_g)$ et le vecteur d'état du robot est $x = (x, y, \theta, \varphi)$.

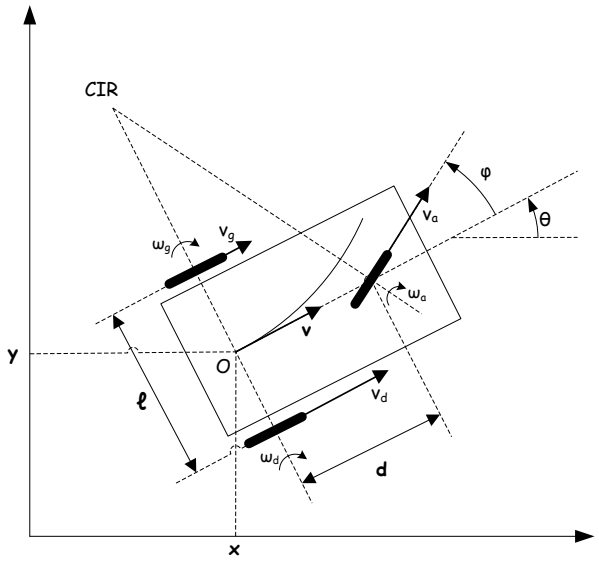


FIG. 2.20 – Un robot mobile unicycle

Pour obtenir l'équation du mouvement d'un robot à roues, on considère que le roulement au niveau de chaque roue est sans glissement, ce qui se traduit mathématiquement par une vitesse nulle au niveau du point de contact entre la roue et le sol. On considère aussi qu'à tout moment, les axes de rotation de toutes les roues se croisent au même point CIR (Centre Instantané de Rotation) autour duquel le robot mobile pivote à cet instant (voir figure 2.20). On obtient ainsi l'équation différentielle suivante :

$$\begin{aligned}
\dot{x} &= \tfrac{r(\omega_d+\omega_g)}{2}\cos\theta, \\
\dot{y} &= \tfrac{r(\omega_d+\omega_g)}{2}\sin\theta, \\
\dot{\theta} &= \tfrac{r(\omega_d-\omega_g)}{\ell}, \\
\varphi &= \arctan\left(\tfrac{2d}{\ell}.\tfrac{\omega_d-\omega_g}{\omega_d+\omega_g}\right)
\end{aligned}$$

Si on retire la dernière équation qui est déjà intégrée, on retrouve la forme de l'équation différentielle d'un système commandé $(\dot{x}, \dot{y}, \dot{\theta}) = f((x, y, \theta), (\omega_d, \omega_g))$. On voit bien que cette équation n'est pas linéaire. Quand cette équation n'est pas intégrable, on parle de non-holonomie.

4.2 Non-holonomie

Les robots mobiles sont très souvent modélisés par des systèmes non-holonomes, un système non-holonome est un système dont l'une des équations n'est pas intégrable. Il est

de ce fait impossible d'éliminer les dérivées premières dans ces équations. Les contraintes de roulement sans glissement sont typiquement des contraintes non-holonomes. Concrètement, l'existence de contraintes non-holonomes implique que le robot ne peut pas effectuer certains mouvements instantanément, il faudra manoeuvrer.

Pour vérifier l'intégrabilité d'un système d'équations, on utilise le théorème de Frobenius qui existe sous plusieurs versions [66, 62]. Ce théorème vérifie que la distribution D déduite du système d'équations est involutive (c-à-d $[D, D] \subset D$ où la relation $[,]$ est le crochet de Lie). Quand cela est vrai, le système est intégrable. Dans le cas contraire, on peut calculer le nombre des équations non intégrables.

4.3 Commandabilité

La commandabilité est une caractéristique structurelle d'un système. Elle vérifie si on peut le commander pour aller d'un état à un autre. Mathématiquement :

Définition 4 *On dit que le système $\dot{x} = f(x, u)$ est commandable si, étant donnés deux points quelconques x_i et x_f de X, il existe $t \mapsto \bar{u}(t)$ de $[0, T]$ dans $\mathbb{R}^m$ t.q la solution unique $\bar{x}(t)$ partant de $\bar{x}(0) = x_i$ arrive à $\bar{x}(T) = x_f$.*

La commandabilité d'un système linéaire peut être vérifiée grâce au critère de Kalman [66], et la commandabilité d'un système non-linéaire peut être vérifiée grâce au théorème de Chow [62].

4.4 Platitude

La propriété de platitude différentielle a été introduite par M. Fliess, J. Lévine, P. Martin et P. Rouchon en 1992 [67, 66], ce principe a révolutionné la commande des systèmes non-linéaires. Un système non-linéaire plat est un système pour lequel il est possible de trouver un système linéaire équivalent. Ainsi, commander ce système non-linéaire plat se résume à commander le système linéaire équivalent, chose qu'on sait résoudre.

Définition 5 *Un système non-linéaire modélisé par l'équation différentielle*

$$\dot{x} = f(x, u), x = (x_1, \ldots, x_n), u = (u_1, \ldots, u_m),$$

est plat s'il existe un vecteur $y = (y_1, \ldots, y_m)$ à m composantes indépendantes et une fonction analytique h t.q

$$y = h(x, u_1, \ldots, u_1^{(\beta_1)}, \ldots, u_m, \ldots, u_m^{(\beta_m)}),$$

et s'il existe deux fonctions analytiques A et B (inverses de h) pour retrouver l'état du système t.q

$$\begin{aligned} x &= A(y_1, \ldots, y_1^{(\alpha_1)}, \ldots, y_m, \ldots, y_m^{(\alpha_m)}), \\ u &= B(y_1, \ldots, y_1^{(\alpha_1+1)}, \ldots, y_m, \ldots, y_m^{(\alpha_m+1)}), \end{aligned}$$

Le vecteur $y = (y_1, \ldots, y_m)$ est appelé sortie plate du système. Le système non-linéaire plat est dit L-B (Lie-Backlund) équivalent au système linéaire

$$y_1^{(\alpha_1+1)} = v_1, \ldots, y_m^{(\alpha_m+1)} = v_m.$$

$(v_1, \ldots, v_m)$ *est le vecteur d'entrées.*

Un système plat peut admettre plusieurs jeux de sorties plates. Dans le cadre d'un problème de suivi de trajectoire, la propriété de platitude permet d'exprimer les commandes de la trajectoire à suivre en fonction des composantes d'une sortie plate et de ses dérivées successives (puisqu'on sait commander des systèmes linéaires). Ensuite, on peut, grâce à A et B, reconstituer les commandes réelles du système u.

Un grand nombre de robots mobiles non-holonomes sont plats. A l'exemple de la voiture à n chariots [68, 67] dont nous nous sommes inspirés pour modéliser et commander notre cas d'étude. Notre chenille de robots est un système non-linéaire plat.

CHAPITRE 3

Modélisation, simulation et implémentation de l'architecture matérielle

Dans ce chapitre, nous adapterons le langage de modélisation UML à la spécification du matériel. Ainsi, la conception du matériel pourra profiter des avancées réalisées dans le cadre de l'IDM. Pour ce faire, nous avons créé le profil HRM qui permet de décrire une architecture matérielle sous différentes vues et à différents niveaux de détail. HRM fait partie du profil MARTE récemment standardisé par l'OMG.

Souvent, le développeur du matériel n'est pas habitué aux techniques de modélisation et se perd dans l'immense nombre de concepts UML. Nous lui proposons donc une méthodologie de conception du matériel qui l'aidera à utiliser UML et HRM de manière efficace. Notre méthodologie restreint les concepts UML et fixe leurs sémantiques. Elle garantit, en conséquence, la consistance des modèles construits.

La simulation du matériel est aujourd'hui une pratique courante dans le développement des systèmes embarqués. Elle offre une flexibilité considérable qui implique un gain de temps et de coût appréciables. Nous fournirons, dans ce chapitre, un outillage complet et automatisé depuis la modélisation du matériel jusqu'à sa simulation. Nous montrerons comment HRM peut interfacer les outils de simulation.

La simulation permet une exploration architecturale plus rapide et plus large car elle n'est pas contrainte par la disponibilité physique du matériel. Cependant une fois que l'architecture matérielle est validée par la simulation, il est nécessaire de l'implémenter. Nous monterons comment HRM peut communiquer avec les standards d'implémentation du matériel.

1 HRM (Hardware Resource Model)

Le domaine du matériel est aujourd'hui extraordinairement varié, différentes architectures et énormément de composants existent. Il continue pourtant à évoluer grâce aux nouvelles technologies. Par conséquent, modéliser un tel domaine nécessite un langage hautement expressif.

Le paradigme objet et l'ingénierie dirigée par les modèles ont remarquablement amélioré l'efficacité du développement logiciel, grâce notamment aux mécanismes tels la généralisation, la composition, l'encapsulation, la séparation des préoccupations entre structure et comportement, l'abstraction sous différentes vues et le raffinement. En effet, UML est à présent couramment utilisé parmi la communauté du logiciel. D'autre part, ces mécanismes d'UML énumérés se prêtent bien à la modélisation du matériel (voir aussi chapitre 2 section 2.3.1). Notre objectif est donc d'adopter UML comme langage de description de plateformes matérielles pour bénéficier de ses avantages et son expressivité et unifier de la sorte le processus de développement entre la composante logicielle et la composante matérielle des systèmes embarqués hétérogènes.

HRM est un profil UML pour décrire du matériel existant ou bien en concevoir un nouveau, à travers plusieurs vues et différents niveaux de détail. HRM regroupe la plupart des concepts du matériel sous une taxinomie hiérarchique avec plusieurs catégories selon leur nature, fonctionnalité, technologie et forme. Il comprend deux vues, une ***vue logique*** qui classifie les ressources du matériel selon leurs propriétés fonctionnelles, et une ***vue physique*** qui les distingue selon leurs propriétés physiques.

1.1 Objectif

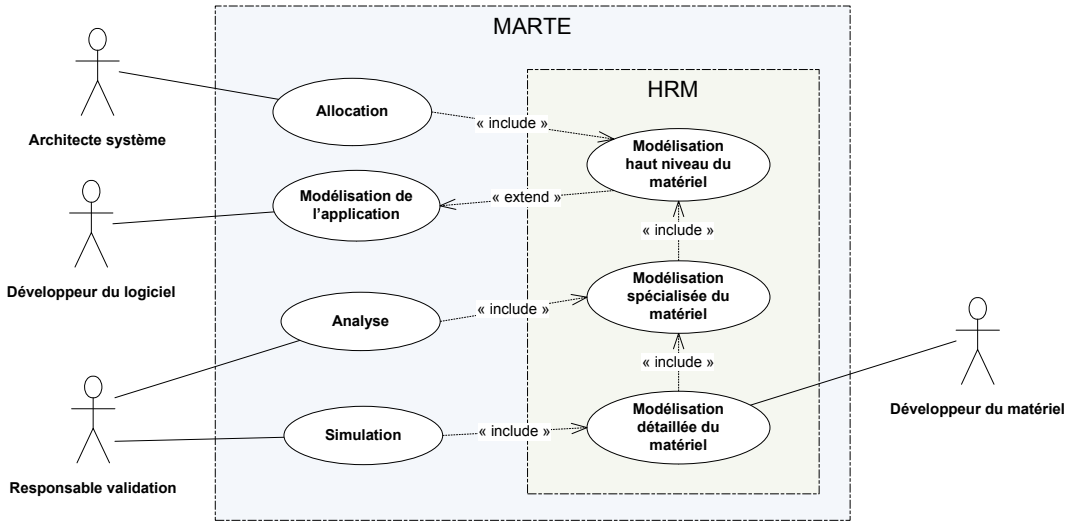


FIG. 3.1 – Cas d'utilisation du profil HRM

Le diagramme de déploiement d'UML, fournit plusieurs concepts dont `DeploymentTarget`, `Node` et `Device`, qui peuvent servir à décrire approximativement une plateforme matérielle ciblée par des artefacts logiciels. Notre objectif est plus large, nous souhaiterions couvrir plusieurs aspects de modélisation dans le cadre de MARTE. La figure

3.1 représente le diagramme de cas d'utilisation de HRM, il s'agit de trois cas d'utilisation correspondant à trois niveaux d'abstraction :

Conception du logiciel et allocation sur la base d'une description haut niveau de l'architecture matérielle en ne mettant en valeur que les propriétés clés des ressources disponibles comme le jeu d'instructions du processeur ou bien la taille mémoire disponible. Nous escomptons qu'un modèle à un tel niveau d'abstraction sera une alternative formelle au diagramme de blocs. Car même si ce dernier est souvent utilisé dans le développement système en général et matériel en particulier, il n'a aucune syntaxe graphique standard, aucune représentation textuelle à l'image de XMI pour UML, et donc aucun moyen de récupération de données pour diverses manipulations comme le raffinement, la transformation de modèles ou la génération de code.
Ce cas d'utilisation profitera principalement aux architectes système pour décrire à un haut niveau d'abstraction une plateforme matérielle existante ou alors proposer une première étape de conception d'une architecture nouvelle.

Analyse d'une description spécialisée du matériel. Il s'agit d'un modèle métier dont la nature des détails dépend du type d'analyse. Car si, par exemple, l'analyse d'ordonnancement requiert des informations sur la puissance du processeur, l'organisation mémoire et la bande passante du bus pour calculer entre autres le WCET (Worst Case Execution Time). L'analyse de la consommation d'énergie va plutôt nécessiter les informations sur la consommation, la dissipation de chaleur et la disposition des différents composants matériels.
Ce cas d'utilisation de HRM permettra de profiter de cette capacité qu'UML possède, à séparer les préoccupations grâce à différentes projections (vues) du même modèle.

Simulation d'une description détaillée du matériel sous forme d'un modèle dont le niveau de détail dépendra de la précision de simulation. En effet, la simulation de performance exige en entrée la microarchitecture exacte du processeur et les timings mémoire, pendant que la simulation fonctionnelle ne réclame que la famille du jeu d'instructions.
Ce cas d'utilisation est amplement traité en section 3.

1.2 Vue d'ensemble

Faisant partie de MARTE, le profil HRM bénéficie des autres profils et librairies de modèles définis dans ce cadre. La vue domaine, illustrée par la figure 3.2, met en évidence ces dépendances. HRM et SRM importent le même modèle générique de ressources GRM et ont donc une structure commune qui facilite les étapes d'intégration entre logiciel et matériel. Rappelons que SRM permet la description des ressources et services de la plateforme d'exécution logicielle en fournissant une interface pour les systèmes d'exploitation multitâches. HRM exploite particulièrement la librairie BasicNFP_Types [56] qui regroupe une riche variété de types complexes et ayant des unités, tels la durée `NFP_Duration`, la taille de donnée `NFP_DataSize`, la bande passante `NFP_TxRate` ou aussi la puissance électrique `NFP_Power`. Ces types sont évidemment indispensables à la modélisation du matériel.

le profil HRM se compose de deux sous-profils, le profil `HwLogical` (section 1.4) explicitant le côté fonctionnel du matériel et le profil `HwPhysical` (section 1.5) explicitant sa

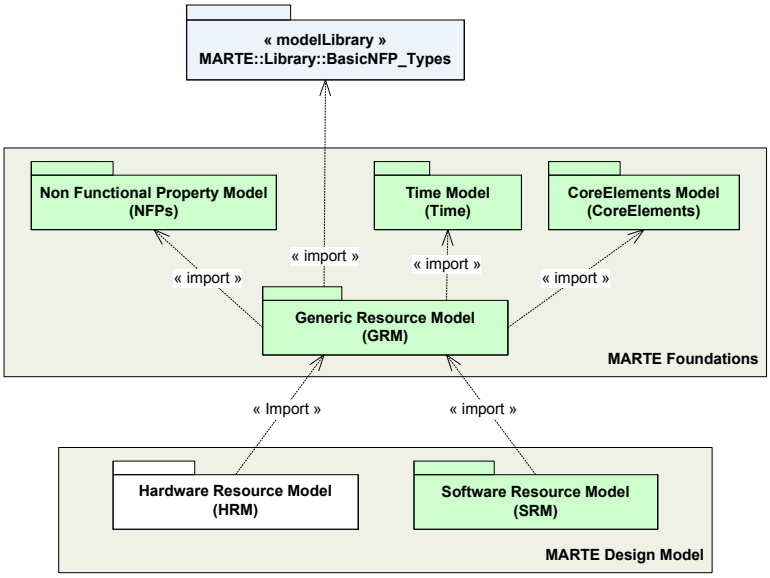


FIG. 3.2 – Dépendances de HRM (vue domaine)

partie physique. Ces deux profils sont des spécialisations d'un modèle général (section 1.3) commun. Ils sont orthogonaux mais complémentaires car ils fournissent deux différentes abstractions du matériel qui en fusionnant complètent le modèle. Chaque modèle de profil est à son tour composé de plusieurs packages qui ne sont point des sous-profils, mais servent seulement de taxinomie (voir figure 3.3).

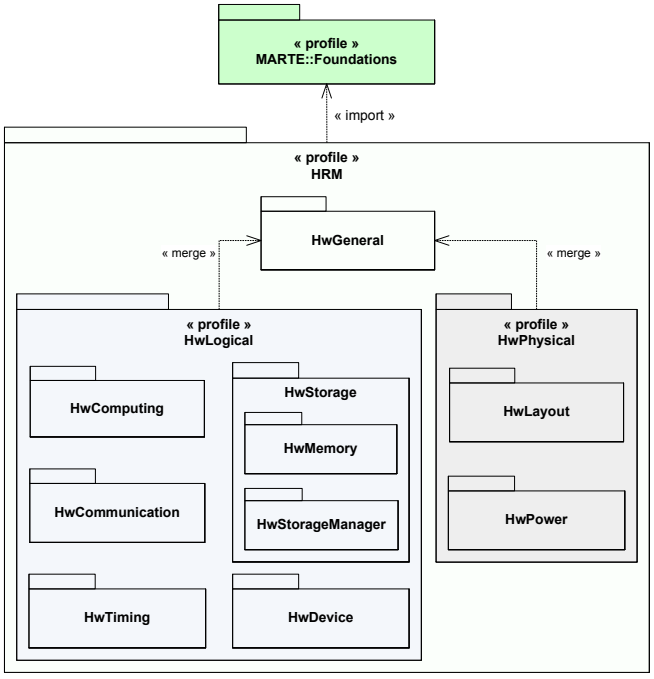


FIG. 3.3 – Structure de HRM

Les stéréotypes introduits dans HRM sont organisés en un arbre d'héritages successifs depuis les stéréotypes génériques jusqu'aux spécifiques, aucun stéréotype n'est orphelin. Ceci explique l'aptitude du profil HRM à supporter plusieurs niveaux de détail. Cet aspect

est aussi renforcé par des `tag definitions` facultatives qui ne sont spécifiées qu'une fois nécessaires. Le profil HRM a une autre qualité, celle de supporter la plupart des concepts du matériel grâce à un grand nombre de stéréotypes et encore une fois, à son architecture en étages d'héritages. En effet, si aucun stéréotype spécifique ne correspond à un composant particulier, un stéréotype générique devrait convenir. Ceci permet donc d'inclure les nouveaux concepts du matériel de nouvelle nature ou issus d'une nouvelle technologie. En plus des stéréotypes, plusieurs types et plusieurs règles OCL sont spécifiés pour enrichir la sémantique et garantir la cohérence du métamodèle.

Afin de faciliter l'utilisation du profil HRM, les noms des stéréotypes et les noms de leurs attributs sont choisis selon la terminologie conventionnelle du monde du matériel. De plus, ils sont préfixés par le label `Hw` pour lever l'ambiguité entre logiciel et matériel. Par exemple `HwTimer` dénote un compteur matériel et non pas un programme logiciel. Graphiquement et en coordination avec SRM, le profil HRM dispose aussi de plusieurs notations. Il associe une icône à chaque concept logique et une forme à chaque concept physique. Il permet ensuite de représenter chaque composant comme une grille rectiligne où chacun de ses sous-composants occupe une position particulière. Ainsi les diagrammes UML sont d'un point de vue graphique plus proches de la disposition réelle (voir exemple 1.6.3).

Dans la suite de cette section, les concepts du matériel seront présentés catégorie par catégorie, en plusieurs diagrammes, et au niveau métamodèle. Dans ce document concis, les méta-concepts sont sommairement expliqués, le lecteur pourra trouver le détail complet avec les descriptions de classe dans le document final de MARTE [56].

1.3 Modèle général

Le modèle général de HRM (figure 3.4) définit la structure typique d'une plateforme d'exécution [69]. Il est ainsi une base commune entre le modèle logique et le modèle physique qui en font deux spécialisations différentes.

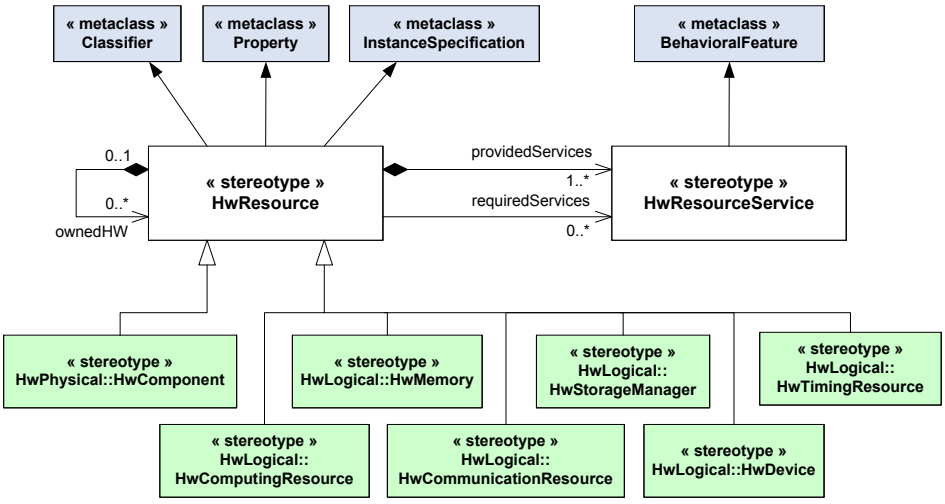


FIG. 3.4 – Le modèle général de HRM

Le stéréotype `HwResource` dénote une entité générique du matériel. Il fournit au moins un `HwResourceService` et en requiert d'autres depuis d'autres `HwResource`(s) de la

même plateforme d'exécution (voir exemple 1.6.1). En effet, la collaboration des ressources par le biais de leurs services caractérise la plateforme d'exécution. Chaque `HwResource` peut contenir des `ownedHW` ressources qui à leur tour peuvent être composites et ainsi de suite. Ce mécanisme de composition permet de modéliser le matériel à différentes granularités. Notons que dans ce schéma d'encapsulation successif, on considère comme plateforme la `HwResource` englobante de granularité zéro, et si une telle ressource n'existe pas, on considère cela comme une erreur de modélisation. D'un point de vue structurel, le stéréotype `HwResource` est similaire à la métaclasse `Component` d'UML, mais sémantiquement il définit une entité d'exécution matérielle dont les services sont annotés par des caractéristiques de qualité de service [70].

Afin d'augmenter la flexibilité lors de la modélisation avec le profil HRM, `HwResource` étend les métaclasses générales du noyau d'UML `Classifier`, `Property` et `InstanceSpecification`. Il est par conséquent possible d'utiliser HRM avec tous les diagrammes de structure d'UML (diagramme de classe, diagramme de composant, diagramme de composite structure...). De même `HwResourceService` étend la métaclasse UML `BehavioralFeature` et pourra donc être associé à toute vue comportementale d'UML. En étendant aussi `InstanceSpecification`, on permet l'application des stéréotypes aux deux niveaux classe et instance. Toutefois, la sémantique diffère, car si les `tag definitions` dénoteraient des caractéristiques statiques et communes à tous les objets de la classe dans le premier cas, elles représenteront plutôt des caractéristiques dynamiques et spécifiques à chaque objet dans le second cas (voir exemples 1.6.2 et 1.6.3).

Tous les stéréotypes du profil HRM héritent de `HwResource`, ils bénéficient alors de la même structure et des mêmes extensions. La figure 3.4 montre le second étage de stéréotypes, il s'agit de `HwComponent` du côté du sous-profil `HwPhysical` et de `HwComputingResource`, `HwMemory`, `HwStorageManager`, `HwCommunicationResource`, `HwTimingResource` et `HwDevice` du côté du sous-profil `HwLogical`. A ce niveau, les stéréotypes restent encore génériques, cependant les étages suivants d'héritages sont spécifiques.

1.4 Modèle logique

L'objectif du modèle logique est de classifier les ressources du matériel selon leurs fonctionnalités, en distinguant les ressources de calcul, de stockage, de communication, de temps et les ressources auxiliaires. Cette classification est commune à plusieurs travaux antérieurs [40, 41, 43]. Elle est basée principalement sur les services que chaque ressource offre et ne porte pas sur sa nature ni sa forme. Toutefois, elle n'est pas catégorique et les différents concepts ne sont pas nécessairement incompatibles. Il existe des composants matériels qui jouent plusieurs rôles au sein de la même plateforme. A l'exemple d'un DMA(Direct Memory Access) qui participe à la gestion de la mémoire et au contrôle de communication entre ressources.

Le modèle logique comprend six packages (voir figure 3.3), chacun pour une catégorie fonctionnelle. Dans la suite de cette section, nous présenterons séparément chaque package avec le diagramme du métamodèle correspondant et le descriptif des concepts qu'il définit dans l'ordre des héritages.

1.4.1 HwComputing package

Le package `HwComputing` définit l'ensemble des ressources actives de calcul. Ces ressources sont souvent complexes et composites, elles contiennent nombre d'autres ressources avec différentes fonctions et provenant d'autres packages.

Métamodèle

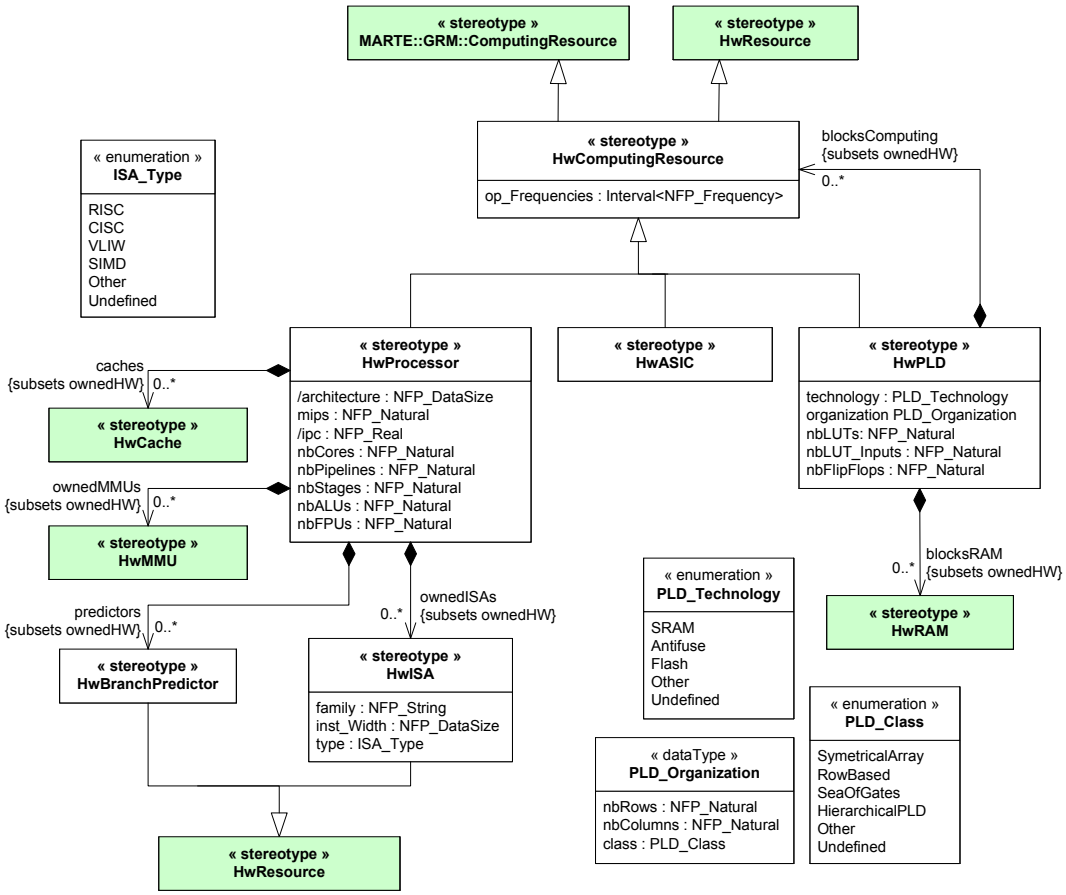


FIG. 3.5 – Le modèle des ressources de calcul

Dans la figure 3.5, `HwComputingResource` est un stéréotype générique. Il peut être spécialisé (`HwASIC`) et ce type de ressources est puissant mais nullement flexible. Il peut être configurable (`HwPLD`) et il existe différentes technologies avec différentes potentialités telle la reconfiguration dynamique pour ceux en SRAM. Et finalement, il peut être programmable (`HwProcessor`) avec une microarchitecture à spécifier.

Description des concepts

➢ ***HwComputingResource*** est un concept haut niveau qui symbolise une ressource active d'exécution. Elle est ordinairement cadencée à une fréquence donnée comprise dans un intervalle de fréquences opérationnelles `op_Frequencies`.
Exemple : CPU (`HwProcessor`), FPGA (`HwPLD`)...

≻ ***ISA_Type*** est un type énuméré qui distingue les différents types des jeux d'instructions. Exemple : `SIMD` (Single Instruction Multiple Data) est adopté par les ressources de calcul vectoriel.

≻ ***HwISA*** (Instruction Set Architecture) est une métaclasse qui modélise un jeu d'instruction implémenté par une partie de la microarchitecture du processeur. Elle a une famille (x86, ARM, MIPS...), une largeur d'instruction en nombre de bits, et un type. Pendant le processus de simulation du matériel, ce concept devrait être raffiné.
Exemple : Intel 386 appartenant à la famille x86, a une architecture 32-bit de type CISC.

≻ ***HwBranchPredictor*** dénote une politique de branchement appliquée par le processeur et qui prédit si un branchement doit être pris ou bien ignoré. Tous les processeurs avec pipeline nécessitent ce dispositif d'anticipation car il accélère considérablement l'exécution. Ce concept doit également être raffiné par une vue comportementale pour toute simulation de performance du processeur.
Exemple : les processeurs Intel Pentium appliquent une prédiction bimodale.

≻ ***HwProcessor*** représente un processeur. Il implémente un ou plusieurs jeux d'instructions et éventuellement des politiques de branchements. Il contient des unités de gestion de mémoire et du cache organisé en différents types et niveaux. `HwProcessor` a plusieurs attributs, `mips` (million instructions per second) et `ipc` (instructions per cycle) caractérisent sa puissance globale pendant que les autres attributs concernent sa microarchitecture.
Exemple : ARM-7, TI-C6000 VLIW DSP... DSP (Digital Signal Processor) est un processeur à structure répétitif adapté au traitement intensif de signal.

≻ ***HwASIC*** (Application Specific Integrated Circuit) est une ressource de calcul dédiée et optimisée pour un usage particulier. Elle offre les services de l'application qu'elle implémente.
D'autres métamodèles tels UML for SoC [37] et UML for SystemC [38] permettent de raffiner ce concept et de décrire l'application implémentée. Ils pourront donc être greffés par l'utilisateur à ce niveau.
Exemple : les récents ASICs de compression vidéo supportent l'encodage en temps réel d'une source vidéo en un format optimisé.

≻ ***PLD_Class*** est un type énuméré qui distingue les différentes classes d'organisation d'un `HwPLD`.

≻ ***PLD_Organization*** est un type structuré qui spécifie l'organisation d'un `HwPLD` en termes de nombre de lignes, nombre de colonnes et classe d'organisation.

≻ ***PLD_Technology*** est un type énuméré qui distingue les différents procédés technologiques des `HwPLD`(s). Chaque procédé implique un certain usage et des fonctionnalités différentes.

≻ ***HwPLD*** (Programmable Logic Device) est un concept générique qui regroupe toute ressource de calcul dont le circuit logique peut être reconfiguré via une interface. Ces ressources sont souvent composées de nombreuses cellules logiques élémentaires librement assemblables, mais de récents composants peuvent aussi contenir plusieurs processeurs, des unités arithmétiques et une quantité considérable de RAM. Chaque `HwPLD` est d'abord caractérisé par son organisation et la technologie de sa réalisation et ensuite par la quantité de ressources qu'il contient.

Exemple : FPGA (Field Programmable Gate Array) désigne des composants basés sur des cellules SRAM pendant que EPLD (electrically Erasable PLD) désigne des composants à technologie Flash.

1.4.2 HwMemory package

Métamodèle

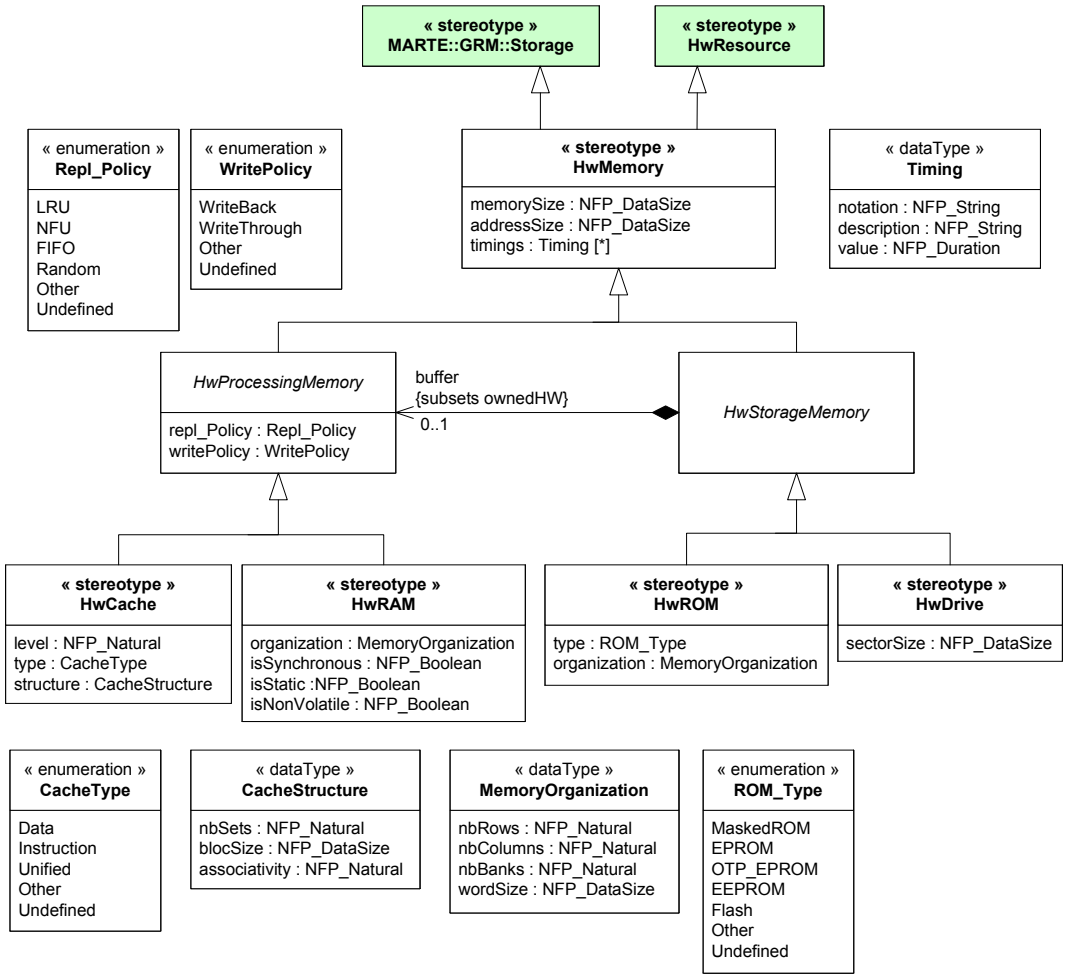


FIG. 3.6 – Le modèle des ressources mémoire

Dans la figure 3.6, `HwMemory` dénote une certaine quantité de mémoire, elle peut être une `HwProcessingMemory` ou une `HwStorageMemory`. `HwProcessingMemory` est une métaclasse abstraite qui représente les mémoires de calcul rapides et volatiles. Par contre, `HwStorageMemory` représente les mémoires de stockage permanentes mais à accès relativement lents. Notons qu'en réalité, la technologie RAM prend plusieurs formes, SRAM (Static RAM) qui est rapide et souvent utilisée comme cache et SDRAM (Synchronous Dynamic RAM) qui est moins rapide mais utilisée comme mémoire principale. Toutefois, puisque la classification logique dépend d'abord de la fonctionnalité, on sépare `HwRAM` pour les mémoires principales et `HwCache` pour le cache.

Description des concepts

➢ ***Timing*** est un type structuré qui permet d'annoter une mesure de temps en relation avec une opération mémoire. Ces mesures de temps sont nécessaires à la simulation de performance avec prise en charge de la mémoire.
Exemple : tCAS (Column Address Strobe) est la latence d'accès colonne, souvent renseignée pour les `HwRAM`(s).

➢ ***HwMemory*** est un concept haut niveau qui dénote toute forme de stockage de données pendant un intervalle de temps quelconque. Il s'agit d'une ressource protégée qui offre des services de lecture et écriture (`HwResourceService`). Une `HwMemory` est principalement caractérisée par sa taille, sa largeur d'adresse et ses timings.
Exemple : disque dur (`HwDrive`), RAM (`HwRAM`)...

➢ ***Repl_Policy*** est un type énuméré qui distingue les différentes politiques de remplacement de données.
Exemple : `NFU` (Not Frequently Used) remplace la donnée qui est le moins fréquemment utilisée.

➢ ***WritePolicy*** est un type énuméré qui distingue les différentes politiques d'écriture des données volatiles.
Exemple : la politique `WriteThrough` répercute automatiquement l'écriture sur la mémoire d'origine.

➢ ***HwProcessingMemory*** est une métaclasse abstraite, à laquelle ne correspond aucun stéréotype, qui dénote une mémoire à accès rapides mais volatile. Ce type de ressources mémoire est souvent une copie d'une mémoire plus lente. En conséquence, si elle est modifiée, elle doit répercuter les changements sur la mémoire d'origine selon une politique d'écriture particulière. En plus, elle est souvent de taille limitée, elle doit donc implémenter une politique adéquate de remplacement de données.
Exemple : caches, RAMs, buffers...

➢ ***HwStorageMemory*** est une métaclasse abstraite à laquelle ne correspond aucun stéréotype et qui en opposition à `HwProcessingMemory` représente une ressource de mémoire permanente à accès relativement lents. Elle peut cependant être accélérée en lui associant un buffer (`HwProcessingMemory`) pour les données fréquemment utilisées.
Exemple : mémoires Flash, ROMs...

➢ ***CacheType*** est un type énuméré qui distingue les différents types de cache.
Exemple : Généralement, pour des raisons de performance et de coût, les premiers niveaux de cache sont séparés (données/instructions), et les derniers niveaux de cache sont unifiés.

➢ ***CacheStructure*** est un type structuré qui décrit la structure du cache. Le cache est organisé sous forme de groupes (`Set`) de blocs (`Block`). Chaque groupe est une image partielle (sous ensemble de blocs) d'une zone mémoire. L'associativité (`Associativity`) est le nombre de blocs par groupe, ce nombre est unique au sein du même cache. Si la valeur de l'associativité est 1, on dit que le cache est direct (direct mapped), par ailleurs si le nombre de groupes est 1, on dit que le cache est associatif (fully associative). De ce qui précède, on peut déduire la taille du cache en multipliant les trois attributs de sa structure. Une règle OCL est de ce fait annexé à `HwCache` vérifiant que :

$$memorySize = nbSets \times blocSize \times associativity$$

Notons encore que la description détaillée de la structure du cache est nécessaire à la simulation de performance.
Exemple : TLB est typiquement un cache associatif.

≻ ***HwCache*** est un stéréotype qui dénote une mémoire de calcul (`HwProcessingMemory`) à accès rapides où les données fréquemment utilisées sont momentanément stockées. Le cache est organisé sous une hiérarchie de niveaux. Chaque `HwCache` occupe un certain niveau, et se définit par son type et sa structure.
Exemple : le processeur PowerPC G4 contient, au niveau 1, 32KB de cache d'instructions et 32KB de cache de données et embarque, au niveau 2, 512KB de cache unifié.

≻ ***MemoryOrganization*** est un type structuré qui définit l'organisation des mémoires `HwRAM` et `HwROM`. Il s'agit d'une ou plusieurs unités (`Bank`) contenant chacune une matrice de mots. En conséquence, la taille de la mémoire est la multiplication des quatre attributs de la `MemoryOrganization`.
Exemple : 64Mo de RAM peut être organisé sous forme de $4096 \times 256 \times 4 \times 16bit$.

≻ ***HwRAM*** (Random Access Memory) est une mémoire de calcul qui fournit des services de lecture et écriture rapides et qui contrairement au `HwDrive` permet des accès aux données dans n'importe quel ordre et avec les mêmes timings. `HwRAM` peut être statique ou dynamique, et dans le deuxième cas, elle nécessite un rafraichissement périodique couteux en temps. `HwRAM` est typiquement volatile, elle perd ses données une fois éteinte mais pour remédier à cela, elle peut être reliée à une source d'énergie (`HwPowerSupply`) permanente ou bien stockée dans une `HwROM` associée avant son extinction. Une `HwRAM` est principalement caractérisée par son `organization` et ses `timings`, car ces propriétés déterminent son comportement et permettent entre autres sa simulation.
Exemple : une barrette SDRAM (Synchronous Dynamic RAM) en exemple 1.6.2.

≻ ***ROM_Type*** est un type énuméré qui distingue les différents types de ROM.
Exemple : la mémoire `Flash` est un cas particulier de `EEPROM` (Electrically Erasable Programmable ROM).

≻ ***HwROM*** (Read Only Memory) est une mémoire de sauvegarde permanente (`HwStrorageMemory`) qui fournit essentiellement des services de lecture. Selon son type, le contenu de la `HwROM` peut être réecrit ou ne peut pas l'être.
Exemple : une mémoire BIOS, la mémoire programme d'un microcontrôleur...

≻ ***HwDrive*** est une mémoire de sauvegarde permanente et généralement de grande taille. Le support de stockage peut être en permanence installé à l'intérieur de cette ressource comme il peut être remplaçable. D'un point de vue fonctionnel, l'attribut `sectorSize` correspond à la plus petite quantité de mémoire physique qu'on peut allouer.
Exemple : de récents micro-disque-durs atteignent la taille de 0.8 pouce pour plus de 4GB d'espace de stockage.

1.4.3 HwStorageManager package

Métamodèle

Dans la figure 3.7, `HwStorageManager` dénote un contrôleur mémoire. Il est de ce fait associé aux mémoires qu'il contrôle. `HwMMU` est une unité qui gère l'adressage et le contenu

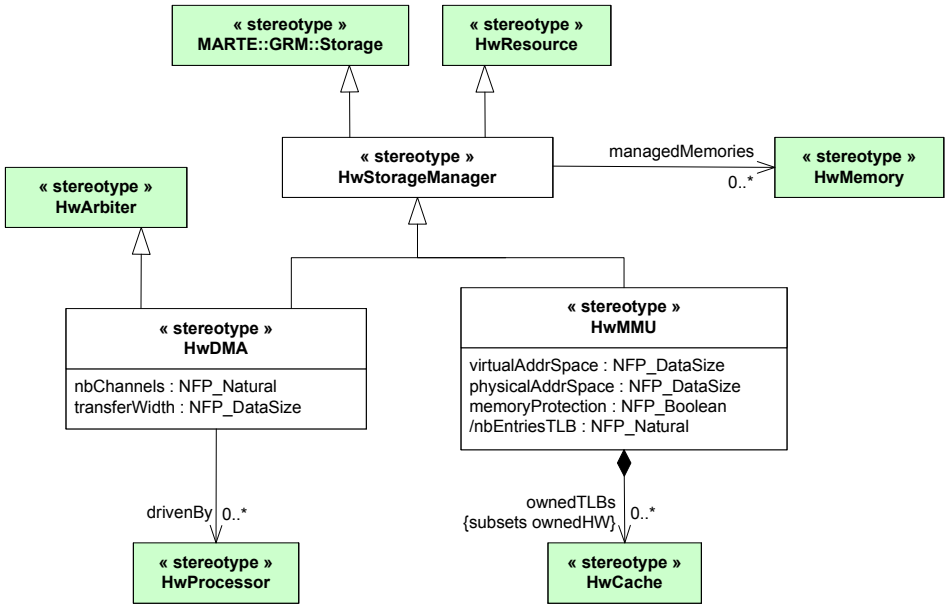


FIG. 3.7 – Le modèle des ressources de gestion de mémoire

mémoire. Pendant que `HwDMA` est une ressource contrôlée à son tour par le processeur et qu'elle libère de la gestion des transferts de blocs de données.

Description des concepts

➢ ***HwStorageManager*** est un concept haut niveau qui dénote un contrôleur mémoire qui gère l'accès et/ou le contenu des mémoires qu'il contrôle.
Exemple : un processeur, `HwMMU`, `HwDMA`...

➢ ***HwDMA*** (Direct Memory Access) est une ressource qui prend en charge l'accès à un ensemble de mémoires. Un `HwDMA` est typiquement contrôlé par un ou plusieurs `HwProcessor`(s) qui lui délèguent la gestion du bus système et des accès mémoire. Il a donc aussi la fonction d'arbitre de communication. L'attribut `nbChannels` correspond au nombre maximum de transferts simultanés que le `HwDMA` peut gérer.
Exemple : `HwDMA` fait souvent partie du chipset de la carte mère.

➢ ***HwMMU*** (Memory Management Unit) dénote une ressource de contrôle mémoire. Elle gère et accélère les accès du processeur vers la mémoire en traduisant les adresses virtuelles en adresses physiques par le biais d'un cache associatif TLB (Transfer Lookaside Buffer). Normalement, une entrée TLB conserve l'adresse virtuelle d'une page mémoire avec l'adresse physique correspondante, mais elle peut aussi conserver des informations concernant la protection mémoire, si cette option est implémentée.
Exemple : tous les processeurs actuels intègrent une `HwMMU`.

1.4.4 HwCommunication package

L'objectif du package `HwCommunication` est de grouper tous les acteurs de communication en un métamodèle indépendant qui fournit le squelette de la plateforme matérielle et qui est conforme aux constructions d'UML.

Métamodèle

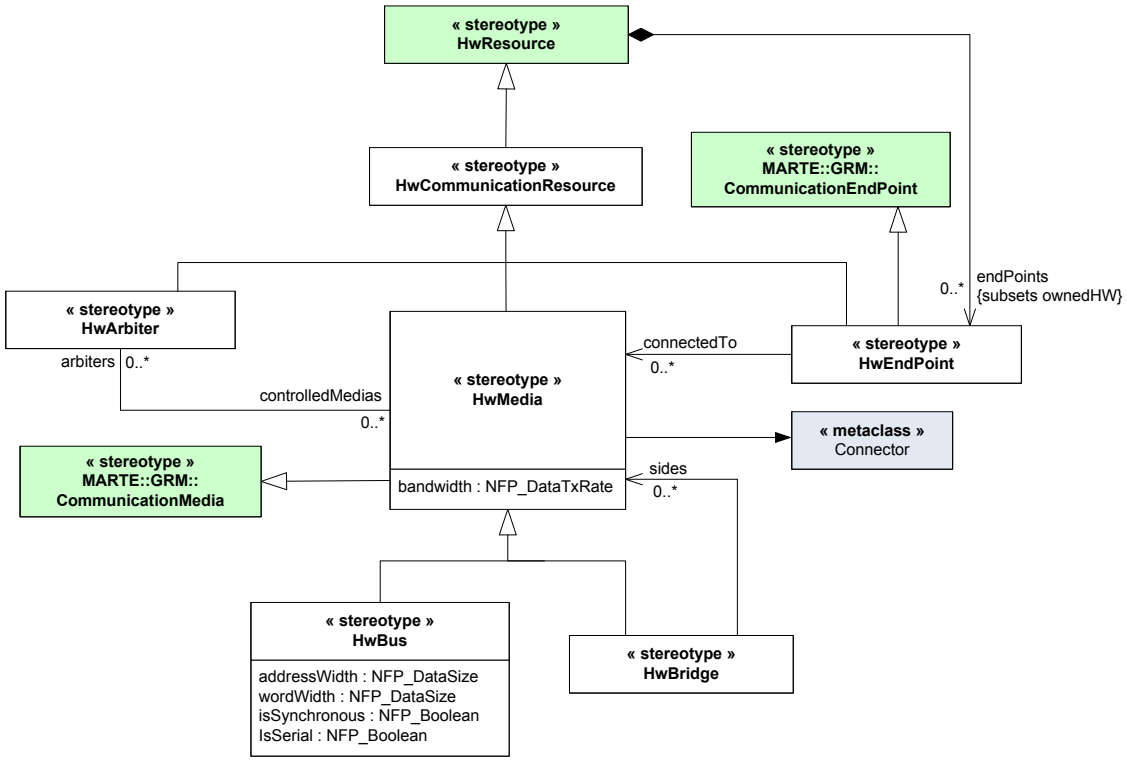


FIG. 3.8 – Le modèle des ressources de communication

Comme illustré dans la figure 3.8, `HwMedia` tient le rôle central, il représente une ressource de communication capable de transférer des données avec une certaine bande passante théorique. Il connecte plusieurs `HwEndPoint`(s). Il peut être arbitré par des `HwArbiter`(s) et il est éventuellement connecté à d'autres `HwMedia`(s) à travers des `HwBridge`(s). Notons qu'un `HwEndPoint` est un point de connexion quelconque d'une `HwRessource`.
Si un *HwMedia* symbolise toute sorte de connexion, `HwBus` dénote une connexion *câblée* avec des propriétés fonctionnelles spécifiques.

Description des concepts

➢ ***HwCommunicationResource*** est un concept haut niveau qui regroupe toute ressource de la plateforme qui participe à la communication, qu'il s'agisse du moyen de communication, des points communicants ou éventuellement d'un arbitre. Cette définition est large et concerne tous les composants du matériel car par essence une `HwResource` échange ses services en communiquant avec les autres ressources de la plateforme.
Exemple : un bus PCI (`HwBus`), un DMA (`HwArbiter`), un port ou une antenne (`HwEndPoint`)...

➢ ***HwArbiter*** est un stéréotype qui caractérise les ressources de contrôle de communication. Un `HwArbiter` prend en charge au moins un `HwMedia` et gère l'arbitrage entre les différents maîtres qui y sont connectés. Notons que *maître* est le rôle transitoire que

joue un HwEndPoint pendant une communication et il est fondamentalement différent du rôle permanent que joue un HwArbiter. Pendant la modélisation de la plateforme, une vue comportementale devrait être attachée aux HwArbiter(s) pour décrire leurs stratégies d'arbitrage.
Exemple : comme on l'a vu précédemment, un HwProcessor arbitre son bus système et délégue de manière temporaire cette tâche au HwDMA.

≻ ***HwMedia*** est un concept générique, il représente toute ressource capable de transférer des données d'un point à un autre. HwMedia a une bande passante dont la sémantique est un point de variation de HRM. L'utilisateur de HRM est libre de renseigner, ou la valeur théorique maximum de la bande passante qui n'est que rarement atteinte, ou la valeur moyenne qui est plus utile.
En plus des extensions héritées depuis HwResource (voir section 1.3), HwMedia étend la métaclasse d'UML Connector. L'utilisateur pourra donc enrichir le diagramme de composite structure d'UML en appliquant ce stéréotype ou l'un de ces fils sur les connecteurs. D'autre part, notons qu'une Association UML est un Classifier et il est donc possible de lui appliquer des stéréotypes de HRM.
Exemple : un bus, une connexion sans fil...

≻ ***HwEndPoint*** symbolise un point de connexion d'une HwResource, Il s'agit d'une interface qui sert à communiquer avec les autres HwResource(s) de la plateforme, et qui se connecte, pour ce faire, à l'extrémité d'un HwMedia. Une HwResource peut avoir plusieurs HwEndPoint(s). Typiquement, on obtient les schémas illustrés en figure 3.36 (page 90).
Exemple : ports, pins, slots...

≻ ***HwBus*** est un stéréotype concret qui dénote un HwMedia physique. Il est caractérisé par la largeur d'adresse supportée et la largeur du mot transporté. On distingue aussi un bus série d'un bus parallèle et un bus synchronisé d'un bus asynchrone. Normalement, si une ressource matérielle est représentée par un HwBus dans une vue logique, elle devrait être représentée par un HwChannel dans la vue physique correspondante (voir exemple 1.6.3) et en comparant ses caractéristiques entre les deux vues on peut tirer quelques conclusions :
– Si le nombre de fils (nbWires) du HwChannel est inférieur à la somme des largeurs de mot et d'adresse en bits ($wordWidth + addressWidth$) d'un HwBus parallèle (isSerial=false), le bus est multiplexé.
– Si de plus le nombre de fils (nbWires) du HwChannel est inférieur à la largeur du mot (wordWidth), le bus est série. Cette règle est formulée sous forme de contrainte OCL et annexée au métamodèle.

D'autres contraintes OCL locales à la vue logique vérifient qu'un bus synchronisé a une fréquence spécifiée et que sa bante passante est plus petite que le produit de sa fréquence et sa largeur de mot.

$$bandwidth \leq wordWidth \times frequency$$

Exemple : ISA (Indusrty Standard Architecture) est un bus parralèle (20bits d'adresse et 8bits de largeur de mot) et synchrone candencé à 4.77MHz.

≻ ***HwBridge*** dénote une ressource qui gère la communication entre deux ou plusieurs HwMedia(s). Elle doit souvent réaliser de complexes traductions de protocoles. Un HwBridge est aussi un HwMedia, il a donc également une bande passante, mais qui est limitée par la plus petite des bandes passantes des HwMedia(s) que celui-ci connecte.

Exemple : PCI-to-ISA est un `HwBridge` qui se comporte comme une cible PCI du côté du bus PCI et comme un maître ISA du côté du bus ISA.

1.4.5 HwTiming package

Métamodèle

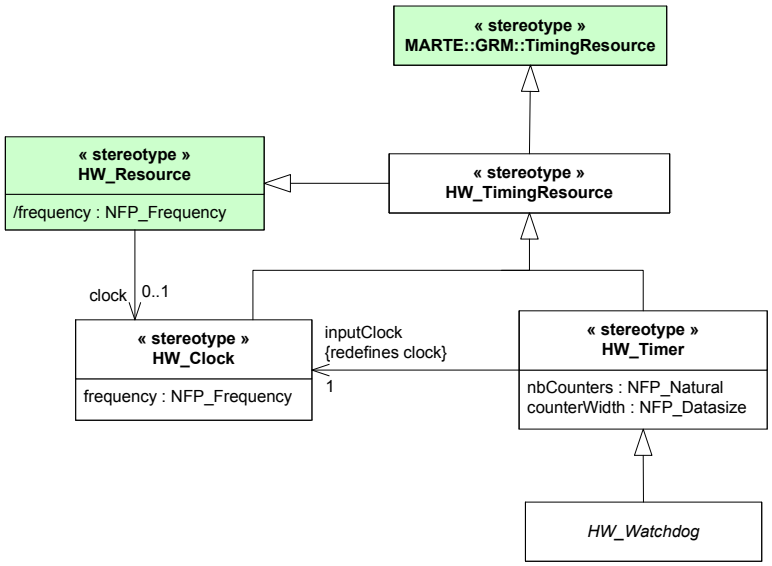


FIG. 3.9 – Le modèle des ressources de temps

La figure 3.9 définit les ressources de temps. `HwClock` est une horloge dont les pulsations sont à fréquence fixe et prédéterminée. Toute ressource du matériel peut être associée à une horloge, et par conséquent avoir sa fréquence. `HwTimer` est un ensemble de compteurs dont la largeur détermine le temps de mesure maximum. `HwWatchdog` est typiquement un compteur à rebours qui interrompt le système quand le zéro est atteint. Les services fournis et requis par ces ressources sont donnés à titre d'exemple en 1.6.1.

Description des concepts

➢ ***HwTimingResource*** est un stéréotype haut niveau qui représente des ressources offrant des services liés au temps.
Exemple : une horloge.

➢ ***HwClock*** est un concept fondamental dans le monde du matériel, il s'agit d'une pulsation périodique à une fréquence donnée. On peut donc confondre `HwClock` avec sa fréquence. Toute `HwResource` peut être liée à une `HwClock` pour son fonctionnement interne ou pour communiquer de manière synchrone avec d'autres ressources.
Exemple : un cristal de quartz.

➢ ***HwTimer*** est un ensemble de compteurs cadencés par la même `inputClock`. La valeur courante de chaque compteur est accessible à tout moment. La largeur du compteur détermine sa mesure maximale de temps en termes de cycles d'horloge ($2^{counterWidth} - 1$). Il

existe cependant des `HwTimer`(s) qui admettent la fusion de compteurs.
Exemple : la plupart des microcontrôleurs embarquent des `HwTimer`(s).

➢ ***HwWatchdog*** est un concept abstrait qui est structurellement identique à `HwTimer` mais qui offre en plus un service de notification déclenché par la fin du comptage. Il doit donc être remis à zéro par le système avant cette échéance.
Exemple : la plupart des microcontrôleurs embarquent des `HwWatchdog`(s).

1.4.6 HwDevice package

Métamodèle

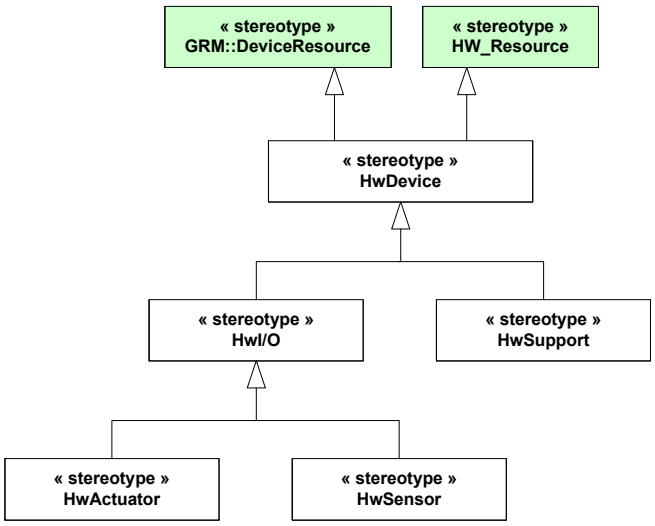


FIG. 3.10 – Le modèle des ressources auxiliaires

D'un point de vue fonctionnel, un `HwDevice` est une ressource auxiliaire dont la fonction n'est pas aussi fondamentale que celles des ressources de calcul, de mémoire ou de communication. Pourtant elle complète le fonctionnement global de la plateforme d'exécution. Il existe deux catégories, `HwI/O` pour les ressources qui interagissent avec l'environnement comme un capteur (`HwSensor`), un actionneur (`HwActuator`), un périphérique ou un écran. Et `HwSupport` qui est une ressource de support matériel tel une source d'énergie (batterie) ou un ventilateur. Vue leur nature, quelques unes des ressources `HwSupport` sont spécifiées dans le modèle physique.

Description des concepts

➢ ***HwDevice*** est un stéréotype haut niveau qui correspond à toute ressource qui fait partie de la plateforme et qui étend sa fonctionnalité.
Exemple : une sonde, une batterie...

➢ ***HwI/O*** (Input/Output) est une ressource générique qui regroupe tout composant qui interagit avec l'environnement extérieur à la plateforme. L'interaction peut être à sens unique, de l'environnement vers le composant (`HwSensor`) ou du composant vers l'environnement (`HwActuator`), comme elle peut être à double sens à l'exemple d'un

écran tactile. AUTOSAR [42] fournit un métamodèle détaillé et principalement dédié à l'automobile concernant les `HwI/O`(s). Ce profil pourra donc être greffé et utilisé à ce niveau. Exemple : une caméra (`HwSensor`), un moteur (`HwActuator`)...

➢ ***HwSupport*** est un concept générique qui dénote une ressource non fonctionnelle d'un point de vue *logique*. Elle reste néanmoins indispensable au bon fonctionnement de la plateforme.
Exemple : les ressources introduites dans le `HwPower` package du profil physique sont typiquement des `HwSupport`(s).

➢ ***HwActuator*** est une ressource de commande qui trasforme une commande de la plateforme en une action physique. Il s'agit d'une sortie du système.
Exemple : un voyant lumineux (LED), un moteur...

➢ ***HwSensor*** est une ressource de mesure qui transforme une grandeur physique observée en une grandeur utilisable par la plateforme. Il s'agit d'une entrée du système. AUTOSAR [42] distingue ces ressources selon leur sensibilité, leur résolution et leur précision.
Exemple : un sonar, une sonde...

1.5 Modèle physique

Le modèle physique représente les ressources du matériel comme des composants physiques en mettant en valeur leurs propriétés physiques. Généralement, les systèmes embarqués sont limités en surface, poids, coût et autonomie et sont amenés à évoluer dans des environnements hostiles. L'objectif de la vue physique est alors de fournir des mécanismes pour projeter les composants matériels sur la plateforme physique et entamer les analyses concernant le coût, la disposition ou l'autonomie du système. Le modèle physique contient deux packages (voir figure 3.3), ***HwLayout*** identifie les ressources du matériel selon leurs formes, leurs dimensions, leurs positions dans la plateforme et leurs conditions environnementales requises. Pendant que ***HwPower*** les identifie selon leurs consommations d'énergie, leurs dissipations de chaleur et leurs radiations électromagnétiques.

1.5.1 HwLayout package

Le package `HwLayout` classifie les composants du matériel selon leurs formes, et permet de les arranger dans des grilles rectilignes. Il rapproche de la sorte la représentation graphique des diagrammes UML, de la disposition réelle de la plateforme (voir exemple 1.6.3).

Métamodèle

Comme illustré dans la figure 3.11, `HwComponent` est une spécialisation de la métaclasse `HwResource` du modèle général avec en plus une multitude de propriétés physiques. HwComponent peut être également composite et peut prendre plusieurs formes : puce, carte, port...

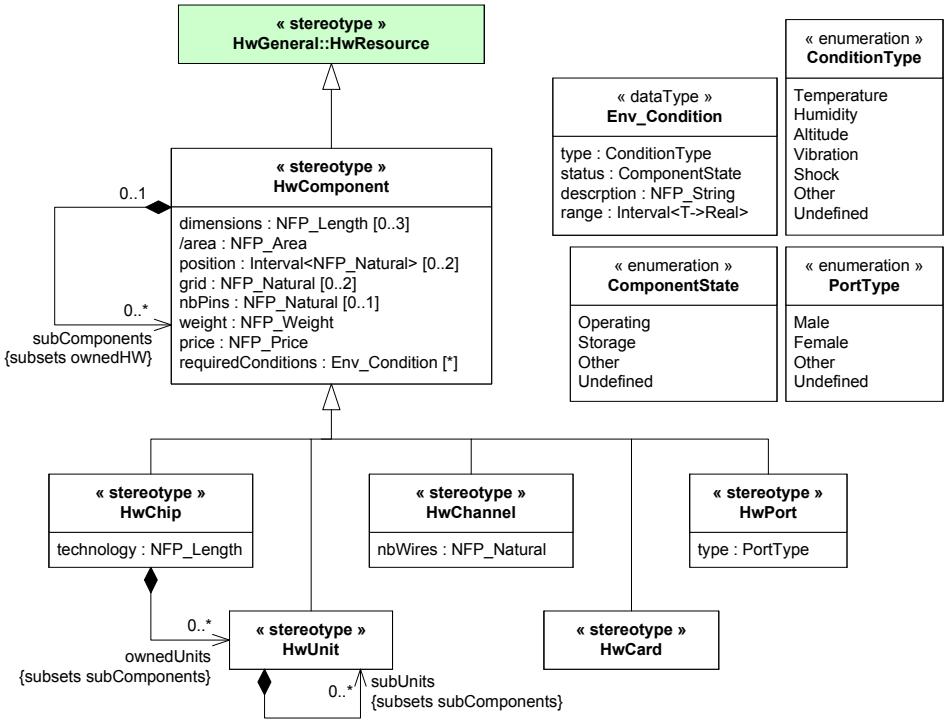


FIG. 3.11 – Le modèle de disposition

Description des concepts

➢ ***ConditionType*** est un type énuméré qui distingue les différents types de conditions environnementales.

➢ ***ComponentState*** est un type énuméré qui distingue l'état du `HwComponent` (fonctionnement ou arrêt) auquel s'applique la `Env_Condition`.

➢ ***Env_Condition*** est un type structuré qui décrit une condition environnementale de sûreté de fonctionnement. Elle se caractérise par un type, un état du HwComponent concerné, une description informelle et un intervalle des valeurs supportées.
Exemple : les plateformes embarquées sont conçues pour être résistantes pendant le fonctionnement comme à l'arrêt, toutefois les conditions d'environnement requises sont généralement moins contraignantes quand le système est à l'arrêt.

➢ ***HwComponent*** est le stéréotype central dans la vue physique. Il représente toute entité physique du matériel. Il peut être basique ou composé de plusieurs `subComponents`, son attribut `dimensions` dénote dans l'ordre la longueur, la largeur et la hauteur du plus petit cube qui enferme le composant, l'attribut `area` est donc dérivé du produit de la longueur et la largeur. On considère ensuite un `HwComponent` comme une grille rectiligne où les `subComponents` occupent des positions, sachant qu'une `position` est un ensemble de rectangles contigus de la grille (voir figure 3.20). Un `HwComponent` a aussi un poids, un prix et il requiert des conditions environnementales pour son bon fonctionnement. Finalement, pour garantir la cohérence du modèle on vérifie récursivement que la somme des valeurs des attributs de taille, de poids... des sous-composants reste inférieure à la valeur du composant contenant.
Exemple : une puce, une plateforme...

➢ ***HwChip*** dénote un circuit intégré ou autrement dit une puce, elle peut être digitale ou analogique et peut contenir plusieurs unités. Son attribut `technology` indique la taille de gravure.
Exemple : un processeur.

➢ ***HwUnit*** est une entité physique qui symbolise une partie identifiée d'une puce ou d'une unité plus grande.
Exemple : l'ALU et la FPU sont des sous-unités de la EU (Execution Unit) qui fait partie de la puce du processeur.

➢ ***HwChannel*** est un ensemble de fils conducteurs qui servent à transporter des données ou de l'électricité.
Exemple : un câble USB est un HwChannel à 4 fils (nbWires).

➢ ***HwCard*** représente un circuit imprimé, Il s'agit typiquement d'un HwComponent composite embarquant des puces et des composants électriques.
Exemple : une carte mère, une barrette mémoire.

➢ ***PortType*** est une type énuméré qui distingue les ports mâles des ports femelles. Notons que ce typage est purement physique, il ne donne aucune indication sur le sens de la communication.

➢ ***HwPort*** est un sous-composant matériel où des équipements externes sont branchés.
Exemple : un port série, un slot mémoire...

1.5.2 HwPower package

Le package `HwPower` établit une description détaillée de la consommation, de la dissipation et de l'émission, qu'on annote aux composants du matériel et aux services qu'ils opèrent. Cette description est nécessaire à l'analyse de la consommation d'énergie et à l'optimisation d'autonomie, et ces étapes sont, à leur tour, essentielles à la conception des systèmes embarqués. Notons que la disposition des composants matériels peut aussi influencer leurs comportements énergétiques.

Métamodèle

Dans la figure 3.12, `HwPowerDescriptor` est la métaclasse clé, elle indique les mesures instantanées du comportement énergétique, il faudra donc les combiner au temps pour obtenir des quantités d'énergie[1]. A chaque service du `HwComponent` (`poweredServices`) on associe une description énergétique particulière (`consumption`) puisque l'activité du composant, et par conséquent sa consommation, dépendent du service en cours d'exécution. Sinon, quand le composant est inactif, il a une consommation statique due aux courants de fuite, à laquelle on associe la description `leakage`.
`HwPowerSupply` et `HwBattery` sont les composants qui fournissent l'énergie nécessaire à la plateforme, pendant que le rôle de `HwCoolingSupply` est d'en réduire la chaleur.

[1]On peut utiliser les diagrammes de séquence d'UML pour mettre en valeur l'exécution des services dans le temps

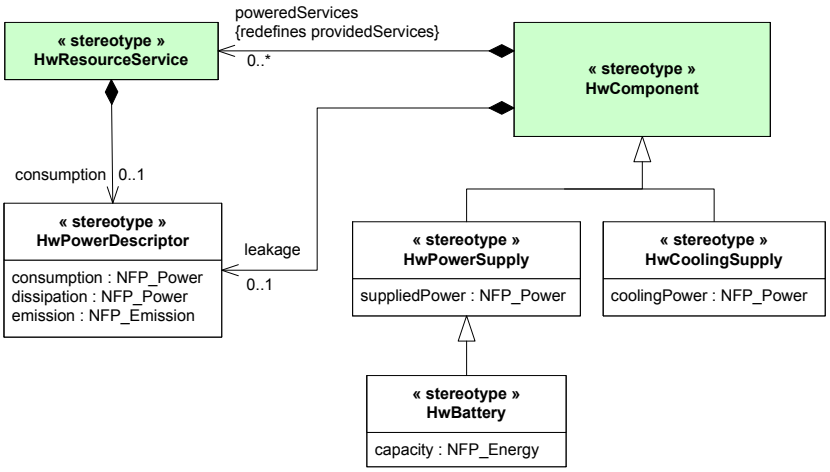


FIG. 3.12 – Le modèle énergitique

Description des concepts

➢ ***PowerDescriptor*** est un dispositif de description énergétique. Il sert à annoter un HwComponent et les services qu'il opère. Il se compose de trois valeurs instantanées concernant la consommation, la dissipation et l'émission. La sémantique de ces valeurs est laissée comme point de variation, l'utilisateur est donc libre de renseigner des valeurs moyennes ou des valeurs maximales.
Exemple : un Intel Pentium cadencé à 200MHz consomme 3W quand il est inactif et jusqu'à 15W à charge maximale.

➢ ***HwPowerSupply*** est une source d'énergie quelconque de la plateforme.
Exemple : une batterie, une cellule photoélectrique…

➢ ***HwCoolingSupply*** est un dissipateur thermique qui sert à refroidir la plateforme pour garder une température adéquate au bon fonctionnement des autres composants.
Exemple : un ventilateur, un radiateur…

➢ ***HwBattery*** dénote une ressource HwPowerSupply non permanente, elle est caractérisée par sa capacité.

1.6 Exemples

Dans cette partie, on présente des exemples élémentaires d'utilisation de HRM. Pour un aspect méthodologique ou des cas d'application plus complets le lecteur devra se référer aux sections 2 et 3 de ce chapitre.

1.6.1 Services des ressources

Comme on a vu précédemment, les services des ressources ne sont pas explicitement spécifiés dans le métamodèle de HRM, même si la taxinomie du modèle logique repose principalement sur les fonctions des ressources et donc sur leurs services fournis. Néanmoins, On peut les déduire de la sémantique et la nature de chaque stéréotype. A titre

d'exemple on fournit les services des ressources de temps définis dans le package HwTiming du modèle logique (voir 1.4.5).

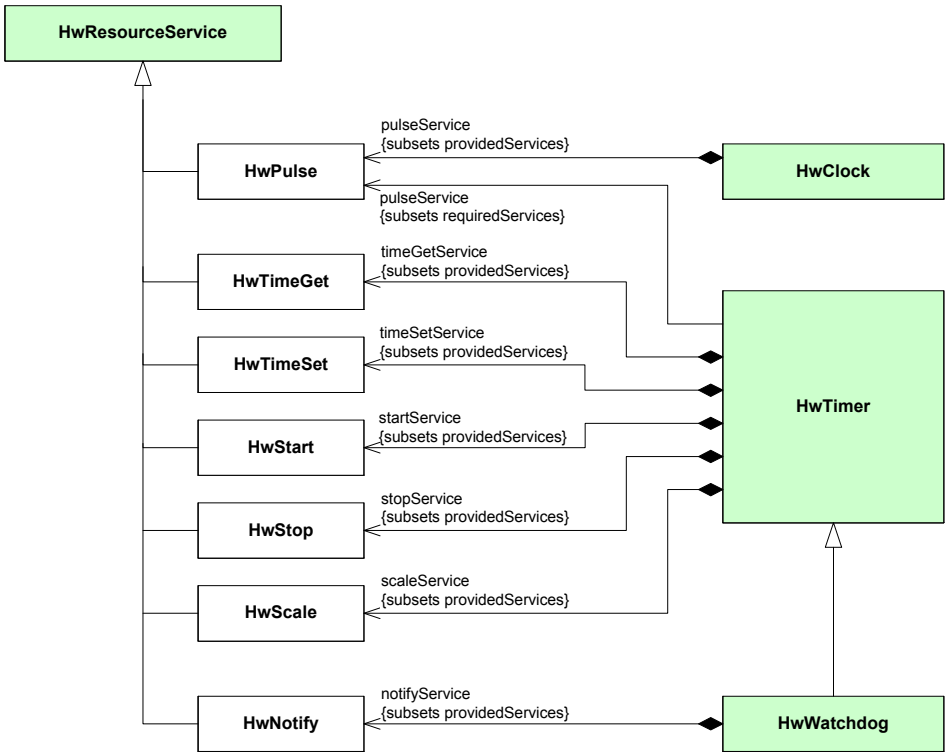


FIG. 3.13 – Les services des ressources de temps

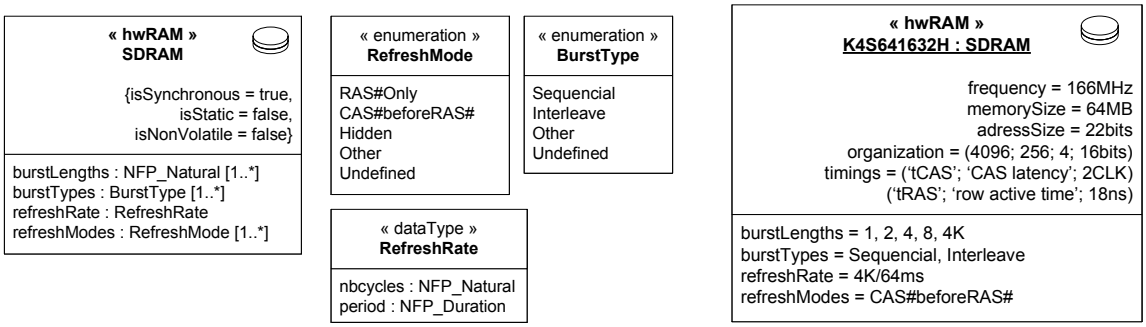
Le modèle de la figure 3.13 est une continuation du métamodèle de temps de la vue logique, il met en valeur les services des ressources de temps. Ces services sont également d'une nature liée au temps, car par exemple, on ne met pas en valeur des services concernant la communication ou l'alimentation. Conformément au modèle général, les services sont des spécialisations du concept `HwResourceService`. `HwClock` fournit un unique service de pulsation, pendant que `HwTimer` fournit plusieurs services de contrôle et de paramétrage du temps dont scaleService qui permet de spécifier le pas d'incrémentation des compteurs en termes de cycles d'horloges, et enfin `HwWatchdog` ajoute aux services de `HwTimer` un service de notification. `HwTimer` requiert le service de pulsation fournit par une `HwClock`, d'où l'association qui parait dans le métamodèle de temps en figure 3.9 qui le lie à une `inputClock`.

1.6.2 Application de stéréotype

Dans le modèle général (figure 3.4), on a étendu parmi d'autres les métaclasses UML Classifier et InstanceSpecification. Le profil HRM permet alors d'appliquer les stéréotypes au niveau classe comme au niveau instance. Prenons en exemple une barrette mémoire de 64Mo SDRAM, On peut donc la modéliser en deux étapes :

– Définir, au niveau modèle, une classe représentant la technologie SDRAM, en appliquant une première fois le stéréotype `HwRAM` et en fixant la partie des *tag values* concernant cette technologie (voir figure 3.14a).

– Instancier la classe SDRAM et appliquer une seconde fois le stéréotype `HwRAM` en fixant d'autres *tags values* concernant cet unité mémoire (voir figure 3.14b)



(a) niveau classe (b) niveau instance

FIG. 3.14 – Application de stéréotype HwRAM

Pour satisfaire les cas d'utilisation de HRM dont la simulation de performance, le stéréotype `HwRAM` a été détaillé. Il permet de spécifier la taille, l'adressage, les timings, la fréquence, l'organisation, les politiques de remplacement et d'écriture et la nature d'une mémoire RAM. Ces caractéristiques sont communes à toutes les technologies RAM, il reste toutefois nécessaire, lors de la modélisation de telles mémoires, d'ajouter leurs propriétés spécifiques après l'application du stéréotype `HwRAM`. Cet exemple illustre ce cas de figure, on y trouve une description du rafraîchissement de la SDRAM qui dépend du cas particulier des mémoires dynamiques (`isStatic = False`).

1.6.3 Modélisation logique/physique

L'exemple qui suit, met en valeur plusieurs aspects de HRM. Il s'agit de modéliser une plateforme matérielle SMP (Symmetric MultiProcessing) avec plusieurs processeurs, où chacun embarque du cache, mais tous partagent la même mémoire principale et un bus système commun. Pour étendre cet exemple, on complète la plateforme SMP avec un DMA et une batterie.

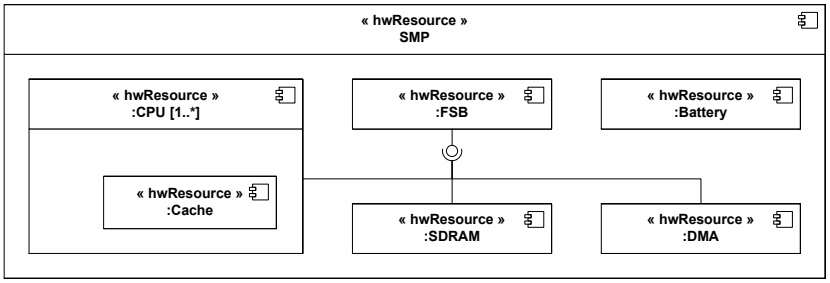


FIG. 3.15 – Vue générale de la plateforme SMP

La figure 3.15 représente une vue générale (voir modèle général en 1.3) d'une plateforme SMP telle elle a été décrite précédemment, et montre à un haut niveau d'abstraction les différentes ressources matérielles, leurs compositions et les services qu'elles échangent. Cette vue est ensuite raffinée séparément en une vue logique (figure 3.16) et une vue physique (figure 3.17).

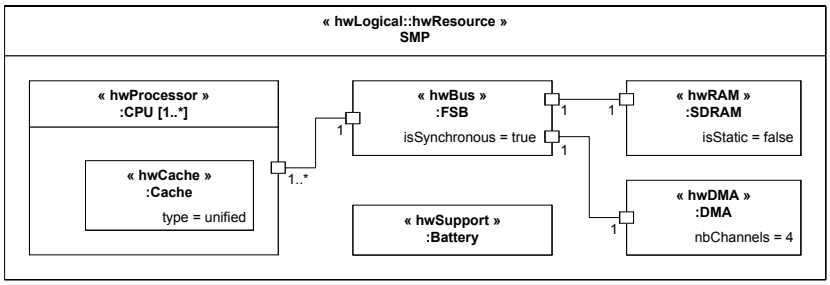


FIG. 3.16 – Vue logique de la plateforme SMP

Il est vrai que HRM peut être utilisé avec tous les diagrammes de structure d'UML comme le diagramme de classe en figure 3.14 ou le diagramme de composant en figure 3.15, cependant le diagramme de composite structure en figure 3.16 reste plus adéquat à la description du matériel, puisqu'il permet de modéliser aussi bien la structure interne des composants et leurs points d'interaction.

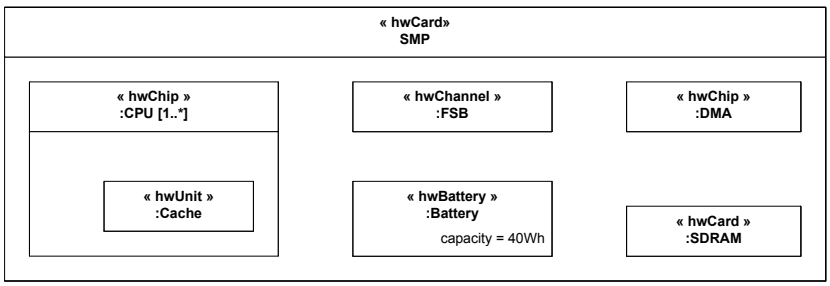


FIG. 3.17 – Vue physique de la plateforme SMP

De par la nature du matériel, l'aspect physique peut influencer l'aspect fonctionnel et vice versa. Les vues logique et physique ne sont donc pas absolument orthogonales et convergent au niveau de plusieurs composants. Autrement dit, il existe des concepts du modèle logique qui correspondent à certains concepts du modèle physique. En effet, dans cet exemple de plateforme SMP, on a `HwBus` qui est typiquement un `HwChannel` et `HwProcessor` qui correspond à un `HwChip`. Réciproquement, le concept physique `HwBattery` est représenté comme un `HwSupport` dans la vue logique. D'autres dépendances au niveau des attributs entre les deux vues existent (voir page 60).

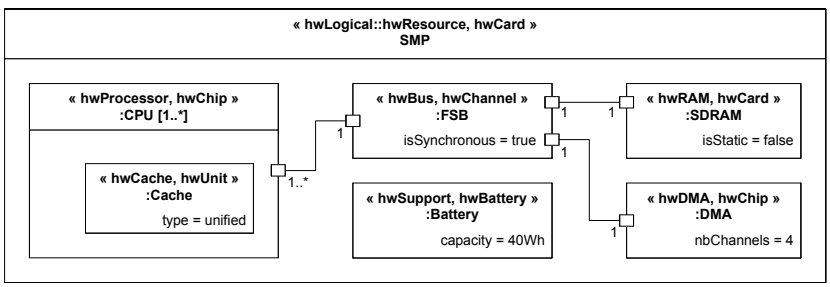


FIG. 3.18 – Vue fusionnée de la plateforme SMP

UML permet l'application de plusieurs stéréotypes sur un même élément, il est alors possible de fusionner les deux vues précédentes logique et physique en une vue unifiée

comme illustré en figure 3.18. Cette possibilité de fusion peut s'avérer pratique, il n'empêche que la séparation des préoccupations reste le meilleur moyen pour remédier à la complexité du matériel et obtenir des vues spécialisées, allégées des données inutiles. Les figures 3.19 et 3.20 sont deux vues détaillées d'une même instance de la plateforme SMP contenant quatre processeurs.

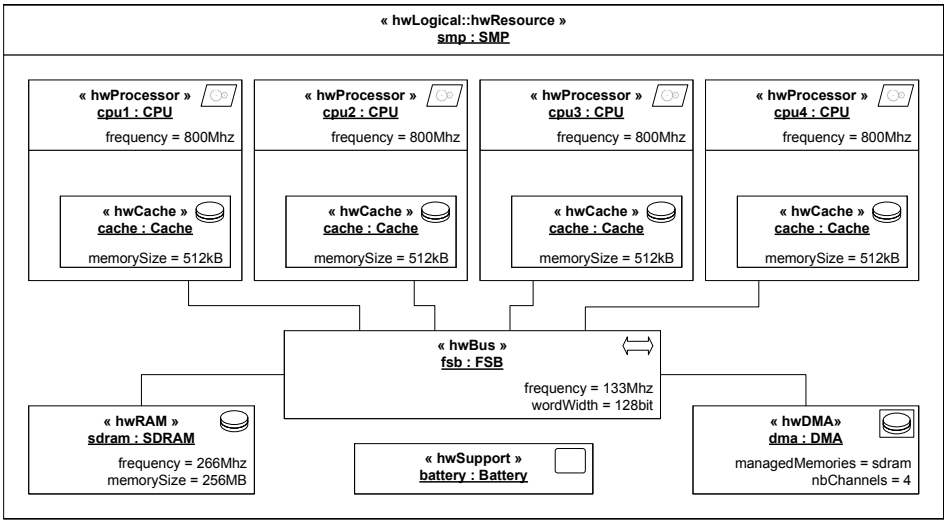


FIG. 3.19 – Vue logique détaillée de la plateforme SMP

le profil HRM comprend plusieurs notations (icônes et formes) et fournit un mécanisme d'arrangement pour approcher les diagrammes d'UML de la disposition réelle des plateformes matérielles.

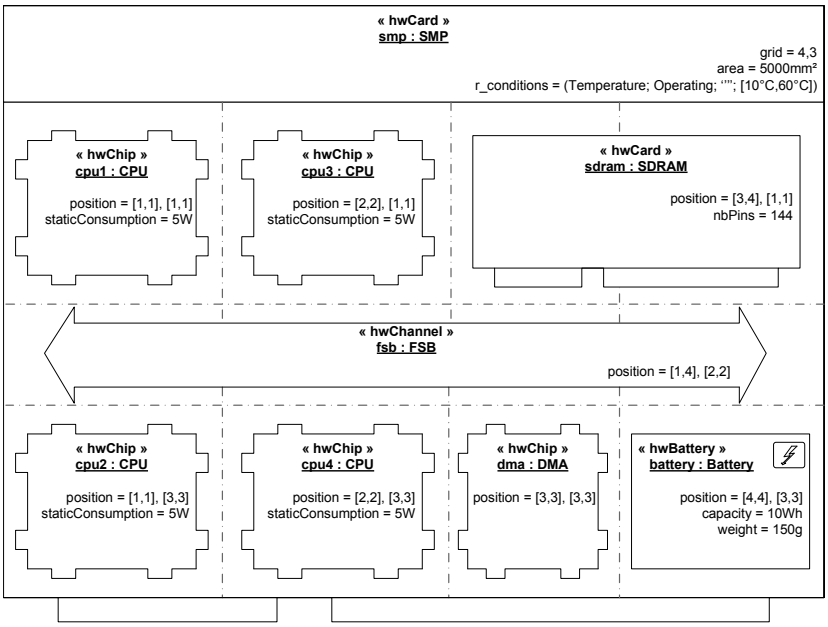


FIG. 3.20 – Vue physique détaillée de la plateforme SMP

Dans cet exemple, la capacité de la batterie au niveau modèle (figure 3.17) est de 40Wh, ce qui correspond à sa capacité maximum de charge. Cette même `tag value` n'est que de

10Wh au niveau instance (figure 3.20) et correspond, cette fois, à la charge courante de la batterie. Cette variation sémantique élargit l'usage de HRM.

1.7 Bilan

HRM est un profil UML pour modéliser le matériel à travers différentes vues et plusieurs niveaux de détail. Il repose sur une séparation en deux sous-profils, un profil logique basé sur les propriétés fonctionnelles du matériel et un profil physique basé sur les propriétés physiques. Ces deux sous-profils partagent une base commune sous forme d'un modèle général similaire au modèle de composants d'UML. Structurellement, HRM est une taxinomie qui regroupe les concepts du matériel en un arbre d'héritages successifs depuis les concepts génériques jusqu'aux concepts spécifiques.

Comme HRM est un nouveau langage de modélisation, cinq questions reviennent régulièrement :

- À quoi sert-il ?
- Est-il correct ?
- Est-il complet ?
- Est-il supporté par des outils ?
- Sur quels exemples a-t-il été testé ?

On répondra séparément à chacune des questions par un ensemble de points.

Usage

- Tout développement conjoint du logiciel et du matériel nécessite un interfaçage entre les deux flots de spécification pour communiquer les décisions de conception, prendre en compte leurs dépendances et préparer de la sorte l'adéquation du déploiement. Idéalement, la communication entre logiciel et matériel s'opère par l'échange de modèles abstraits et compréhensibles des deux bords. Largement adopté parmi la communauté du logiciel, le langage de modélisation UML a, entre autres, cette capacité d'abstraction qui lui permet de décrire les aspects architecturaux et haut niveau d'une application. Le profil HRM étend donc UML de manière à prendre en charge la sémantique du matériel, bénéficier ainsi de ses nombreux mécanismes et faciliter ensuite la communication logiciel/matériel en unifiant le langage de modélisation utilisé (voir chapitre 2).
- Etendre UML pour la modélisation du matériel permet d'exploiter son aptitude à réduire la complexité et séparer les préoccupations en fournissant plusieurs projections d'un même modèle selon les points de vue de la modélisation. Toute analyse qui prend en charge le matériel aura par conséquent en entrée des modèles allégés selon son type d'analyse. La vue logique de HRM servira à l'analyse du WCET, l'analyse de performance, l'analyse d'ordonnançabilité, l'analyse de sureté et tolérance aux pannes, l'analyse de l'utilisation mémoire et toute analyse d'ordre fonctionnel, pendant que celles telles l'analyse de consommation d'énergie, l'analyse des contraintes de dimensions, poids, coût et conditions environnementales se baseront plutôt sur la vue physique de HRM.
- La simulation du matériel est le troisième cas d'utilisation de HRM. En effet, pendant

la conception du profil HRM, on a poussé le niveau de détail dans la description du grand nombre de concepts introduits dans le but de prendre en charge la simulation. Pour ce faire, les stéréotypes de HRM sont annotés de plusieurs propriétés concernant leur fonctionnement interne à l'image de `HwCache`, `HwRAM` ou `HwProcessor`. La simulation peut être considéré comme une implémentation du modèle du matériel, exactement comme des composants physiques peuvent l'être, on obtient donc un simulateur qui, selon la précision de la simulation, a plus ou moins le même comportement que la plateforme physique (voir section 3).

Correction

La notion de correction (*correctness*) d'un langage de modélisation dénote l'exactitude syntaxique et sémantique de ses concepts. Elle garantit de ce fait la cohérence des modèles réalisés.

- HRM fût développé dans le cadre de MARTE et dans l'esprit d'être standardisé. Par conséquent, HRM a été rigoureusement conçu et ne regroupe qu'un minimum de concepts génériques, communs et corrects. Et pour cette raison, on a omis de définir les services de chaque stéréotype dans HRM sachant qu'ils divergent souvent entre ressources de même nature.
- Chaque attribut de stéréotype dans HRM n'a été spécifié que s'il répond à plusieurs critères. Il doit d'abord représenter une propriété caractéristique commune à toutes les ressources désignées par le stéréotype. Ensuite, il doit correspondre au niveau d'abstraction du stéréotype (générique/spécifique) et à la vue en question (logique/physique). Enfin, il doit être nécessaire à au moins un des cas d'utilisation du profil HRM énuméré en début de section.
- Plusieurs contraintes OCL sont annexées au profil HRM pour assurer la cohérence des modèles et sont vérifiées à toute modification du modèle. Elles sont, dans la plupart des cas, locales au stéréotype appliqué astreignant ses *tag values* (voir `CacheStructure` en page 56) ou bien concernant une double application de stéréotypes (voir `HwBus/HwChannel` en page 60). Toutefois, ces contraintes sont difficilement énumérables et seul un usage exhaustif de HRM peut les révéler. Aussi, d'autres contraintes difficiles voire impossibles à exprimer, concernant la consistance de la plateforme modélisée dans sa globalité, restent à la charge de l'utilisateur, par exemple, un bus système est normalement parallèle et synchrone.
- La question de correction peut sous entendre l'exécution des modèles réalisés, dans le sens où l'interprétation du langage est non-ambigue. UML privilégie l'expressivité, il contient de ce fait plusieurs points de variation, cependant, HRM a une sémantique plus précise car il cible un usage spécifique. L'utilisation de HRM dans la simulation du matériel (voir section 3) en est la preuve.

Complétude

On parle de complétude d'un langage de modélisation si, pour toute utilisation, on n'a besoin de rien lui ajouter. Autrement dit, le langage est complet s'il comprend l'ensemble des mécanismes et des concepts nécessaires à l'usage qu'il prétend satisfaire.

- UML est trés expressif et complet, on peut notamment exprimer toute chose de différentes manières, depuis différents points de vue et selon différentes approches. Le profil HRM ne fait qu'étendre UML pour encore plus d'expressivité.
- HRM définit un grand nombre de stéréotypes qui couvrent tous les concepts du matériel et à plusieurs niveaux de détails. Signalons que HRM ne se limite pas à la modélisation de matériel embarqué, il est adapté à la description de tout matériel informatique, comme par exemple des super-calculateurs ou des réseaux distribués.
- HRM hiéarchise ses stéréotypes partant des stéréotypes génériques vers les spécifiques, il supporte ainsi les nouveaux concepts du matériel de nouvelle nature ou issus d'une nouvelle technologie. En effet, si aucun stéréotype spécifique ne correspond à un composant particulier, un stéréotype générique devrait convenir.
- Les héritages entre stéréotypes dans HRM sont complets, c'est-à-dire que les métaclasses filles d'une même métaclasse mère, représentent un partitionnement de l'espace des ressources représentées par cette métaclasse mère. Pareillement les types énumérés définis dans HRM listent, au maximum, tous les literaux éventuels y compris `Other` et `Undefined`.
- Il est vrai que HRM est plutôt adapté à la description d'architectures d'ordinateurs (*computer-based architectures*) à bases de processeurs, de mémoires, de bus, etc... Cependant le concept `HwASIC` dénote une ressource de calcul spécialisée qui peut être spécifiée en joignant un fichier SystemC ou VHDL au modèle ou bien en utilisant d'autres profils à l'image de UML for SystemC et UML for SoC. D'autre part, AUTOSAR fournit un métamodèle dédié à l'automobile et particulièrement détaillé concernant les `HwI/O`(s), il pourra donc être appliqué en plus de HRM en cas de besoin. GASPARD [43] est aussi un profil qui peut être appliqué efficacement (voir chapitre 4 section 4), quand la plateforme matérielle modélisée a une structure répétitive.
- En raison de multiples extensions, les stéréotypes de HRM peuvent être utilisés dans tous les diagrammes UML au niveau classe comme au niveau instance, ce qui élargit leur sémantique. On a aussi introduit dans HRM quelques points de variations sémantiques pour couvrir plusieurs usages, comme au niveau des propriétés quantitatives telle la bande passante d'un HwMedia qui peut signifier une valeur moyenne mesurée ou bien une valeur théorique maximale.

Outillage

- Les modèles UML ont une représentation XML standard et précise grâce au XMI. Dans le cadre du standard MARTE, on fournit librement la description sérialisée en XMI de tous les profils définis, y compris les profils `HwLogical` et `HwPhysical` définis dans la partie HRM. Ils peuvent donc être utilisés avec tous les outils de modélisation UML supportant l'import de profils en XMI.
- La représentation en XMI permet aussi d'exploiter les technologies XML pour diverses manipulations sur les modèles telles l'extraction de données, les transformations de modèles et la génération de code.
- Pour ce travail de thèse, nous avons principalement utilisé Papyrus développé au sein de notre laboratoire (voir section 1.6 du chapitre 2). Papyrus est un plugin Eclipse

complet et libre pour la modélisation en UML2.1. Il prend parfaitement en charge l'application de profils et propose librement un plugin MARTE implémentant le profil.

Tests

Vu l'absence de profils UML pour répondre aux besoins de description haut niveau d'architectures matérielles (voir section 2.3.1 du chapitre 2) et grâce à la standardisation de MARTE, HRM est voué à un usage massif de la part des industriels. Dans le cadre de ce travail, trois exemples d'utilisation exhaustive de HRM sont accomplis :

- La modélisation de la vue logique du microcontrôleur *TC1796* [71] d'*Infineon* en section 2. Il s'agit d'un microcontrôleur performant et complexe embarquant une plateforme d'execution entière sur une même puce (voir figure 3.21). Cet exemple met en valeur la richesse de HRM en concepts (stéréotypes) et le niveau de détail de leur description. Il montre aussi que HRM se prête parfaitement aux aspects méthodologiques.
- La modélisation de toute une librarie de composants matériels simulés par *Simics* [48] en section 3. On peut paramétrer ces composants et les connecter entre eux pour créer des plateformes d'exécution complexes. A l'image de la plateforme illustrée en figure 3.46 qui représente un système distribué, fortement hétérogène, regroupant deux architectures matérielles *PowerPC* et *Itanium*. Cet exemple réaffirme la richesse de HRM en termes de concepts et met en valeur son troisième cas d'utilisation qui est la simulation.
- La modélisation de l'architecture matérielle d'une chenille de robots. Ce cas d'étude développé en section 4 du chapitre 4, nous a permis d'utiliser HRM pour le développement de plateformes matérielles répétitives. Nous avons, pour ce faire, appliquer conjointement les profils HRM et RSM (GASPARD) de MARTE. Cet exemple met en valeur la possibilité d'utiliser HRM avec d'autres profils, notamment RSM pour la modélisation de structures répétitives.

Finalement, comme on répond positivement aux cinq questions, une sixième question est posée :

- Comment l'utiliser ?

On suggère alors la méthodologie décrite dans la section suivante.

2 Méthodologie de modélisation du matériel

UML offre un grand nombre de mécanismes et de concepts sous forme d'une multitude de diagrammes et de notations diverses. Pour maximiser l'expressivité, UML possède aussi plusieurs points de variation sémantique. Il est donc difficile pour un nouvel utilisateur d'UML de s'y retrouver parmi autant de possibilités et d'approches (ex : approche objet, approche composant…). Par conséquent, plusieurs méthodologies de modélisation ont vu le jour. Selon le domaine du logiciel ciblé, elles limitent les mécanismes UML à utiliser, fixent leur sémantique et leur rajoutent des contraintes. A l'image de la méthodologie ACCORD/UML introduite en section 2.4.1 du chapitre 2, qui propose un ensemble de concepts, une sémantique et des règles de conception adaptées au développement des applications temps-réel.

2.1 Vue d'ensemble

Par souci de complétude, les stéréotypes de HRM étendent les métaclasses générales du noyau UML et peuvent donc être appliqués sur la plupart des concepts UML. Il est alors possible d'utiliser HRM dans tout diagramme de structure au niveau classe comme au niveau instance. En considérant que la modélisation en UML avec application de profils n'est pas une pratique courante chez les développeurs du matériel, on propose une méthodologie pour les guider dans la modélisation de plateformes matérielles en utilisant pleinement HRM et en bénéficiant de ses dispositifs. Cette méthodologie est conseillée mais nullement obligatoire ; un utilisateur expert reste libre d'adopter ses propres pratiques ou de n'appliquer que partiellement la méthodologie.

Notre méthodologie est itérative, elle itère **dix étapes de modélisation** et chaque itération permet de modéliser entièrement une ressource de la plateforme matérielle. Elle est également incrémentale et va de bas en haut, elle commence par modéliser les ressources élémentaires et atomiques, et avec des compositions successives, elle finit en un modèle complet de la plateforme. Une incrémentation correspond donc à une composition.

Cette méthodologie emploie principalement le diagramme UML de composite structure [72] car il est adapté à la représentation du matériel pour plusieurs raisons dont les suivantes :

- Il exprime la composition et permet d'arborer et modéliser la structure interne des classes composites. En effet, il décompose un `StructuredClassifier` en un ensemble de `Part`(s) où chaque `Part` est une `Property` typée par un autre `Classifier`.
- Il met en valeur le concept de `Port` qui dénote un point d'interaction d'un `Classifier` qu'on connecte à son environnement extérieur. Ce diagramme permet de la sorte de modéliser une entité indépendamment de son environnement. Notons que les `Port`(s) sont typés et connectés via des `Connector`(s) également typés.
- Il est graphiquement très intuitif et plus proche de la réalité dans sa représentation de la composition, à l'inverse du diagramme de classe qui la représente sous forme d'une `Association`. Il fournit ainsi une vue formelle équivalente au schéma de blocs couramment employé dans le monde du matériel.

Cependant l'introduction du diagramme de composite structure dans UML est récente,

puisqu'il fait partie des nouveaux diagrammes UML de la version 2.0. Par conséquent, ce diagramme est sujet à plusieurs modifications et comprend de nombreux points de variation. Notre méthodologie aura donc pour but d'expliquer d'abord la sémantique des différents concepts du diagramme et de montrer ensuite comment les utiliser dans la modélisation du matériel.

Afin de mettre en valeur l'efficacité de la méthodologie dans la modélisation de plateformes matérielles complexes, nous avons choisi comme exemple d'application, le microcontrôleur *TC1796* du constructeur *Infineon*. Grâce à son architecture *Tricore* [71], *TC1796* est un microcontrôleur de haute performance conçu pour combiner l'exécution d'applications embarquées temps réel et le traitement de signal. Il concentre sur la même puce une grande plateforme d'exécution contenant un *Tricore* CPU, des mémoires de programmes et de données séparées, des bus, des unités d'arbitrage, des Bridges, un contrôleur d'interruption, un DMA et divers périphériques tels des contrôleurs séries, des timers, des convertisseurs Analogic-Digital, plusieurs ports...

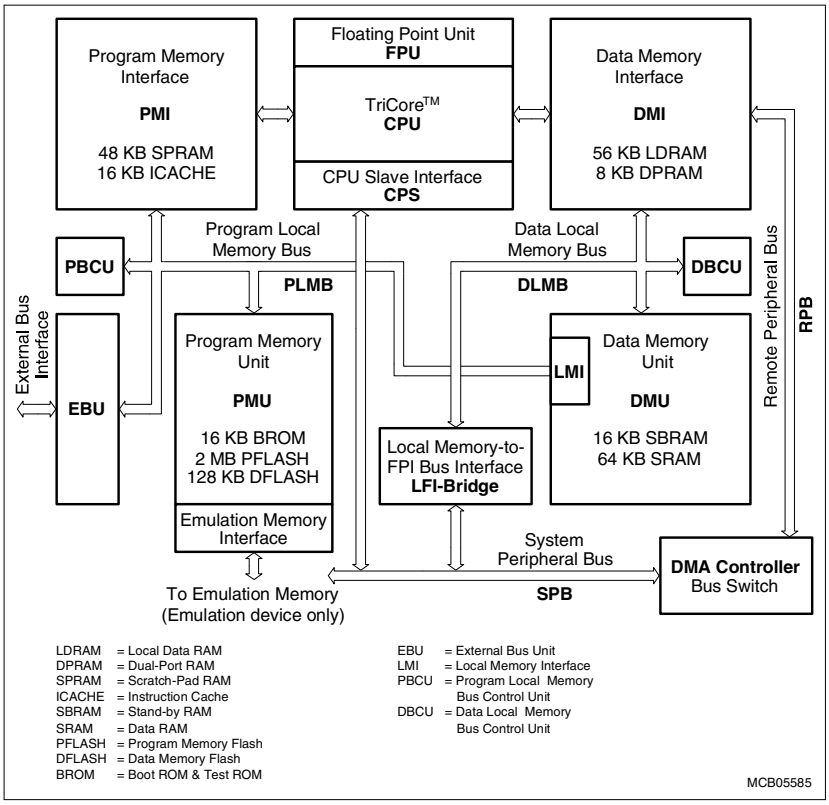


FIG. 3.21 – Sous système de calcul du TC1796

Nous modéliserons principalement une vue logique du sous-système de calcul du TC1796, illustré par un diagramme de blocs en figure 3.21. Ce sous-système est complexe et riche en composants du matériel :

- Un processeur *Tricore* 32-bits, superscalaire à pipeline en 4 étages et cadencé à 150MHz. Il intègre en plus de son architecture RISC (Reduced Instruction Set Computer), une unité DSP (Digital Signal Processor).
- Une architecture mémoire sophistiquée répartie en quatre unités composites avec au total, plus de 2MB de mémoire de programme en Flash, 192KB de mémoire RAM

statique et 16KB de cache d'instruction.
- Une structure de communication complexe à quatre bus, deux bus mémoire 64bits séparant programme et données, un bus 32bits pour joindre les périphériques embarqués et un autre pour les périphériques externes.
- Un DMA à 16 canaux, deux Bridges et deux unités d'arbitrage de bus.

2.2 Etapes de modélisation

On illustre en figure 3.22, par un diagramme d'activité, le séquencement des étapes de modélisation dans notre méthodologie. On constate dix étapes dont l'étape 9 qui renvoie vers l'étape 1 tant que la classe de la plateforme n'est pas atteinte, engendrant ainsi une nouvelle itération. Les étapes 2,3,5,6 et 7 peuvent être sautées sous des conditions qu'on explicitera au cours des différentes étapes. On distingue trois phases, la première pour la définition de la classe de la ressource, la seconde pour la définition de la structure interne de la resource et la troisième pour l'instanciation de la plateforme.

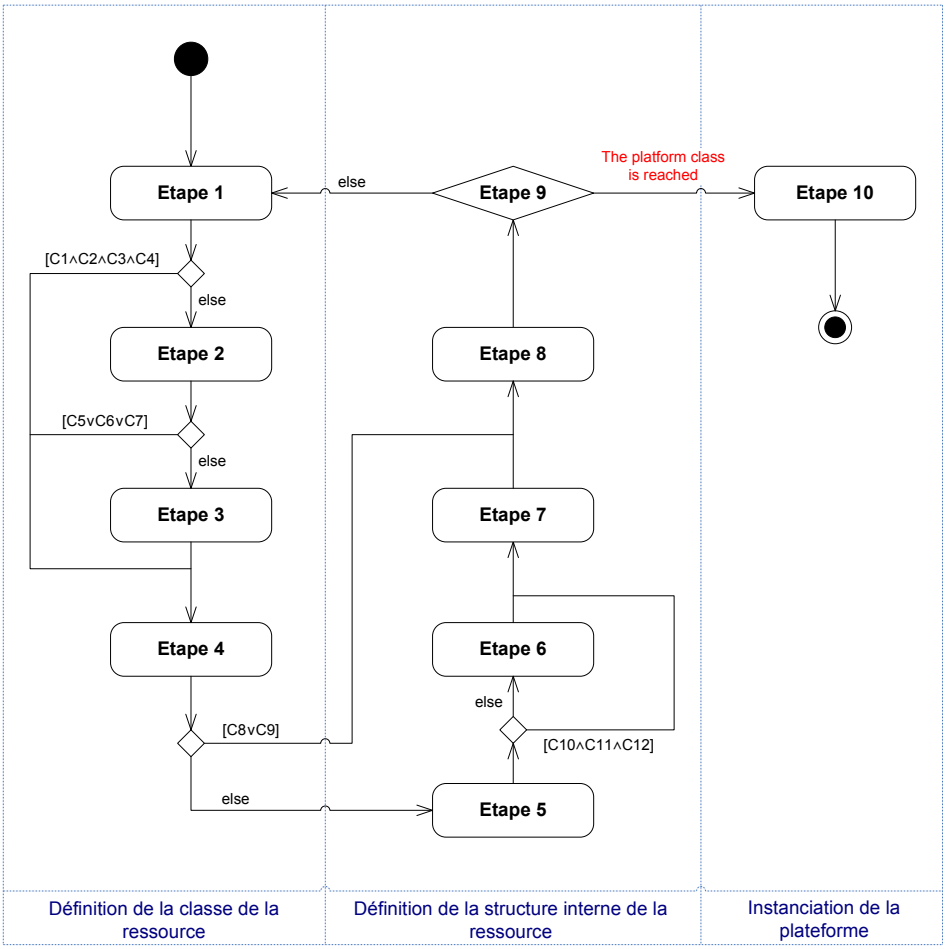


FIG. 3.22 – Séquencement des étapes de modélisation

Brièvement, la méthodologie consiste à (1) définir la classe de la ressource à modéliser, (2) lui appliquer le stéréotype correspondant, (3) fixer une partie des `tag values`, (4) ajouter les associations et les attributs concernant cette classe, ensuite dans un diagramme

de composite structure, (5) définir ses parts et les typer par des classes définies lors de précédentes itérations, (6) leur réappliquer les stéréotypes s'il est nécessaire de fixer une deuxième partie des `tag values`, (7) connecter ces parts entre elles, et puis (8) définir les ports du composant courant et les délégations qui les lient aux ports des parts. Enfin, (9) on itère ce processus jusqu'à atteindre la classe correspondante à la plateforme entière (10) qu'on instancie.

Avant d'entamer notre méthodologie, nous exigeons du lecteur une maîtrise du mécanisme d'extension d'UML via des profils et une connaissance du métamodèle de HRM. Nous l'invitons pour cela à lire respectivement la section 1.4 du chapitre 2 et la section 1 de ce chapitre. Dans la suite nous présenterons notre méthodologie étape par étape et certaines étapes seront à leur tour décomposées en sous-étapes. Pour chaque étape, nous fournirons des exemples de la modélisation du *TC1796* qu'on a réalisée avec l'outil Papyrus.

Etape 1

On entame la première phase concernant la définition de la classe de la ressource.

Choisir la ressource :

Rappelons qu'il s'agit d'une méthodologie incrémentale qui commence par modéliser les composants élémentaires de la plateforme et réitère ensuite selon l'ordre des compositions. On expliquera plus amplement ce processus en étape 9 où l'itération s'effectue.

Dans l'exemple du *TC1796*, on trouve, en premier, en haut à gauche du diagramme de blocs (figure 3.21) l'unité PMI (Program Memory Interface). PMI est une mémoire de calcul (`HwProcessingMemory`) qui contient les instructions du programme en cours d'exécution, elle est cependant composite et on choisit donc de modéliser son plus petit sous-composant ICACHE qui dénote le cache d'instruction. A notre niveau d'abstraction, on considére que la ressource ICACHE est atomique.

Créer la classe :

Une fois qu'on a choisi la ressource à modéliser, on la représente par une classe portant de préférence le même nom.

Dans notre exemple, on crée simplement la classe ICACHE (figure 3.23).

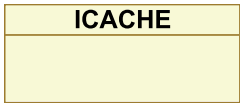


FIG. 3.23 – Classe ICACHE

Définir les héritages :

Il s'agit de définir d'éventuels héritages de la classe courante, dans le cas où elle spécialise des ressources déjà modélisées pendant de précédentes itérations. Notons que l'héritage entre ressources aide à les classifier.

Dans notre exemple, ICACHE est créée en premier et n'a par conséquent pas de classes mères. Cependant les deux bus PLMB et DLMB sont deux spécialisations du bus mémoire LMB de *TriCore*, l'un pour les accès au programme et l'autre pour les

accès aux données. On commence donc par modéliser la classe LMB et aux prochaines itérations, on spécifie PLMB, par exemple, comme classe héritière (figure 3.24).

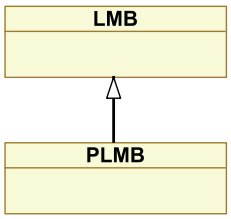


FIG. 3.24 – Classe PLMB

Etape 2

Appliquer le stéréotype :

Selon la nature de la ressource en cours de modélisation, on applique le stéréotype de HRM correspondant. De cette manière, on récupère la sémantique, les mécanismes et la notation du stéréotype HRM appliqué et aussi ceux de ses stéréotypes parents.

Pour ICACHE, on applique le stéréotype `HwCache` qui spécialise `HwMemory` et `HwResource` (figure 3.25).

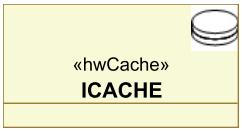


FIG. 3.25 – Classe ICACHE stéréotypée

Appliquer les autres stéréotypes du même profil :

UML laisse à l'utilisateur la possibilité d'appliquer à la même classe plusieurs stéréotypes provenant du même profil (et pas nécessairement liés par héritage). Dans le cadre de la modélisation d'une vue logique avec HRM, il est fréquent qu'une ressource joue plusieurs rôles au sein de la même plateforme. Il est donc pratique de l'annoter de différents stéréotypes correspondant aux différentes fonctionnalités, sans être obligé de la décomposer en plusieurs ressources.

Revenons à notre exemple *TC1796*, on y trouve le composant EBU (External Bus Unit) qui en même temps, connecte le bus mémoire externe EMB (External Memory Bus) au PLMB et opère l'arbitrage entre composants maîtres du EMB. Par conséquent, on lui appliquera les stéréotypes `HwBridge` et `HwArbiter`. Normalement l'ordre est facultatif, mais graphiquement, les outils n'affichent que l'icône du premier stéréotype appliqué (figure 3.26 vs figure 3.30).

Appliquer les autres stéréotypes de profils différents :

UML permet aussi d'appliquer plusieurs profils au même modèle, et d'annoter ensuite une même classe de plusieurs stéréotypes provenant de profils différents. Dans le cas de HRM, il est possible de combiner les stéréotypes du profil logique et ceux du profil physique comme on l'a montré précédemment dans la figure 3.18 de l'exemple 1.6.3.

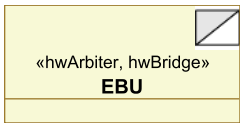


FIG. 3.26 – Classe EBU multi-stéréotypée

Sauter l'étape :

Même si l'étape d'application de stéréotypes est importante pour la prise en charge du matériel dans la modélisation UML, elle pourrait potentiellement être ignorée si toutes les conditions suivantes sont remplies :

[C1] Les classes parents de la classe en cours de modélisation appliquent les mêmes stéréotypes. Plus précisément, pour chaque stéréotype à appliquer, il existe au moins une des classes parents qui l'applique déjà.

[C2] Les valeurs des `tag definitions` précédemment annotées sur les classes parents coïncident. On n'a donc pas besoin de les surcharger en annotant de nouvelles `tag values` après application des mêmes stéréotypes.

[C3] Nul besoin de spécifier des `tag values` supplémentaires et absentes des classes parents.

[C4] Nul besoin des notations qu'implique l'application de stéréotypes. En effet, pour améliorer l'aspect graphique du diagramme UML, il est parfois utile de profiter des icônes et des formes définies dans le profil HRM.

En figure 3.28, lors de la modélisation de PLMB (ou DLMB) après une définition exhaustive de la classe LMB (stéréotypage et paramétrage), seul l'aspect graphique peut justifier la réapplication du stéréotype `HwBus`.

Etape 3

Paramétrer les stéréotypes :

Une fois les stéréotypes appliqués, il est important de les paramétrer. Chaque stéréotype a une sémantique qui dépend des valeurs de ses `tag definitions`, autrement dit, les valeurs de ses propriétés (attributs). Dans HRM, les propriétés des stéréotypes sont toutes optionnelles, et l'utilisateur ne doit les paramétrer qu'en cas de besoin. Pendant cette étape de modélisation, l'utilisateur ne doit spécifier que les propriétés qui relèvent d'un aspect technologique, c'est à dire qui caractérisent globalement la classe modélisée et qui sont communes à l'ensemble des ressources représentées. Il pourra spécifier par la suite et notamment en étape 6 et étape 10 les propriétés relevant du niveau instance. L'utilisateur doit également prendre en compte le niveau de détail où il se place dans ce paramétrage.

Dans notre exemple, après l'application du stéréotype *HwCache* à la classe ICACHE, on fixe les `tag values` relevant du niveau classe comme illustré en figure 3.27. On constate que seules les propriétés caractéristiques d'un cache d'instruction *TriCore* sont définies, tel son niveau, sa largeur d'adresse, son associativité et ses politiques. Par contre, ni la taille, ni la fréquence ne sont définies à cette étape.

Sauter l'étape :

cette étape peut également être ignorée dans l'un des cas suivants :

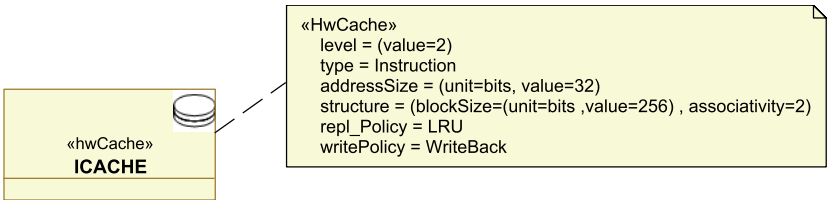


FIG. 3.27 – Classe ICACHE stéréotypée et paramétrée

[C5] L'étape 2 est auparavant sautée, c'est-à-dire qu'il n'y a aucune application de stéréotypes.

[C6] Le stéréotype en question n'a pas de tag definitions. Notons que ce cas de figure ne s'applique pas au profil HRM, car même si plusieurs concepts n'ont pas leurs propres tag définitions à l'exemple de `HwDevice` et `HwASIC`, ils héritent au moins de celles de `HwResource` induites des associations avec `HwResourceService` (voir modèle général 1.3). Cependant de tels stéréotypes existent dans le profil MARTE comme le stéréotype `NFP`.

[C7] Les valeurs par défaut des `tag definitions` ou celles attribuées par les classes parents pareillement stéréotypées, sont égales à celles qu'on cherche à spécifier.

Revenons à notre cas d'exemple, vu que les deux bus PLMB et DLMB sont identiques, on choisit de paramétrer le concept général LMB plutôt que de dupliquer les annotations pour chacun des bus (figure 3.8).

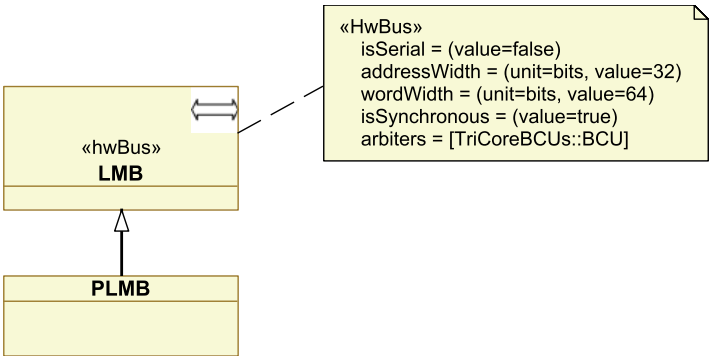


FIG. 3.28 – Classe PLMB non stéréotypée

Etape 4

Même si HRM offre un grand niveau de détail dans la description du matériel, il est avant tout un standard, ne regroupant que les concepts et les mécanismes génériques et communs. Par conséquent, si à ce stade de modélisation, l'utilisateur a toujours besoin de spécifier des données supplémentaires concernant la classe courante, il devra utiliser UML de manière ordinaire indépendamment de HRM.

Définir les attributs :

En complément à HRM, il s'agit d'ajouter à la classe courante les propriétés qu'on ne peut exprimer avec l'ensemble des annotations du profil HRM. Ces propriétés doivent

être rigoureusement typées, et pour ce faire, on pourra exploiter les mécanismes de typage UML en définissant des types structurés (`DataType`) ou des types énumérés (`Enumeration`). On pourra également exploiter la librairie MARTE de types basiques BasicNFP_Types ou définir de nouveaux types complexes grâce au profil NFP.
L'exemple 1.6.2 de modélisation d'une SDRAM illustre l'intérêt de cette étape, puisqu'on y décrit par le biais de quatre attributs, les modes de communication et les modes de rafraichissement de cette mémoire dynamique indépendamment de HRM. Pour cela, on définit deux types énumérés `RefreshMode` et `BurstType`, un type structuré `RefreshRate` et on se sert de la librarie BasicNFP_Types.

Définir les associations :

Dans le cadre de la modélisation du matériel, l'agrégation, n'a pas, à notre sens, une sémantique cohérente, car dans le monde réel, on ne peut appartenir physiquement à deux entités disjointes. Toutefois si l'utilisateur requiert ce concept, il lui suffit de spécifier en premier sa sémantique. D'un autre côté, on préfère exprimer la composition en étape 5 grâce aux mécanismes du diagramme de composite structure lesquels sont plus lisibles. Il reste donc à l'utilisateur de spécifier, pendant cette sous-étape, les associations «ordinaires» pour exprimer les rapports entre ressources.
Une association entre entités matérielles (`HwResource`) dénote généralement d'éventuels échanges de services (`HwResourceService`). Ces services peuvent être unidirectionnels comme ils peuvent être bidirectionnels. Dans le profil HRM plusieurs associations sont déjà définies au niveau métamodèle. L'utilisateur devra vérifier d'abord si l'association qu'il cherche à exprimer existe au niveau métamodèle, et le cas échéant, il devra utiliser de préférence les `tag definitions` induites de cette méta-association sans surcharger le modèle.
On illustre ce point dans la modélisation du bus LMB en figure 3.28. Parmi les `tag definitions`, `arbiters` est induite de la méta-association qui lie un `HwMedia` et un `HwArbiter` (voir figure 3.8). Dans notre cas, un bus LMB est arbitré par une unité BCU (Bus Control Unit) logiquement stéréotypée par `HwArbiter`. En utilisant ainsi le profil HRM on n'a nul besoin de redéfinir, au niveau modèle, l'association qui lie un bus à son arbitre.
Quand l'association ne figure pas dans le profil HRM, l'utilisateur doit la définir de manière ordinaire et avec la syntaxe UML. Toutefois si on considère une association comme un échange de services, qui ne peut se réaliser dans la pratique que par une connexion physique entre les ressources associées, il est dès lors permis à l'utilisateur de considérer une association comme un lien de communication entre classes de ressources. Il pourra ainsi la stéréotyper, si besoin, par `HwMedia` ou l'un de ses stéréotypes fils (`HwBus` et `HwBridge`) pour joindre à l'association une sémantique de communication.
Revenons à notre classe courante ICACHE, elle est associée à une TagRAM (voir figure 3.29) à l'intérieur de la mémoire PMI. TagRAM est une mémoire statique qui permet de mémoriser les adresses des blocs de mémoire stockés en cache.

Définir les opérations :

Cette sous-étape sert à décrire la structure des services requis ou fournis par la ressource en cours de modélisation. Il s'agit de définir, si nécessaire, un ensemble d'opérations à la classe, et de les stéréotyper en `HwResourceService`(s) pour les

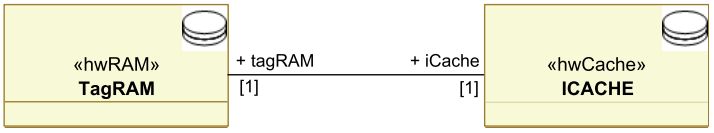


FIG. 3.29 – Association entre ICACHE et TagRAM

déclarer comme étant des services du matériel. Ensuite, on référence chaque opération dans la `tag definition providedServices` du stéréotype correspondant et appliqué à la classe courante, on confirme ainsi que le service est fourni aux autres ressources de la plateforme et à quel titre. En effet, dans le cas où la classe applique plusieurs stéréotypes, il faut lier l'opération, selon sa nature, au stéréotype adéquat.
Dans une deuxième étape, il fautdra spécifier les opérations requises `requiredServices` pour chaque stéréotype appliqué, en référençant des opérations d'autres classes de la plateforme précédemment définies. On vérifie également quelques règles de cohérence pour chaque opération ajoutée :

– L' opération doit être stéréotypée par `HwResourceService`.
– L'opération doit être référencée en `providedServices` dans l'un des stéréotypes appliqués à la classe source.
– Il existe une association (ou méta-association) qui lie la classe qui requiert le service à celle qui le fournit.

Dans la figure 3.30, on reprend l'exemple du composant EBU qui occupe deux fonctions dans *TC1796*. D'un côté, EBU fait le pont entre le bus mémoire local et le bus mémoire externe, il applique alors le stéréotype `HwBridge`, il est méta-associé de chaque coté (`sides`) aux bus PLMB et EMB et il fournit le service matériel de traduction de protocole et de données (`translate`). De l'autre côté, EBU arbitre le bus mémoire externe EMB auquel il est méta-associé (`controlledMedias`), il fournit à ce titre un service d'acquisition de bus (`holdExtBus`) aux maitres qui y sont connectés et leur exige sa libération (`requestExtBus`) si nécessaire.

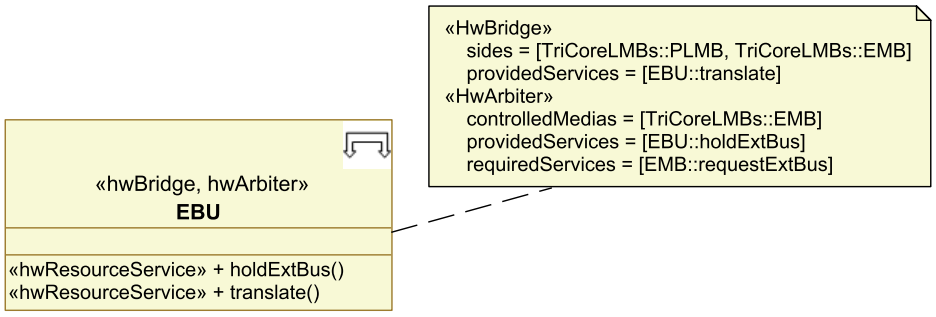


FIG. 3.30 – Classe EBU stéréotypée et paramétrée

Etape 5

On entame la seconde phase concernant la définition de la structure interne de la ressource courante. Avant cette étape l'utilisateur est libre de choisir le diagramme de structure de

son choix puisqu'il fait usage d'extensions HRM applicables à des concepts communs aux diagrammes de classe, de composant, de composite structure, de déploiement, etc. Dans notre cas, nous avons simplement utilisé le diagramme de classe. Mais dorénavant, c'est-à-dire dans les étapes suivantes jusqu'à la prochaine itération, seul le diagramme de composite structure doit être utilisé.

Définir les sous-composants :

On définit dans cette étape la structure interne de la ressource en cours de modélisation. On y insère des `Part`(s) UML typées par des classes spécifiées lors des précédentes itérations. On voit ainsi l'intérêt dans notre méthodologie de suivre un ordre incrémental basé sur des compositions successives. Chaque `Part` porte un nom facultatif généré automatiquement si l'utilisateur ne le spécifie pas, et une multiplicité qui est un mécanisme simple et puissant pour la représentation de structures répétitives, trés fréquentes dans le domaine du matériel. Chaque `Part` doit également afficher ses ports, qui correspondent à ceux de sa classe type (voir étape 8). Comme ICACHE est une ressource élémentaire, prenons en exemple son conteneur, le composant PMI (Program Memory Interface). PMI est composite et contient en plus du ICACHE, une TagRAM, une SPRAM et une DCU (Data Control Unit). La DCU est une ressource de contrôle et d'interfaçage de mémoires *TriCore* qu'on retrouve également dans la DMI. La clarté de lafigure 3.31 illustre l'intérêt d'utiliser le diagramme de composite structure.

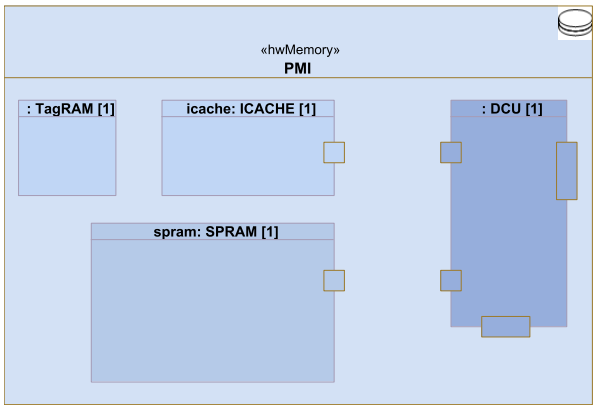


FIG. 3.31 – Classe composite PMI : Structure interne

Dans le métamodèle général de HRM (voir figure 3.4), on a une méta-association de type composition qui lie chaque ressource à ses sous-éléments avec le rôle `ownedHW`. En plus, plusieurs méta-associations dans le reste du métamodèle HRM en définissent des sous-ensembles grâce notamment au dispositif `subsets` d'UML. Signalons que ces rôles qu'on retrouve comme `tag definitions` en tout stéréotype HRM, ne doivent pas être spécifiés dans l'étape 3 pendant le paramétrage au niveau classe, puisqu'ils sont mieux représentés dans cette étape grâce au mécanisme de composition du diagramme de composite structure d'UML.

Sauter l'étape :

L'étape 5 ainsi que l'étape 6 et l'étape 7 sont trois étapes réservées à la description de la structure interne de la classe courante. Elles sont donc sautées seulement si :

[C8] La classe courante représente une ressource basique de la plateforme et est considérée comme atomique (non composite).

[C9] La structure interne de la classe courante est identique à celle de ses classes parents et elle a donc été décrite lors des précédentes itérations.

Tel est le cas de la classe ICACHE modélisée dans notre exemple pendant la première itération.

Etape 6

Réappliquer les stéréotypes :

Une fois les `Part`(s) définies et typées par les classes correspondantes et en prenant en considération leur nouveau contexte, il est important de leur réappliquer des stéréotypes afin de prendre en charge leurs nouvelles caractéristiques matérielles. En effet, si on a modélisé précédemment une ressource dans l'absolu indépendamment de son cadre, c'est-à-dire en dehors du composant qui l'enveloppe, et qu'on la retrouve maintenant comme `Part` au sein d'une ressource composite, on doit pouvoir paramétrer plus amplement et plus précisément ses `tag definitions`. Cela est rendu possible grâce au fait que dans UML une `Part` est une `Property` et que les concepts HRM étendent aussi cette métaclasse.

Même si l'outillage n'interdit rien, l'utilisateur devrait se limiter à la réapplication de stéréotypes précédemment appliqués à la classe type ou l'une de ses classe parents. Il s'agit d'une règle de consistance entre la nature de la classe et les rôles qu'elle peut avoir dans différents environnements.

Reprenons la classe PMI, la figure 3.32 met en valeur une réapplication de stéréotypes concernant les `Part`(s) de type ICACHE, SPRAM et DCU, tandis que la `Part` de TagRAM n'est pas re-stéréotypée pour les raisons exposées ci-dessous.

Paramétrer les stéréotypes :

Cette sous-étape permet de paramétrer les stéréotypes appliqués à chaque `Part` en surchargeant les valeurs de `tag definitions` spécifiées au niveau de la classe type ou l'une de ses classes parents, ou bien en annotant de nouvelles `tag definitions` spécifiques au nouvel environnement de la ressource en question.

Toujours concernant la modélisation de la structure interne de la PMI, on voit dans la figure 3.32, le paramétrage de la taille mémoire de ICACHE et de SPRAM. Car si ces dernières ont été modélisées au niveau classe en ne mettant en valeur que leurs aspects technologiques, on connaît maintenant et de façon exacte les quantités de mémoire comprises dans la ressource PMI. De même, si la DCU est une unité de contrôle mémoire générique, son usage au coeur d'une PMI permet d'interfacer les deux mémoires particulières icache et spram.

Sauter l'étape :

De façon similaire à l'étape 2, cette étape ne doit être ignorée que sous trois conditions vérifiées dans l'ordre :

[C10] Les valeurs des `tag definitions` précédemment annotées au niveau classe restent correctes. On n'a donc pas besoin de les surcharger en annotant de nouvelles `tag values` au niveau de la `Part`.

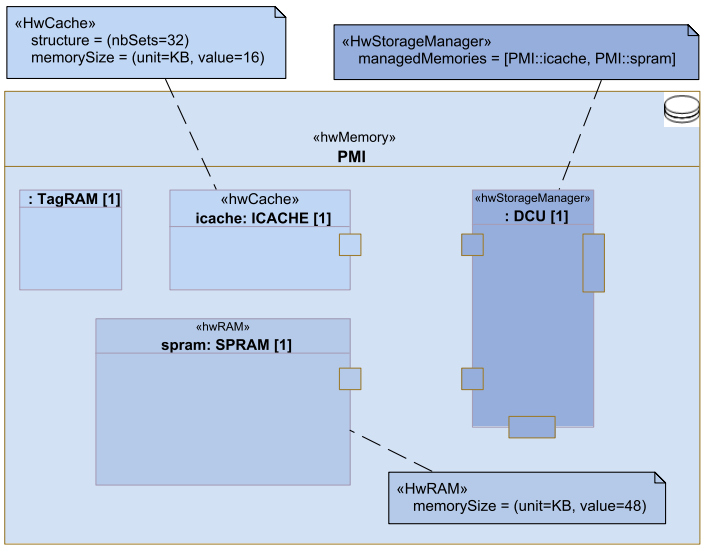


FIG. 3.32 – Classe composite PMI : Stéréotypage et paramétrage

[C11] Nul besoin de spécifier des `tag values` supplémentaires et absentes au niveau classe.

[C12] Nul besoin des notations graphiques qui accompagnent l'application de stéréotypes HRM.

La `Part TagRAM` de la PMI remplit ces trois conditions et n'est donc pas re-stéréotypée.

Etape 7

Globalement, pendant l'étape 7, on cherche à connecter entre elles les `Part`(s) appartenant à la structure interne de la classe courante. On exprime ainsi un échange de services et/ou une communication de données.

Connecter les sous-composants associés :

Rappelons que pendant l'étape 4, nous avons modélisé des associations entre classes du matériel, pour exprimer un éventuel échange de services entre ressources, non pris en charge par le profil HRM. Par conséquent, si à l'intérieur de la classe en cours de modélisation, on retrouve des `Part`(s) typées par des classes associées lors de l'étape 4, et si cette association prend sens dans le contexte de la classe courante, on les connecte par un `Connector` UML qu'on type par l'association en question. Ce Connecteur peut lier les sous-composants directement ou bien les lier à travers leurs ports si ces derniers ont une existence matérielle.

Jusqu'ici UML est utilisé de manière ordinaire sans employer le profil HRM, mais si l'utilisateur juge utile d'appliquer ou de réappliquer (voir étape 4) les stéréotypes `HwMedia`, `HwBus` ou `HwBridge` pour décrire sommairement une communication entre ces ressources, HRM lui offre cette possibilité en étendant `Connector` (voir modèle de communication en figure 3.8).

Dans la figure 3.33, on retrouve un connecteur entre les `Part`(s) représentant une ICACHE et une TagRAM typé par l'association qu'on a modélisée précédemment dans l'étape 4 (voir figure 3.29). Ici, l'usage de ports n'est pas justifié du point de vue

matériel puisque dans la PMI, la TagRAM et le ICACHE fusionnent physiquement dans une même unité et n'échangent point leurs services à travers des ports.

Connecter les sous-composants méta-associés :

Cette seconde sous-étape est réservée à la connexion des sous-composants liés, de par leurs fonctionnalités et les services qu'ils échangent, par des méta-associations définies dans HRM. A l'exemple de celle qui lie une `HwResource` et une `HwClock` dans le métamodèle `HwTiming` (voir figure 3.9) ou celle qui lie un `HwBus` à un `HwArbiter` dans le métamodèle `HwCommunication` (voir figure 3.8). De façon similaire à la sous-étape précédente, l'utilisateur doit modéliser cela via des `Connector`(s) en passant ou pas par des ports et en appliquant si nécessaire un des stéréotypes de communication énumérés.

On connecte, dans l'exemple de la figure 3.33 illustrant la classe composite PMI, le contrôleur mémoire de type DCU aux sous-composants icache et spram via deux connecteurs. Ces connecteurs correspondent à la méta-association qui lie un `HwStorageManager` à ses `managedMemories` dans le métamodèle `HwMemory` (voir figure 3.7). On connecte cette fois les ports adéquats (voir étape 8) et on applique le stéréotype `HwBus` pour exprimer un échange de mots en `128bits` de largeur.

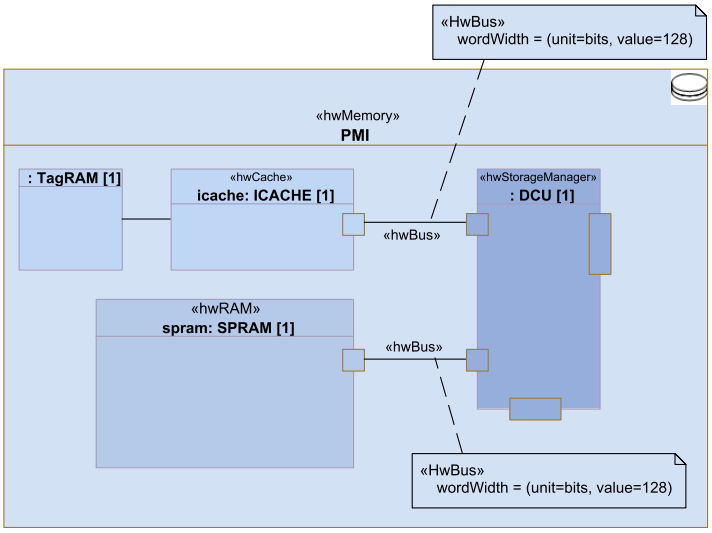


FIG. 3.33 – Classe composite PMI : Connexions

Ouvrons ici une parenthèse concernant le système d'extension UML via des profils. Comme il a été discuté pendant le positionnement de ces travaux en section 1.4 du chapitre 2, il est vrai que les méta-associations entre métaclasses d'un métamodèle sont transformées en `tag definitions` au niveau des stéréotypes du profil. En effet, dans le cas de HRM, on retrouve ces méta-associations sous forme de `tag definitions` que l'utilisateur est normalement supposé spécifier pendant l'étape 3 et l'étape 6.

Dans l'exemple du LMB en figure 3.28 (étape 3) ou l'exemple de la PMI en figure 3.32 (étape 6), on voit que les tag definitions (`arbiters`, `managedMemories`) générées depuis des méta-associations de HRM, sont déjà spécifiées.

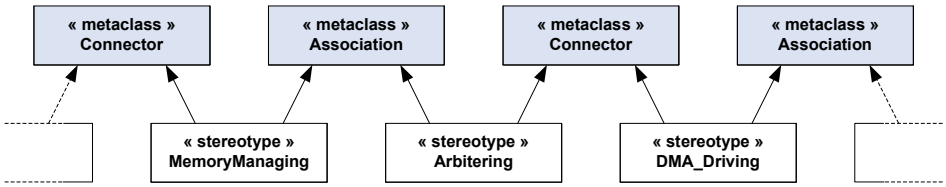
On en déduit qu'utiliser les connecteurs à cette étape induit une redondance au niveau

du modèle. Cependant si on prend en compte la clarté des modèles et la facilité d'utilisation de HRM, il est évident que les connecteurs sont des mécanismes plus intuitifs, plus proches de la réalité et plus faciles à utliser. On aimerait donc les employer comme alternative aux `tag definitions` générées. Malheureusement, comme on ne peut typer un `Connector` par une méta-association du profil HRM et qu'il est de ce fait impossible de connaître quelle méta-association est représentée par un connecteur, les connecteurs restent insuffisants sémantiquement.
Pour remédier à cette insuffisance sémantique nous proposons deux solutions techniques quasi-équivalentes afin d'étendre le concept UML de `Connector` et prendre en charge la sémantique des méta-associations de HRM.

Solution 1

La première solution, illustrée en figure 3.34a, consiste à définir un stéréotype pour chaque méta-association de HRM et que chaque stéréotype étend les métaclasses UML `Connector` et `Association`. A l'utilisation et comme le montre la figure 3.34b, il suffit, lors de cette sous-étape, d'appliquer le stéréotype adéquat à chaque connecteur (de méta-association) afin d'exprimer le type d'échange de services opéré. La spécification des `tag definitions` correspondant aux méta-associations devient ainsi optionnelle.

(a) Stéréotypes et extensions UML



(b) Exemple d'utilisation

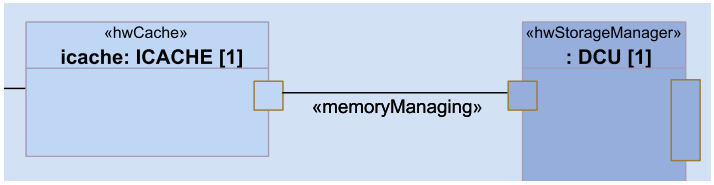
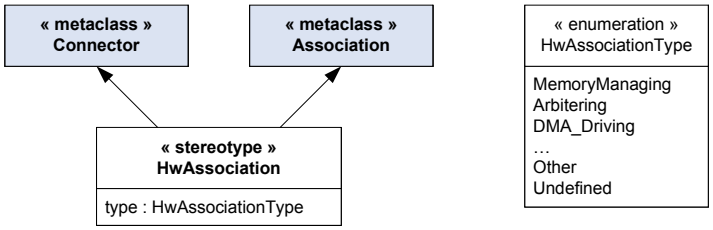


FIG. 3.34 – Première solution

Solution 2

La seconde solution ne diffère que techniquement de la première (voir figure 3.35a), il s'agit de définir un seul stéréotype `HwAssociation` qui dénote un échange de service du point de vue matériel et qui est paramétré par un type énuméré `HwAssociationType` qui regroupe la liste des méta-associations de HRM. Notons que dans HRM, mis à part les compositions, on énumère seulement huit méta-associations. A l'utilisation nous appliquerons le stéréotype à tout connecteur (de méta-association), puis on précisera son type. L'exemple de la connexion entre les sous-composants ICACHE et DCU de la ressource PMI est fourni en figure 3.35b.

(a) Stéréotype, énumération et extensions UML



(b) Exemple d'utilisation

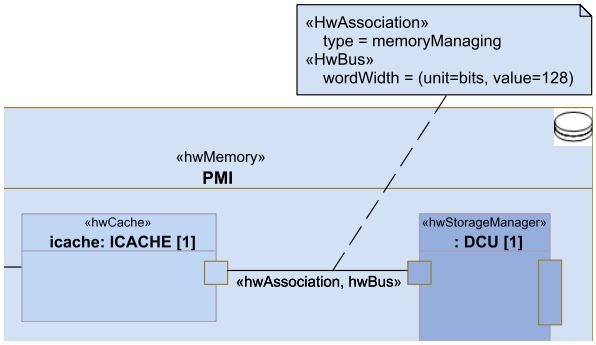


FIG. 3.35 – Seconde solution

Connecter les sous-composants communiquants :

Contrairement aux deux sous-étapes précédentes qui modélisent des connexions correspondant aux divers échanges de services tel l'arbitrage, la gestion mémoire, etc, cette sous-étape est consacrée à la modélisation des communications de données entre les `Part`(s) de la ressource composite courante. Ces communications devront être conformes au métamodèle `HwCommunication` de HRM (défini en figure 3.8). Elles devront d'abord lier des ports stéréotypés en `HwEndPoint` (voir étape 8), ensuite on distingue deux cas de figure, des communications point à point illustrées en figure 3.36a, et des communications multipoints illustrées en figure 3.36b. Dans le premier cas, on utilise un `Connector` UML qu'on stéréotype par `HwMedia` ou l'un de ses stéréotypes fils `HwBus` et `HwBridge` et qu'on connecte de chaque coté aux ports communiquants. Cependant dans le second cas, on a un sous-composant (`Part`) qui sert de moyen de communication et qui est stéréotypé par `HwMedia` (ou `HwBus` ou `HwBridge`), auquel on connecte les ports communiquants via des connecteurs non nécessairement stéréotypés cette fois. Aussi, dans le second cas de figure, il faudra impérativement vérifier que la connexion est conforme à la `tag value` de `connectedTo` du stéréotype `HwEndpoint` appliqué au port communiquant.

Dans le cadre de l'exemple `TC1796` illustré entièrement en figure 3.39, on constate l'existence des deux types de connexions développés dans cette sous-étape. Deux communications multipoints autour de PLMB et DLMB et plusieurs communications point à point dont celle entre le CPU et la PMI et entre le CPU et la DMI.

Récapitulons l'étape 7, il s'agit de connecter les sous-composants de la classe courante via des connecteurs :

– typés s'ils dénotent une association du modèle
– méta-typés (c-à-d stéréotypés) s'ils dénotent un échange de services de HRM

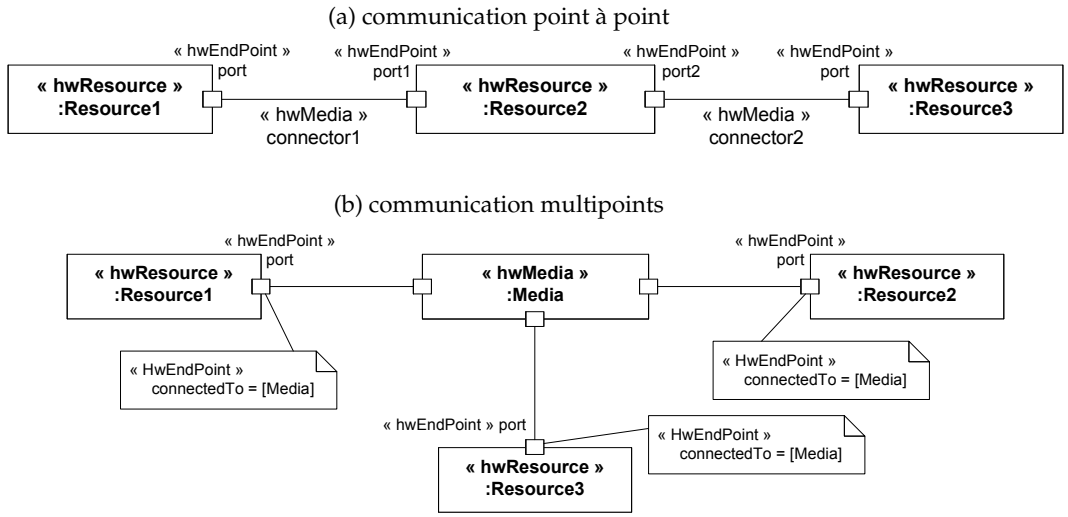


FIG. 3.36 – Connexions de communications

– non-typés s'ils représentent des communications de données, sachant que UML n'impose pas le typage pour le concept Connector [15].

Etape 8

Définir les ports :

Après avoir décrit la structure interne de la classe de ressource en cours de modélisation, on spécifie, dans cette étape, ses ports extérieurs qui lui permettent d'interagir avec son environnement. Dans le diagramme de composite structure UML, un `Port` est une `Property` qui n'est pas nécessairement typée mais qui porte un nom et une multiplicité. Dans le cadre de cette méthodologie, on n'impose pas de typage des ports cependant on exige que tout port soit stéréotypé en `HwEndPoint` et/ou que sa classe type le soit.

La classe PMI a deux ports (voir figure 3.38), toCPU qui est non typé et qui applique le stéréotype `HwEndPoint`, et toPLMB qui est typé par LMB_Interface. LMB_Interface est une ressource définie lors d'une itération antérieure (voir figure 3.37), elle désigne un port qui se connecte au bus mémoire LMB, on lui a donc appliqué le stéréotype `HwEndpoint` dont on a paramétré la `tag definition connectedTo` par la classe LMB, puis on a différencié plusieurs types de ports : maître, esclave ou les deux à la fois. Enfin, signalons qu'on aurait pu re-stéréotyper toPLMB pour surcharger sa tag value en PLMB, qui est une ressource héritière de LMB.

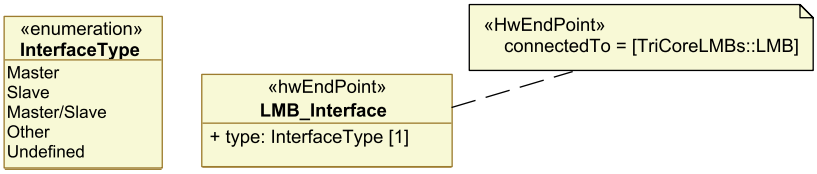


FIG. 3.37 – Classe LMB_Interface

Définir les délégations :

Maintenant que les ports de la classe sont définis, il faudra les associer au ports des sous-composants afin d'exprimer les délégations des traitements de données et de services. On utilise pour cela des connecteurs auxquels on peut appliquer, si nécessaire, les stéréotypes de communication.

La classe composite PMI délègue ces ports à la DCU via deux bus de données, un à 64 bits de largeur pour les transferts PLMB et un bus 128bits pour les instructions CPU (voir figure 3.38).

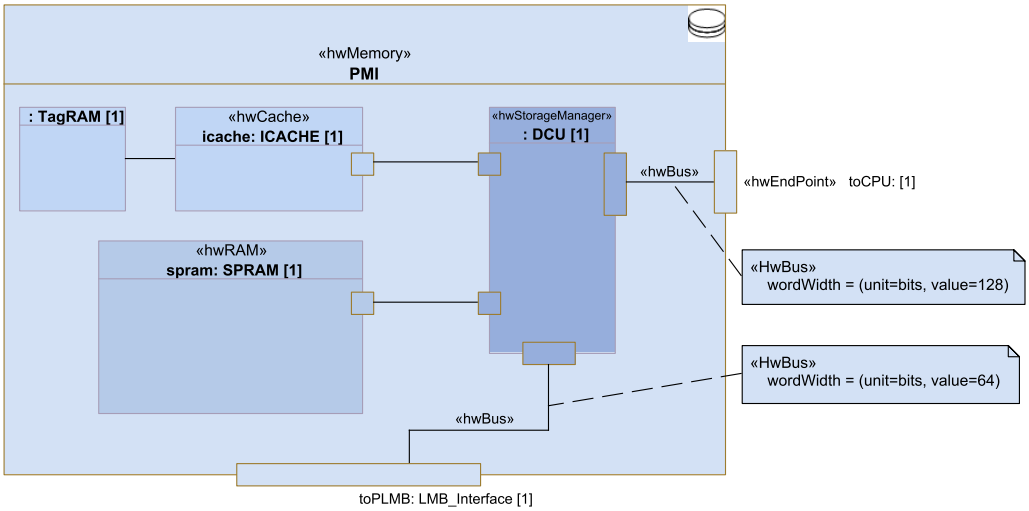


FIG. 3.38 – Classe composite PMI : Ports

Etape 9

L'étape 9 est une étape test, qui contrairement au reste des étapes, n'apporte rien au modèle en construction. Rappelons que notre méthodologie est itérative et incrémentale dans le sens des compositions, puisqu'on commence par modéliser les ressources élémentaires et on itère ensuite vers des ressources de plus en plus composites. Chaque itération permet de modéliser entièrement une ressource du matériel, en spécifiant ses caractéristiques, sa structure interne et son interaction avec son environnement. L'étape 9 consiste à tester si la classe courante est la classe représentant la plateforme entière, et dans ce cas on entame notre dernière étape (étape 10), sinon, dans le cas contraire, on réitère depuis la première étape (étape 1).

Terminer l'itération :

Quand la classe courante correspond à la plateforme entière, cela signifie que la modélisation au niveau classe est terminée, et que normalement à ce stade, toutes les ressources sont déjà modélisées et référencées depuis la classe de la plateforme. On passe donc à l'étape 10 pour instancier notre modèle.

Dans la figure 3.39, on obtient la classe TC1796_CPU_Subsystem qui correspond au

diagramme de blocs de la figure 3.21. On a donc réalisé grâce à cette méthodologie une représentation formelle du sous système de calcul du *TC1796* d'*Infineon*.

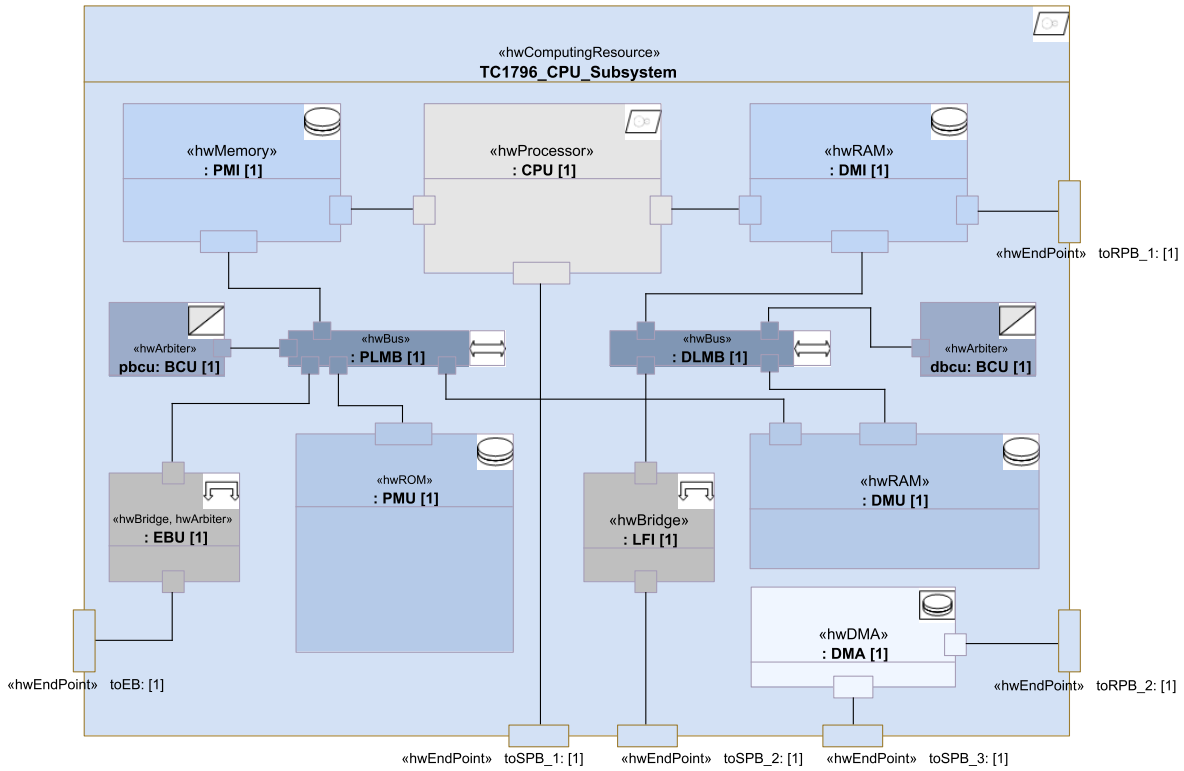


FIG. 3.39 – Classe TC1796_CPU_Subsystem

Réitérer :

Quand la classe courante ne correspond pas à la plateforme entière, cela signifie qu'on n'a pas fini de modéliser les ressources qu'elle contient, et qu'on continuera les itérations. Il faudra donc choisir la prochaine ressource à modéliser en respectant l'ordre des compositions. Une ressource ne doit être élue pour faire l'objet d'une itération, que si tous ses sous-composants ont déjà été modélisés lors d'itérations précédentes.

D'un point de vue algébrique, la composition dans le monde du matériel est une relation d'ordre strict, car elle est naturellement irréflexive, transitive et fortement antisymétrique. Si on note la relation «*contient*» par $\supset$, avec R l'ensemble des ressources et M l'ensemble de celles déjà modélisées, notre règle revient donc à choisir une ressource parmi l'ensemble N où :

$$N = \{r \in R/M | \forall s \in R, r \supset s \Rightarrow s \in M\}$$

La composition est aussi une relation d'ordre partiel, et en conséquence, l'ensemble N n'est pas nécessairement un singleton. On laisse alors libre le choix d'une ressource parmi N, l'utilisateur pourra prendre en compte les bonnes pratiques de la modélisation objet en suivant l'ordre des héritages important en étape 1 et étape 2 et en respectant le sens des associations du modèle et des méta-associations de HRM importants pendant l'étape 4. Un exemple de respect de méta-association unidirectionnelle de HRM, est de modéliser une `HwMemory` avant son éventuel

contrôleur `HwStorageMemory`.

D'un deuxième point de vue, si on considère le graphe orienté dont les noeuds sont les ressources de notre plateforme et dont des arcs lient chaque ressource aux ressources qu'elle contient. On obtient, dans ce cas, un DAG (graphe acyclique orienté) avec un sommet unique représentant la classe de la plateforme. Notre règle d'itération est alors un parcours en profondeur de ce graphe. Notons qu'une multitude de tels parcours existe.

On illustre en figure 3.40 une partie du graphe courant correspondant au sous système de calcul du `TC1796`. On y voit les étages de compositions qui imposent un certain ordre de parcours. Dans cette configuration plusieurs ressources peuvent faire l'objet de la prochaine itération, il s'agit de $N = \{DMA, DMI, SPRAM, ICACHE, TagRAM\}$, cependant PMI et TC1796_CPU_Subsystem ne sont pas candidates.

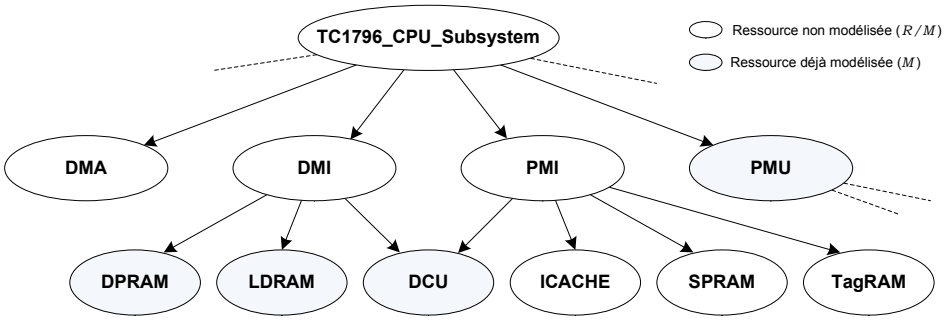


FIG. 3.40 – Graphe de compositions du *TC1796*

Pour conclure cette étape, signalons que ce schéma itératif, présenté ci-dessus, a été préféré au schéma récursif, qui consiste à

- Interrompre la modélisation de la ressource courante en étape 5 dans le cas où l'un de ses sous-composants (parts) n'est pas encore modélisé.
- Modéliser ensuite et en entier la ressource correspondant à ce sous-composant.
- Reprendre enfin la modélisation depuis l'étape 5.

Nous justifions ce choix comme bonne pratique, puisqu'il est plus rigoureux et efficace de modéliser une ressource en faisant abstraction de tout environnement particulier.

Etape 10

Nous arrivons à la dernière étape, où on instancie le modèle qu'on a construit en plusieurs itérations. Il s'agit de concrétiser la plateforme matérielle en spécifiant ses caractéristiques particulières. Pour cela nous procédons en plusieurs sous-étapes qu'on itère en respectant, de préférence, le même ordre de parcours des ressources adopté lors de la modélisation des classes entre l'étape 1 et l'étape 8.

Instancier la ressource :

On instancie chaque ressource depuis la classe correspondante modélisée précédemment.

Dans la figure 3.41, avant d'instancier pMI, on commence par instancier iCACHE, tagRAM, sPRAM, dCU et pMItoPLMB.

Spécifier les slots (attributs) :

On spécifie les `Slot`(s) correspondant aux attributs définis en étape 4.
Dans la figure 3.41, on fixe l'attribut type du port pMItoPLMB comme Master/Slave (voir figure 3.37).

Spécifier les slots (parts) :

On spécifie les `Slot`(s) correspondant aux `Part`(s) définies en étape 5. Nous exigeons donc de l'utilisateur de respecter le même ordre de compositions appliqué pendant la modélisation
Dans la figure 3.41, on instancie chaque `Part` de la pMI définie en figure 3.31.

Spécifier les slots (ports) :

On spécifie les `Slot`(s) correspondant aux `Port`(s) définis en étape 8. Il ne s'agit, de toute évidence, que des ports typés.
Dans la figure 3.41, on instancie le port toPLMB qui lie la mémoire PMI au bus PLMB cependant on omet d'instancier le port toCPU qui la lie au CPU (voir figure 3.38).

Lier les slots :

Afin d'améliorer la lisibilité, on spécifie les `Link`(s) qui lient les slots entre eux, et qui correspondent aux associations définies dans l'étape 4, aux connecteurs définis dans l'étape 7 ou aux délégations définies dans l'étape 8. Dans UML, on ne peut typer un lien que par une association, et il est donc impossible de spécifier qu'un link correspond à un connecteur ou une délégation. Nous proposons à l'utilisateur de calquer ce que le diagramme UML de composite structure propose entre instance, role et classe (voir page 190 du document [15]), il s'agit d'appliquer la règle de nommage suivante :

$$link/connector : association$$

Dans la figure 3.41, on a une liaison entre tagRAM et iCACHE correspondant à une association (voir figure 3.29), deux liaisons entre dCU et iCACHE et entre dCU et sPRAM correspondant à des connecteurs (voir figure 3.33) et une liaison entre pMItoPLMB et dCU correspondant à une délégation (voir figure 3.38).

Réappliquer les stéréotypes :

Les stéréotypes de HRM étendent parmi d'autres la métaclasse UML `InstanceSpecification` (voir modèle général en figure 3.4). Il est donc possible de réappliquer, une dernière fois à l'instance, les stéréotypes déjà appliqués à la classe. Cela afin d'annoter des caractéristiques qui relève du niveau instance et qui ne sont pas communes aux autres ressources de même nature.
Dans la figure 3.41, on réapplique des stéréotypes à iCACHE, sPRAM et pIM pour annoter leur fréquence de fonctionnement à l'intérieur de notre exemple *TC1796* qu'on a cadencé à `150MHz`.

2.3 Bilan

Par la présente méthodologie, nous proposons en plusieurs étapes une utilisation efficace du profil HRM dans UML. Nous limitons, pour ce faire, les mécanismes UML à utiliser et nous leur donnons une sémantique claire. Ensuite nous guidons l'utilisateur étape par étape et de manière précise dans la modélisation de la plateforme entière. Tout au long de cette méthodologie, la prise en charge du matériel est garantie par l'application de stéréotypes

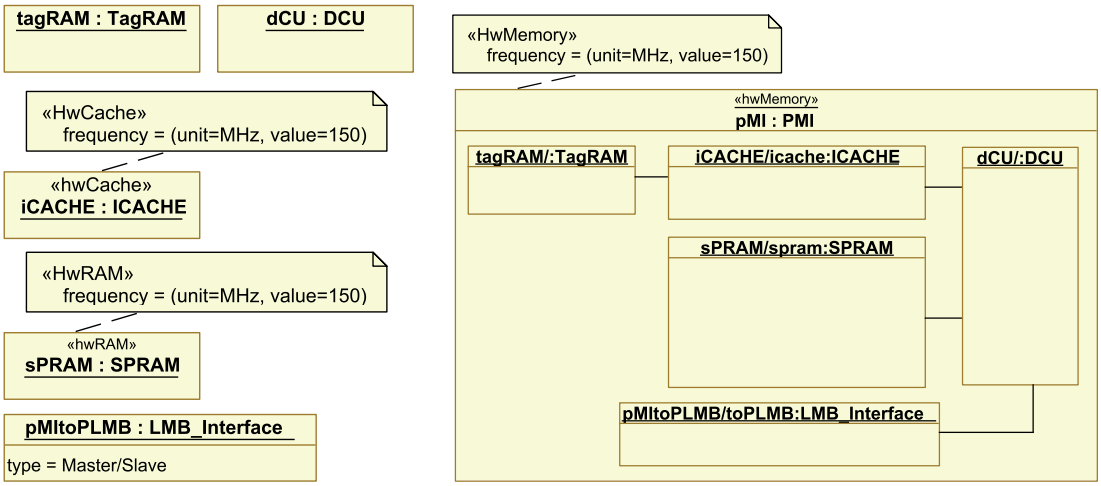


FIG. 3.41 – Instanciation de la PMI

HRM sur les différents concepts, une ressource peut se voir appliquer des stéréotypes jusqu'à trois fois pendant le processus de modélisation, à la création de la classe (étape 2), à son insertion dans la classe contenante (étape 6), et lors de son instanciation (étape 10).

Notre méthodologie reste cependant optionnelle, et même si elle regroupe de bonnes pratiques, l'utilisation de HRM indépendamment d'elle est évidemment possible. Elle est également orthogonale à plusieurs aspects, c'est à dire qu'elle n'a pas d'influence sur eux, il s'agit de :

Cas d'utilisation : La méthodologie est utilisée de la même manière dans les trois cas d'utilisation de HRM énumérés en section 1.1. Elle ne dépend point du niveau de détail du modèle ou d'un quelconque cas d'application.

Raffinement : La méthodologie peut s'inscrire parfaitement dans un processus de raffinement, puisqu'elle décrit à un instant donné une plateforme matérielle, et pourra donc être appliquée à chaque étape de raffinement.

Organisation du modèle : L'utilisateur est libre d'organiser les ressources modélisées sous la hiérarchie de paquetages (`package`) de son choix. A titre d'exemple il peut se baser sur des critères comme le constructeur, la nature ou la complexité.

Comportement : La méthodologie n'appuie que la modélisation structurelle de la plateforme, l'utilisateur peut, à tout moment, décrire le comportement des ressources par des vues comportementales grâce aux diagrammes UML de comportement.

Application d'autres profils : Comme on a vu en étape 2, UML permet l'application de plusieurs profils sur un même modèle, il est donc possible de combiner les concepts HRM avec d'autres concepts provenant d'autres profils tel GASPARD [43] (voir chapitre 4 section 4), AUTOSAR [42], UML for SystemC [38]...

Il est vrai que cette méthodologie est stricte et ne laisse que rarement le choix à l'utilisateur quant aux mécanismes UML à utiliser. Elle est néanmoins adaptée aux nouveaux utilisateurs d'UML, garantit une consistance du modèle final et nous épargne la spécification de multiples règles OCL. Quand un modèle de plateforme est consistant c'est-à-dire bien formé, on peut l'exploiter pour diverses manipulations, telle la simulation du matériel qu'on détaillera dans la section suivante.

3 Simulation de la plateforme matérielle

La simulation d'une architecture matérielle pour tester sa capacité à fournir une plateforme d'exécution adéquate à l'application logicielle, concentre énormément d'avantages. Elle améliore la flexibilité, accélère le processus de développement, fait gagner du temps et de l'argent, et permet une communication efficace entre les flots logiciel et matériel. En effet, les développeurs ne dépendent plus de la disponibilité du matériel et peuvent explorer ainsi plusieurs architectures, y compris de nouvelles configurations. En plus, le code de l'application en cours de développement peut être testé sur la plateforme simulée dès les premières étapes de conception. La simulation offre aussi plusieurs avancées techniques dans le débuggage du logiciel et du matériel [49].

Comme la simulation du matériel devient une pratique courante dans le processus de développement de systèmes embarqués hétérogènes, nous l'avons désignée comme un des trois cas d'utilisation de HRM explicités en section 1.1. Il s'agit d'utiliser UML et HRM comme interface commune aux outils de simulation. En effet, l'utilisateur pourra profiter des mécanismes de UML/HRM pour décrire une architecture matérielle dans un modèle qui sera traduit de manière automatique puis interprété par un outil de simulation. La plupart des outils de simulation sont basés sur des simulateurs de jeu d'instructions (ISS) et ne simulent donc qu'un processeur avec éventuellement une mémoire vive exécutant un code assembleur. Cependant, nous cherchons à simuler une plateforme d'exécution entière en prenant en charge les processeurs, les mémoires, les périphériques, et les différents moyens de communication. Un tel environnement de simulation, nous permettra d'exécuter des applications complexes sans aucune modification et de démarrer des systèmes d'exploitation.

Après diverses manipulations, nous avons opté pour l'outil *Simics* [48] de *Virtutech* introduit en section 2.3.2 du chapitre 2. *Simics* est capable de simuler un grand nombre de composants du matériel dont les processeurs *PowerPC*, *ARM*, *SPARC*, *x86*, les mémoires *SRAM*, *DDR*, *Flash*, les bus *I2C*, *PCI*, les ports série, les *Timers*, etc. *Simics* permet d'exécuter une application avec éventuellement un système d'exploitation et des pilotes de composants de manière identique à une plateforme matérielle réelle. *Simics* comprend deux niveaux de simulation :

- Une simulation fonctionnelle très rapide mais basée uniquement sur un modèle comportemental du matériel.
- Une simulation au cycle près plus lente mais simule de manière précise une description du matériel spécifiée par le dispositif *MAI* (Micro-Architectural Interface) de *Simics*.

Ce second niveau de simulation est nécessaire à la prise en charge des contraintes de temps lors des simulations de systèmes temps-réel. *Simics* propose également deux façons de définir une plateforme d'exécution matérielle :

- La première consiste à réutiliser des machines prédéfinies en les paramétrant si nécessaire. *Simics* en définit un grand nombre couvrant la plupart des architectures existantes et permet typiquement de paramétrer les tailles mémoires, le nombre et les fréquences des CPUs, etc.
- La seconde consiste à configurer sa propre machine à base de composants qu'on paramètre et qu'on connecte les uns aux autres, ces composants peuvent provenir de la riche librairie de composants fournie par *Simics* ou bien ils sont implémentés par

l'utilisateur en C++, Python et/ou le langage DML (Device Modeling Language) de *Simics*.

Pour donner plus de liberté à l'utilisateur, nous allons réaliser, dans le cadre de ce travail, la deuxième procédure.

3.1 Vue d'ensemble

Par notre introduction ci-dessus, nous avons expliqué que la simulation est une vraie alternative à l'utilisation du matériel physique pendant le processus de développement de systèmes à composante matérielle. Nous allons donc considérer, dans ce travail, que la simulation est une implémentation des modèles du matériel exactement comme la réalisation physique du matériel l'est. Nous allons ensuite appliquer l'un des principes de l'approche MDA [6] qui consiste à séparer le modèle de conception de son implémentation sur une plateforme particulière. Pour ce faire, nous utiliserons HRM comme un langage de modélisation général et indépendant de tout environnement de simulation, et on traduira automatiquement les modèles du matériel spécifiés en UML/HRM (PIMs : Platform Independant Models) vers le langage d'entrée de l'outil de simulation cible. Pour illustrer cette approche, nous la réaliserons avec `Simics` comme environnement de simulation cible.

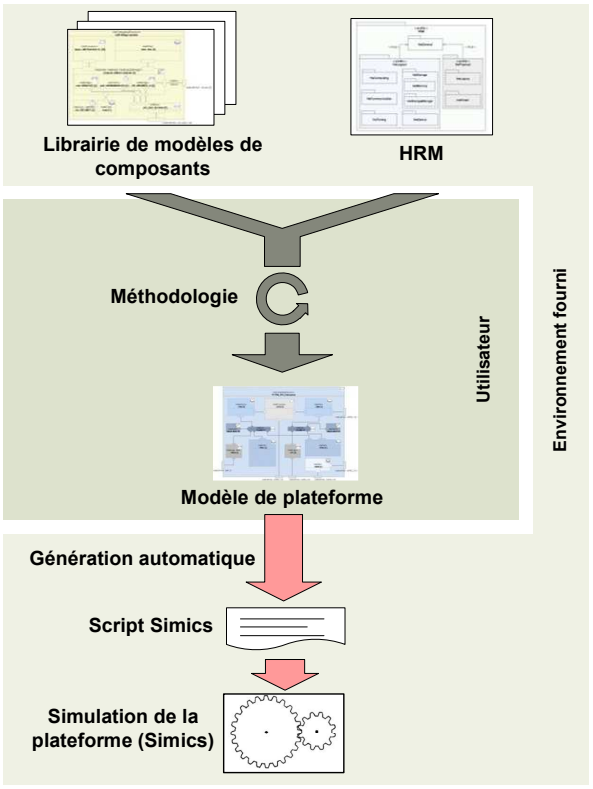


FIG. 3.42 – Processus de simulation de la plateforme matérielle

Pour commencer nous allons modéliser grâce à HRM les composants matériels de la librairie *Simics*, nous obtiendrons ainsi une librairie de modèles UML/HRM de ressources matérielles. Cette librairie sera ensuite fournie à l'utilisateur qui appliquera notre méthodologie (voir section 2) pour modéliser une plateforme matérielle. L'utilisateur pourra

donc utiliser les ressources de la librairie comme des composants de sa plateforme et grâce à notre méthodologie c-à-d à travers des itérations successives dans le sens des compositions, il pourra décrire la plateforme imaginée. Une fois que le modèle de la plateforme matérielle est achevée, nous allons générer automatiquement le script équivalent qu'on exécutera sous `Simics`. L'acheminement de ce processus est illustré en figure 3.42, nous démontrons ainsi que grâce à HRM, il est possible d'utiliser UML dans la conception d'architectures matérielles et de profiter pleinement de ces mécanismes. Dans la suite nous expliquerons séparément chacune des phases de ce processus, ainsi que l'environnement et l'outillage fournis.

3.2 Modélisation de la librairie de composants

L'objectif principal de cette section est d'expliquer comment nous avons modélisé la librairie de composants *Simics* et les choix conceptuels et techniques que nous avons adoptés pour ce faire.

3.2.1 Structure de la librairie

Notre tâche a été facilitée du fait que *Simics* aie une structure objet. En effet *Simics* est implémenté sous Python et C++, cependant il a une terminologie spécifique et une sémantique plus large que le paradigme objet. Nous avons donc traduit chacun de ses concepts en UML/HRM selon la table suivante :

Simics	**UML/HRM**
module	UML : :Package
class	UML : :Class
object	UML : :InstanceSpecification
component	HRM : :HwResource
toplevel component	HRM : :HwComputingResource
connector	HRM : :HwEndPoint

Pour *Simics*, le concept de *component* est central, il dénote une ressource du matériel (`HwResource`) qui peut être employée dans la construction d'une plateforme (appelée *machine* ou *configuration* par *Simics*), un *component* peut être implémenté par une ou plusieurs classes. On représente par `HwComputingResource` le cas particulier de *toplevel component* qui dénote une ressource pouvant à elle seule exécuter une application.

Comme illustré en figure 3.43, nous nous limiterons à la modélisation d'une partie de la librairie de composants *Simics*, cette partie suffira amplement à illustrer notre processus de simulation. Ils s'agit entre autres de processeurs *PowerPC*, de processeurs *Itanium64*, de plusieurs cartes *PCI* et *ISA*, de différentes mémoires *SRAM*, *SDRAM*, *DDR*, *DDR2*, et de ressources standards comme des bus et des disques durs. Notons que la classification des composants dans *Simics* n'est pas fonctionnelle à l'image du sous-profil `HwLogical` de HRM, elle est pour des raisons d'adéquation du matériel, organisée par familles de produits, par types d'architecture ou par protocoles de communication. Nous respecterons au mieux la taxinomie et la terminologie *Simics*.

FIG. 3.43 – Structure de la librairie de ressources

3.2.2 Modélisation des attributs

Simics permet de paramétrer les composants de sa librairie via des attributs typés. Ce typage n'est pas complexe en comparaison aux puissants mécanismes de typage fournis par le profil NFP de MARTE, et la richesse de la librairie de types BasicNFP_Types utilisée par HRM. Il s'agit de *string*, *integer*, *boolean*, *floating-point*, *list* et *dictionary* auxquels nous associons respectivement `NFP_String`, `NFP_Integer`, `NFP_Boolean`, `NFP_Real`, `VSL::TupleType` et `UML::DataType`. Signalons aussi que *Simics* ne supporte pas les unités, l'usage de HRM permet donc de satisfaire cela.

Un attribut dans *Simics* est également désigné comme *Required* ou bien *Optional*, nous traduisons cela comme une multiplicité UML de valeur [1] pour *Required* et [0..1] pour *Optional*. Toutefois, si un attribut est défini comme *Required*, la `tag definition` ou l'attribut qui lui correspond doit impérativement avoir une valeur par défaut. Cette valeur sera utilisée par notre générateur de code si l'utilisateur omet de paramétrer cet attribut pendant la modélisation de la plateforme. Dans la figure 3.44, les attributs de *connector* sont *Required*, ils ont donc une multiplicité à 1 et des valeurs par défaut, aussi, la `tag definition` du stéréotype `HwPort` est fixé à `Other` par défaut.

3.2.3 Modélisation des ports

Dans notre processus l'utilisateur pourra paramétrer les composants de la librairie et les connecter entre eux. La connexion entre composants revient à lier leurs ports, nous commencerons donc par modéliser ces derniers en prenant en compte les spécificités de *Simics*.

Le modèle de la figure 3.44 représente les différents types de ports définis dans `Simics`, il fait partie du module *libsimics-common* (voir figure 3.43). *Simics* désigne le concept de port par *connector*, et le caractérise par :

type qui spécifie la nature du port, nous avons donc défini une classe abstraite *connector*

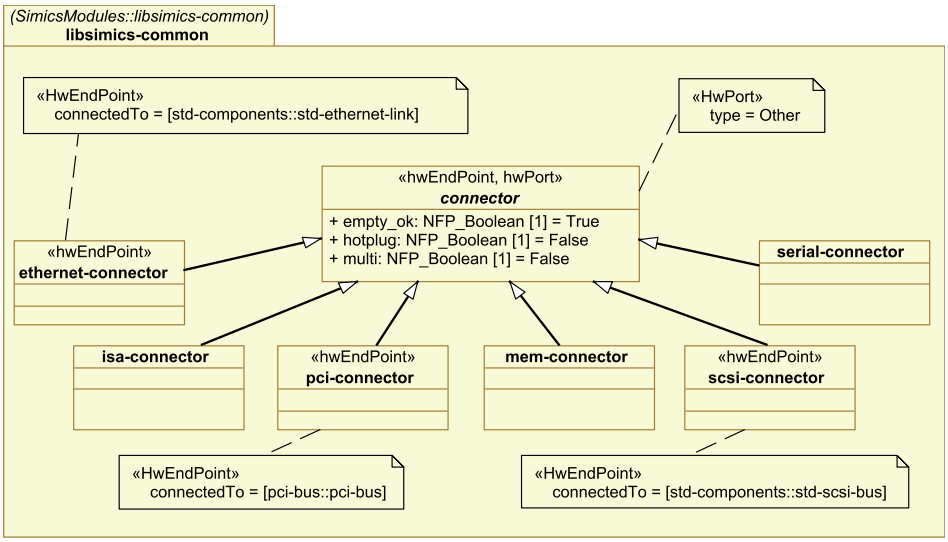


FIG. 3.44 – Modèle des ports

que nous avons spécialisée en plusieurs classes (*serial-connector*, *mem-connector*...). Ce typage permet de vérifier que seuls des ports de même type peuvent être connectés.

direction qui spécifie le sens physique du port, *Simics* distingue trois directions : *up*, *down* et *any*. Nous exprimons cela grâce au stéréotype `HwPort` qui distingue les trois types de port : `Male`, `Female` et `Other` (voir le métamodèle `HwLayout` en figure 3.11).

empty_ok, hotplug et multi sont trois booléens qui spécifient respectivement si le port peut rester non connecté, s'il peut être connecté à chaud (c-à-d pendant le fonctionnement) et s'il peut recevoir plusieurs connexions. Nous les représentons par trois attributs au niveau de la classe *connector*.

3.2.4 Modélisation des composants

Maintenant que nous avons montré comment modéliser les attributs et les ports de *Simics* en UML, nous allons appliquer rigoureusement notre méthodologie (voir section 2) pour modéliser, un à un, chaque composant de la librairie Simics. Nous garantissons par l'application de la méthodologie, la construction de modèles bien formés, consistants et non-ambigus. Nous avons réalisé autant d'itérations que de ressources modélisées dans notre librairie, nous nous sommes arrêtés évidemment à l'étape 9, car l'étape 10 ne sera franchie que lors de la modélisation de la plateforme d'exécution dans la section suivante.

Comme nous ciblons un outil de simulation particulier, nous nous sommes limités pendant le paramétrage des stéréotypes lors de l'étape 3 et l'étape 6, aux tag definitions supportées par *Simics*, c-à-d dont les valeurs sont prises en charge dans la simulation du composant. Dans l'autre sens quand Simics est plus riche que HRM en termes de paramètres de configuration pour un certain composant, nous utilisons l'étape 4 de notre méthodologie pour les spécifier sous forme d'attributs de la classe correspondante. Parmi les ressources que nous avons modélisées, beaucoup sont atomiques et nous avons donc modéliser principalement leurs caractéristiques et leurs ports, en sautant les étapes 5, 6 et 7

concernant leur structure interne. D'autres ressources sont cependant composites et dont la structure interne se compose de ressources plus petites de la librairie.

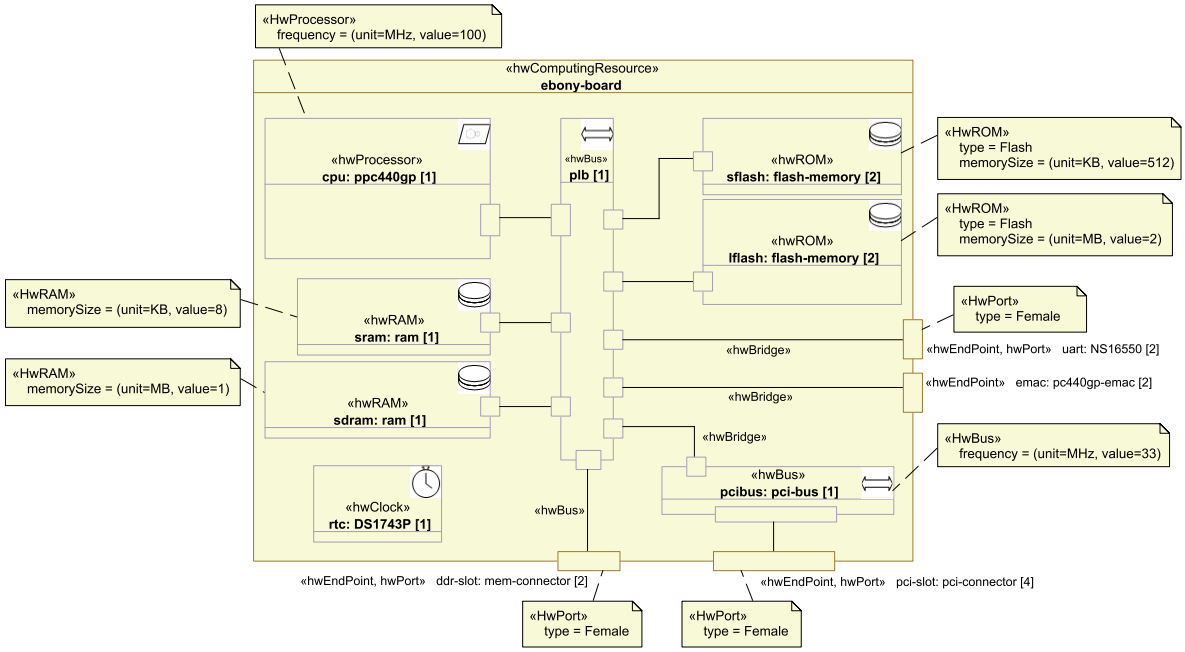


FIG. 3.45 – Modèle d'une carte *Ebony*

Nous avons modélisé, en tout, une cinquantaine de ressources, et il nous a fallu donc une cinquantaine d'itérations de notre méthodologie. Nous avons utilisé pour cela l'outil *Papyrus* [24] et nous avons construit ainsi une librairie conforme au *XMI* [24]. La figure 3.45 illustre une ressource composite de notre librairie, il s'agit de la carte *Ebony* qui référence une «*IBM/AMCC PPC440GP Board*», elle comprend un processeur *PowerPC PPC440GP*, deux mémoires *RAM* : une statique et une dynamique, quatre mémoires *Flash* : deux petites et deux larges, deux emplacements (slots) pour mémoire vive (*DDR*), quatre emplacements *PCI*, deux interfaces *Ethernet*, deux ports série et une horloge temps-réel.

3.3 Modélisation de plateformes

Après avoir expliqué comment nous avons réalisé une librairie de composants matériels simulés par *Simics*, nous montrerons comment l'utilisateur pourra modéliser une plateforme d'exécution matérielle à partir de ces éléments. D'abord, il devra importer dans son modèle de plateforme la librairie de composants que nous avons définie précédemment et que nous lui fournissons comme partie de l'outillage. Signalons, cependant, que l'utilisateur n'a accès dans cette librairie qu'aux ressources représentant des *Components Simics*, c-à-d celles qui ont une identité matérielle et qui appartiennent aux packages portant le suffixe «-components» que nous distinguons dans la figure 3.43. Ensuite l'utilisateur devra appliquer notre méthodologie de modélisation du matériel pour créer des ressources composées de celles de la librairie. Une fois la classe de la plateforme est atteinte, l'utilisateur aura créé une machine matérielle en paramétrant des composants de la librairie et en les connectant entre eux.

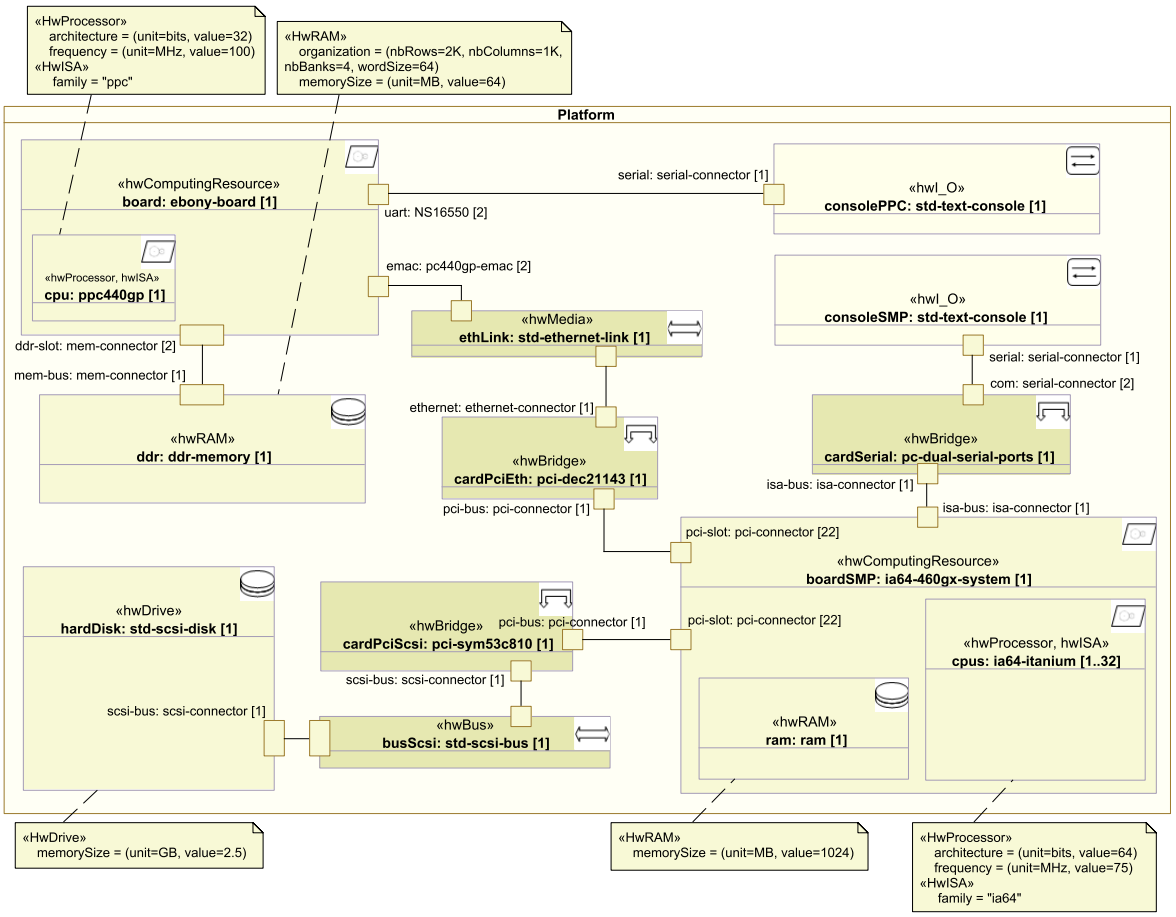


FIG. 3.46 – Modèle de la plateforme

Dans la figure 3.46, nous avons modélisé, grâce à notre processus et à titre d'exemple, une plateforme matérielle, fortement hétérogène, puisqu'elle réunit deux ressources de calcul très différentes : *board* et *boardSMP*. La première est un monoprocesseur de la famille *PPC* avec une architecture *32bits*. Pendant que la seconde ressource est un multiprocesseur à mémoire partagée (SMP) de la famille *Itanium* avec une architecture *64bits*, elle peut contenir jusqu'à 32 processeurs partageant une mémoire vive de *1GB*. Nous avons connecté *board* et *boardSMP* via un lien *Ethernet* (*ethLink*), mais il a fallu, d'abord, doter *boardSMP* d'une carte *Ethernet* (*CardPciEth*) que nous avons connecté à un port *PCI*. Nous avons aussi doté *boardSMP* d'un disque dur *SCSI* (*hardDisk*) via une carte `HwBridge` de type *pci-sym53c810* liant un port *PCI* au bus *SCSI*. La ressource *board* (*PPC*) est muni d'une mémoire vive *ddr* de taille *64MB* et dont la description exacte de son organisation est spécifiée (selon le métamodèle 3.6). Finalement cette plateforme comprend deux consoles série d'affichage *consoleSMP* et *consolePPC* qui lui permettent de communiquer avec l'extérieur. Pour finir, notons que tous ces sous-composants proviennent de la librairie fournie, comme le composant *board* qui est une *ebony-board* représentée dans la figure 3.45 et provenant du package *ppc440gp-components*.

Selon notre méthodologie, quand la classe de la plateforme est modélisée, c-à-d que nous sommes au niveau de l'étape 9 de la dernière itération, alors, nous passons à l'étape

10 qui consiste à instancier le modèle de classe pour le configurer une dernière fois. Cette instanciation est une tâche qui peut s'avérer assez longue quand il s'agit d'une classe de plateforme très composite, à l'exemple de celle de la figure 3.46 dont l'instanciation ne nécessite pas moins d'une soixantaine d'instances, y compris celles des classes de la librairie. Nous avons donc développé une extension à *Papyrus* qui réalise pour une classe quelconque cette fonctionnalité d'instanciation, l'utilisateur devra donc l'appeler au niveau de la classe de la plateforme. Il s'agit récursivement de :

1. créer une instance de la classe
2. parcourir un à un les sous-composants de la classe c-à-d les parts et les ports
   a. créer un slot pour chacun
   b. instancier une ou plusieurs fois la classe correspondante au slot (par un appel récursif), le nombre d'instances de la classe dépend de la multiplicité renseignée dans le modèle de classe. Quand il s'agit d'un intervalle nous prenons la borne inférieure (`/lower`). Dans l'exemple de notre plateforme, seul un processeur *ia64-itanium* sera donc instancié.
   c. référencer ces instances comme valeurs du slot et nous adoptons une numérotation adéquate lors du nommage de ces instances de même nature.
   d. référencer le slot dans l'instance
3. retourner l'instance

Il ne reste plus à l'utilisateur qu'à :

- renseigner les valeurs des attributs quand elles diffèrent des valeurs par défaut
- définir les instances manquantes par rapport aux multiplicités supérieures à celles spécifiées comme valeurs minimales dans le diagramme de classe
- réappliquer les stéréotypes si nécessaire pour paramétrer le matériel.

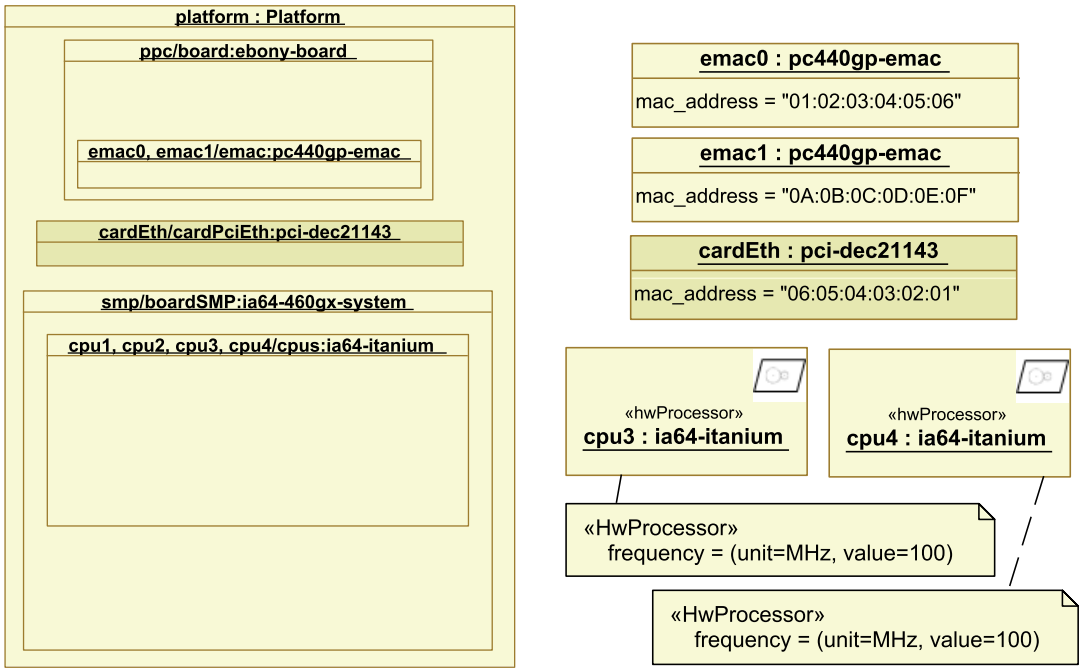


FIG. 3.47 – Instanciation de la plateforme

Revenons à notre exemple de plateforme, nous avons en premier instancié automatiquement la classe de la plateforme (voir figure 3.47). Ensuite, nous avons spécifié les valeurs des attributs tels les adresses *MAC* des différents ports et cartes *Ethernet*. Nous avons choisi

de réaliser une ressource de calcul *smp* formée de quatre processeurs, nous en avons donc défini trois de plus dont les deux derniers (*cpu3* et *cpu4*) seront cadencés à *100MHz* au lieu de *75MHz* pour les deux premiers. La modélisation de cet *overclocking* nous est permise grâce à la réapplication du stéréotype `HwProcessor` au niveau instance.

3.4 Simulation de plateformes

A ce stade, l'utilisateur a fini de modéliser la plateforme, la suite du processus est automatisée et fait partie de l'outillage fourni (voir figure 3.42). Il s'agit de générer le script *Simics* correspondant au modèle de plateforme puis lancer la simulation.

3.4.1 Génération de code

Il est vrai que *Simics* a une structure objet, dont nous avons bénéficié lors de la modélisation de la librairie de composants, cependant *Simics* prend en entrée un fichier script qui décrit la configuration de la machine à simuler. Ce script contient des commandes de différentes formes, comme celles pour :

Créer un composant :

```
(create-<component> "c_name" [<attribute> = <value>]*)
```

Trivialement, les attributs diffèrent d'un composant à l'autre et il faut donc souvent faire du cas par cas lors de la génération de code.

Connecter deux composants :

```
(<c_name1>.connect component = <c_name2> connector = <connector1>
                   dst-connector = <connector2>)
```

Accéder aux objets :

```
(<c_name>.get-compoenent.object <o_name>)
```

Cela permet de paramétrer les attributs non pris en charge par la commande de création. Cependant elle ne peut être exécutée qu'après la commande d'instanciation :

```
(instanciate-components)
```

Au niveau de l'outillage, nous avons principalement utilisé *Acceleo* [25], renforcé d'un ensemble de services que nous avons développé en *Java* grâce notamment à l'API du plug-in UML2 d'*Eclipse*. *Acceleo* est un outil puissant de génération de code à partir de modèles, qui grâce à un système de règles déclaratives parcourt efficacement le modèle pour générer le code correspondant.

Création des composants

Notre première étape de génération de code consiste à parcourir les sous-composants de la classe plateforme à simuler, et générer à chaque fois la commande de création adéquate quand ceux-ci appartiennent à notre librarie. Pendant la modélisation de notre librairie, nous avons représenté chaque attribut d'un composant *Simics*, par une tag definition d'un stéréotype HRM appliqué à la classe correspondante, ou par un attribut de cette classe le cas échéant c-à-d dans l'absence d'une équivalente tag definition. Par conséquent, afin de récupérer dans le modèle les valeurs des attributs *Simics*, nous avons développé principalement deux méthodes :

getTagValue(slot, stereotype, tagDefinition) : Ce service vérifie si le *stereotype* est appliqué et la *tagDefinition* est paramétrée afin de retourner la bonne `ValueSpecification`. Pour cela, il parcourt en premier l'instance attachée au *slot*, puis la part correspondante au *slot*, puis la classe type de la part, et finalement, les classes parents qu'il parcourt du parent le plus proche au parent le plus éloigné. Il s'arrête à la première valeur trouvée. Cet ordre de parcours est dû au séquencement de notre méthodologie qui permet de stéréotyper un composant à plusieurs étapes de la modélisation.

getAttrValue(instance, property) : Ce service vérifie si un slot correspondant à *property* est défini au niveau de *instance*, sinon il récupère la valeur par défaut spécifié dans la classe instanciée.

Grâce à la règle de conception que nous nous sommes imposés lors de la modélisation des attributs dans notre librairie (voir section 3.2.2), ces deux méthodes trouvent toujours une valeur à retourner. Notons aussi, que comme ces valeurs sont souvent typées par des types complexes avec unités provenant de la librairie MARTE : BasicNFP_Types, nous avons procédé à chaque fois à la mise en échelle de la valeur vers l'unité requise par *Simics*. Pour finir, quand une `tag definition` spécifiée par l'utilisateur n'est pas prise en charge par *Simics*, nous le signalons au moment de la génération des commandes.

```
import-ia64-components
import-memory-components
import-pci-components
...
(create-itanium-cpu cpu2 cpu_frequency= 75)
(create-itanium-cpu cpu3 cpu_frequency= 100)
...
(create-ddr-memory-module inst45_ddr rank_density= 64
rows= 11 columns= 10 module_data_width= 64 banks= 4)
...
(create-pci-dec21143 cardEth mac_address= "06:05:04:03:02:01")
...
```

Il s'agit ci-dessus d'un extrait du script généré par notre outillage depuis l'exemple de plateforme (voir figures 3.46 et 3.47). Après les commandes d'import, les deux premières commandes créent deux des quatres processeurs *Itanium* de la carte *smp* en récupérant les fréquences adéquates au niveau de la part pour *cpu2* et au niveau de l'instance pour *cpu3*. La commande suivante crée une mémoire *DDR* en paramétrant plusieurs attributs, dont *rank_density* qui correspond à la taille mémoire en *MB*, cette mesure est spécifiée par la `tag definition` `memorySize` de `HwRAM`. Puis *rows*, une puissance de 2 qui correspond à la valeur de `nbRows` ($2K = 2 \times 2^{10} = 2^{11}$) spécifiée dans la `tag definition` `organization` du même stéréotype. Nous obtenons, en tout, quinze commandes de création.

Connexion des composants

La seconde étape de génération de code consiste à produire les commandes de connexion entre les composants *Simics* créés pendant l'étape précédente. Pour ce faire, nous parcourons un à un les connecteurs du modèle de la plateforme qui lient les ports des différents sous-

composants. Nous vérifions que les ports sont de même type et de directions compatibles (voir section 3.2.3). Nous évitons ainsi la génération de commandes erronées. Notons tout de même que des commandes de connexion peuvent s'avérer impossibles pour des raisons technologiques ou générationnelles et seront de ce fait rejetées par *Simics*.

```
...
(ppc.connect component= inst45_ddr connector= ddr-slot1 dst-connector= mem-bus)
...
(cardEth.connect component= smp connector= pci-bus dst-connector= pci-slot2)
...
```

L'extrait ci-dessus du script généré, contient deux commandes de connexion, l'une pour connecter la mémoire *DDR* à la ressource de calcul *ppc*, et l'autre pour connecter la carte *Ethernet* dans un emplacement *PCI* de la ressource de calcul *smp*. Nous obtenons, en tout, quatorze commandes de connexion.

3.4.2 Simulation

Contrairement à un ISS (Instruction Set Simulator), *Simics* permet la simulation de la plateforme matérielle en entier, et prend en charge les différents composants. Simics permet aussi de simuler le matériel de manière si fidèle que le logiciel embarqué ne peut deviner la différence et ne nécessite aucun effort d'adéquation ou de portage vers cette plateforme virtuelle. Le simulateur que nous générons, pourra donc accueillir des applications natives ou démarrer tout un système d'exploitation avec les pilotes nécessaires. Ce simulateur est purement logiciel et n'a nul besoin d'un matériel particulier, il fonctionnera sur toute machine hôte. La simulation du matériel sous *Simics* rend le processus de développement/test beaucoup plus efficace. En effet la console de *Simics* permet d'interrompre/reprendre la simulation à tout instant, et donne accès à tous les états des composants simulés et aux valeurs de leurs registres. En plus d'être observés, ces états peuvent également être manipulés pour simuler, par exemple, des défaillances matérielles. En conclusion, l'utilisateur de cet outillage sera capable de tester et *debugger* logiciel et matériel conjointement.

Pour simuler notre exemple de plateforme, nous avons démarré deux noyaux *Linux 2.4*, le premier sur le noeud de calcul *ppc* compilé pour le jeu d'instructions *ppc32* et le second sur le noeud de calcul *smp* compilé pour le jeu d'instructions *ia64* avec l'option *SMP*. La figure 3.48 est une prise d'écran des deux consoles : *consolePPC* et *consoleSMP*. En haut, sur *consolePPC*, nous voyons qu'il s'agit bien du noyau indiqué. Nous voyons également que la taille mémoire de la *RAM*, les adresses *MAC* des ports *Ethernet* et les caractéristiques du processeur *PPC440GP* correspondent à celles spécifiées auparavant dans le modèle de la plateforme. En bas sur *consoleSMP*, le noyau reconnaît le quadri-processeurs *Itanium*, la sortie *ISA*, et les deux contrôleurs *SCSI* et *Ethernet*. Nous constatons aussi que les deux processeurs 2 et 3 sont plus puissants (*BogoMIPS*) que les processeurs 0 et 1, car nous les avons cadencés à 33% de fréquence en plus (*100MHz* au lieu de *75MHz*). Pour finir, signalons que nous avons réalisé cette simulation sur une machine hôte monoprocesseur *x86* à architecture *32bits*, et que grâce à *Simics*, nous simulons un système distribué, multiprocesseurs et à architecture hybride *32* et *64bits*.

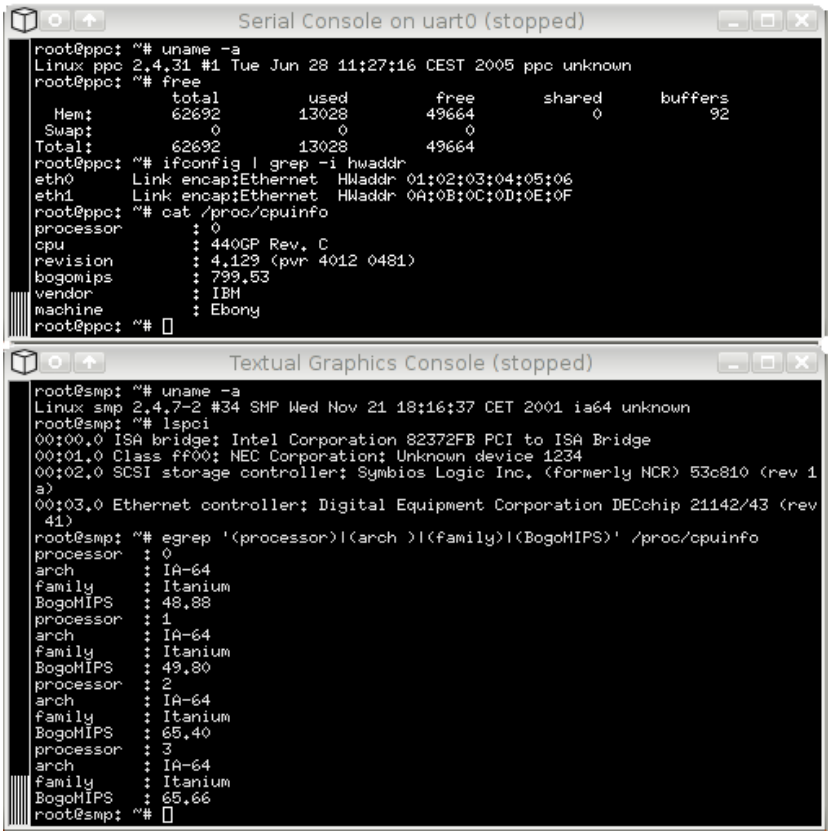


FIG. 3.48 – Exécution de la plateforme

3.5 Bilan

Après avoir démontré dans les précédentes sections que UML et HRM sont adaptés à la conception du matériel, le premier objectif de cette section était de réaffirmer que HRM est complet et qu'il offre un niveau de détail suffisant pour servir d'interface aux outils de simulation les plus précis. Le second objectif était de fournir au développeur du matériel un outillage riche et automatisé pour l'assister dans la conception de plateformes. Grâce à cet outillage, il pourra profiter des mécanismes de modélisation et des avancées qu'ils procurent, notamment en termes de visibilité, de réutilisabilité et de gain de temps, tout en gardant le même pouvoir d'expression et la même efficacité dans la conception.

Dans la suite de ce travail, nous utiliserons cet outillage à maintes reprises, surtout lors de la réalisation de notre cas d'étude en section 4 du chapitre suivant. Nous allons simuler plusieurs architectures matérielles sur lesquelles nous allons exécuter notre application temps-réel et toute l'infrastructure ACCORD.

La simulation permet une exploration architecturale plus rapide et plus large pendant la conception du système car elle n'est pas contrainte par la disponibilité du matériel. Cependant une fois que l'architecture matérielle est choisie et validée par la simulation, il est nécessaire de l'implémenter. Dans la section suivante nous allons montrer comment HRM peut communiquer avec les standards d'implémentation du matériel.

4 Implémentation du matériel

Afin de gagner en temps et en efficacité, le développement de systèmes embarqués est partitionné en plusieurs flots parallèles. Ces flots doivent garder entre eux une forte communication pour éviter toute redondance ou incohérence. Rien que pour la partie matérielle, on distingue deux flots. Le premier utilise une vue fonctionnelle du matériel à l'image de celle réalisée avec HRM, pour décider de l'architecture logicielle et de son allocation ou effectuer certaines analyses. Alors que le second est orienté implémentation et s'intéresse plus aux circuits et à la réalisation des composants. Il est vrai qu'UML est de plus en plus utilisé par ces deux courants et plusieurs profils ont été standardisés de part et d'autre, mais aucun travail jusqu'ici n'a réussi à les rapprocher, considérant qu'il s'agit de deux niveaux de détail disjoints et de deux natures de description différentes. Dans cette section, on montrera comment on peut synchroniser ces deux flots de conception en unifiant leurs deux standards respectifs HRM(MARTE) et IP-XACT [73].

4.1 Vue d'ensemble

Comme on l'a vu dans le chapitre 2 de positionnement en section 2.3.1, le système d'extension via des profils a encouragé l'adoption d'UML dans la modélisation du matériel en général et dans le flot d'implémentation en particulier. Plusieurs profils ont vu le jour, mais tous ont pour but de générer du code et utilisent de ce fait UML comme HDL (Hardware Design Language) c-à-d langage d'implémentation du matériel. En effet, le profil UML for SoC standardisé par l'OMG et le profil UML for SystemC donnent tous les deux une transcription un à un des concepts de SystemC en UML et garantissent ainsi une génération automatique du code SystemC. Ces profils sont par conséquent trop proches de l'implémentation et ont pour effet d'étendre UML avec la sémantique particulière au HDL généré. Un travail récent réalisé en collaboration avec STMicoelectronics s'est inspiré de IP-XACT pour résoudre ce problème. IP-XACT est un nouveau standard établi par le consortium SPIRIT dans le but de définir en schéma XML une représentation des concepts du matériel en dissociant les caractéristiques structurelles en termes d'interfaces, ports, registres, etc, de leurs implémentations en VHDL, SystemC ou autre HDL. Ce travail a donné naissance à un profil ESL (Electronic System Level) qui étend UML par les concepts définis dans IP-XACT. Une chaîne d'outils permettant la génération de code automatique depuis ESL vers SystemC en passant par UML for SystemC, fût également développée.

Comme notifiée ci-dessus, la séparation entre les deux flots de conception qui nous intéressent est principalement due à deux raisons :

Différence entre niveaux de détail :

Avant HRM, d'autres standards comme SPT et AADL permettaient de décrire fonctionnellement une plateforme matérielle, mais les modèles obtenus étaient à un tel niveau d'abstraction que cela impliquait une réelle rupture sémantique entre de telles vues fonctionnelles et tout modèle d'implémentation. Aujourd'hui, le métamodèle HRM est un langage fonctionnel assez détaillé et IP-XACT est un langage d'implémentation assez abstrait pour que la jonction soit possible par le biais d'UML et de leurs profils respectifs HRM et ESL.

Différence entre natures de détail :

Entre HRM et ESL cette différence reste vraie, car les deux vues obtenues sont orthogonales. Cependant on constate qu'elles sont complémentaires car si par exemple IP-XACT considère les composants comme des boites noires et se focalise sur leurs interfaces, HRM décrit la fonctionnalité de chaque composant sans fournir de détail au niveau des points de connexion. L'orthogonalité et la complémentarité seront amplement justifiées dans la suite.

UML a cette capacité de fournir plusieurs vues d'un même modèle. Grâce à l'orthogonalité, on va appliquer simultanément les deux profils HRM et ESL sur un même modèle du matériel sans créer d'incohérence sémantique. Puis grâce à la complémentarité on obtiendra un modèle complet dont on peut générer en permanence deux projections conceptuellement liées, une fonctionnelle et une d'implémentation. Il suffit pour cela de filtrer les stéréotypes de chaque profil. La figure 3.49 résume notre idée.

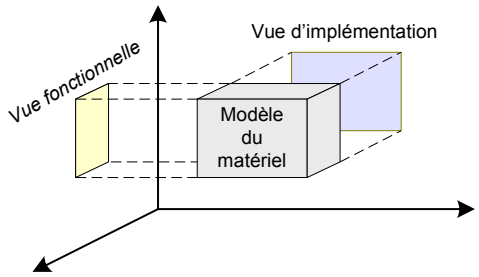


FIG. 3.49 – Projections du modèle du matériel

Les deux flots de conception seront dorénavant parfaitement synchronisés puisqu'ils partagent le même modèle du matériel.

4.2 Unification des flots de conception du matériel

UML autorise l'application de plusieurs stéréotypes sur un même élément, ces stéréotypes peuvent provenir d'un même profil et cela signifie que cet élément joue plusieurs rôles dans le domaine spécifié par ce profil (voir étape 2 de notre méthodologie en section 2), comme ils peuvent provenir de plusieurs profils différents et cela est un moyen de fusionner des concepts de différents domaines. On retiendra les deux options et on imposera au développeur que parmi les n stéréotypes appliqués à chaque ressource, il doit y avoir au moins un qui provient de HRM et au moins un qui provient de ESL.

Si on veut que la double-application de stéréotypes soit réalisable et qu'elle ne provoque pas d'incohérence, il suffit que les profils HRM et ESL remplissent deux conditions :

Structure commune :

Les stéréotypes de HRM et ESL étendent les mêmes concepts UML, les deux profils sont utilisés principalement dans le diagramme de classe pour la définition des ressources et dans le diagramme de composite structure pour la description de leur agencement et leur hiérarchie de compositions. Autrement dit, le métamodèle de ESL a une méta-structure équivalente au HwGeneral (voir section 1.3) de HRM. On en déduit que les deux profils peuvent structurellement être appliqués sur un même modèle UML.

Sémantiques orthogonales :

Il s'agit de vérifier qu'il n'existe pas de conflits entre la sémantique de HRM et celle de ESL. Si HRM décrit la fonctionnalité de chaque IP (Intellectual Property) dans la plateforme, ESL se focalise sur son implémentation. En effet, HRM définit un stéréotype spécifique à chaque fonctionnalité du matériel, pendant que ESL n'a qu'un seul stéréotype HWComponent qui dénote toute ressource matérielle. Dans l'autre sens, ESL donne une description fine des interfaces de chaque composant, des connecteurs qui les lient, de leurs registres accessibles depuis l'extérieur et du plan mémoire, mais tous ces points ne sont pas adressés par HRM.

Le but de ce travail est de faire communiquer et synchroniser plusieurs flots de conception d'un système hétérogène (logiciel/matériel), il s'agit d'un côté des flots dits à caractère fonctionnel et qui utilisent une vue du matériel réalisée avec HRM tels les flots :

- Développement logiciel pour décider l'architecture de l'application. Par exemple, quand la plateforme matérielle comprend plusieurs ressources de calcul, il serait important de distribuer et paralléliser le code en conséquent.
- Analyse de toute nature, WCET, ordonnançabilité, performance, sûreté, tolérance aux fautes, consommation d'énergie, dimensionnement, etc
- Allocation et le choix entre différentes configurations, cela est idéalement influencé par les résultats des analyses.
- Simulation fonctionnelle à l'image de ce qu'on a montré dans la section précédente.

D'un autre côté, on trouve les flots d'implémentation qui utilisent dans notre cas le profil ESL :

- Implémentation du matériel, où on rassemble les IPs, on vérifie leur adéquation, on définit le plan mémoire, etc
- Génération de code en HDL cible (ex. SystemC TLM pour ESL)
- Simulation matérielle. Quand celle-ci est opérée au niveau RTL (Register Transfer Level), elle offre un niveau de précision maximum.

4.2.1 Processus d'unification

Maintenant, on propose d'unifier les flots de conception énumérés ci-dessus grâce à un dispositif simple illustré en figure 3.50. On commence par définir un ensemble d'IPs dans une librairie commune de composants matériels. Cette librairie UML applique simultanément les deux profils HRM et ESL, et chaque ressource de la librairie doit être annotée par un stéréotype HRM qui relate sa fonction et un stéréotype ESL qui dénote son implémentation. Ensuite l'architecte du matériel doit importer la librairie et créer à partir de ses IPs une plateforme matérielle, tout en respectant les règles de constructions et les schémas de connexion et d'adéquation définis dans IP-XACT. Une fois la plateforme est modélisée, on peut automatiquement générer une projection du modèle en vue graphique fonctionnelle qui met en valeur les stéréotypes de HRM et cache ceux de ESL. Comme on peut générer automatiquement une vue d'implémentation qui ne met en valeur que les stéréotypes ESL et qui est conforme à IP-XACT.

Les flots de développement du matériel partagent grâce à notre processus un même modèle du matériel et chacun n'en voit que l'abstraction qui l'intéresse. La communication entre flots est ainsi garantie à son maximum. En effet, si lors d'un raffinement quelconque

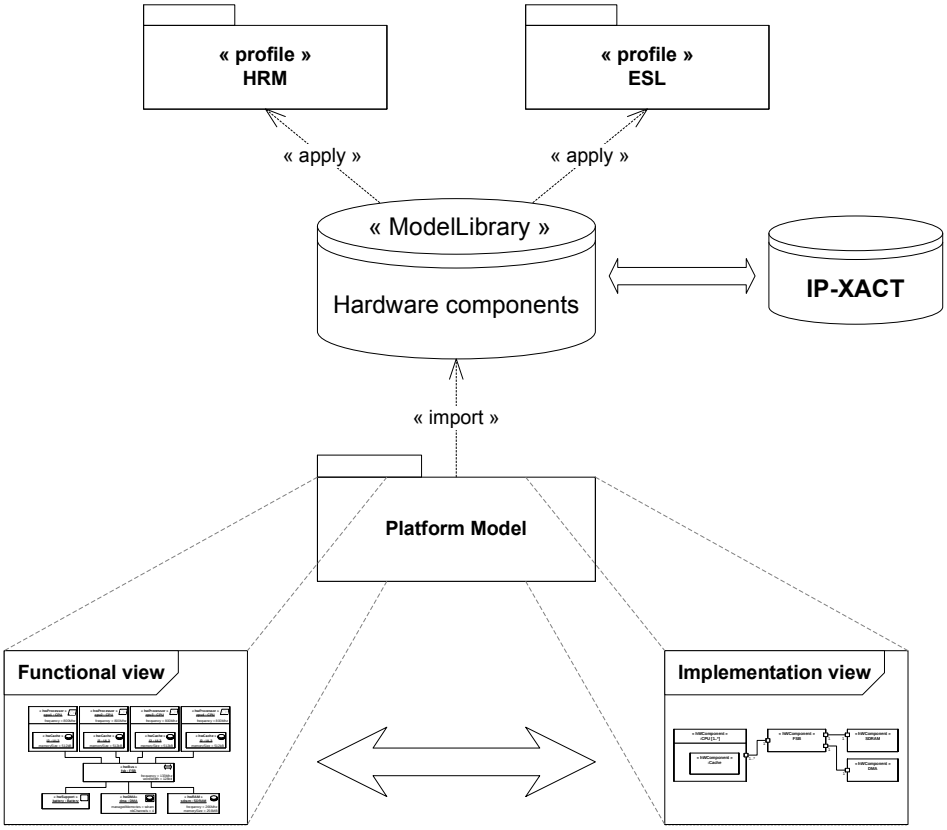


FIG. 3.50 – Processus d'unification

un des flots de conception modifie le modèle de la plateforme depuis la vue qu'il en a, cela est immédiatement répercuté sur les vues des autres flots. Signalons que comme pour toute donnée partagée, l'accés en écriture au modèle de la plateforme doit être exclusif.

4.2.2 Exemple

La figure 3.51 montre un exemple type d'une IP de la librairie de composants, il s'agit de *pwp_timer* auquel on applique à la fois les stéréotypes HRM qu'on voit au niveau de la classe *HwTimer* et des ports *HwEndPoint* et les stéréotypes d'implémentation ESL tel le stéréotype générique *HWComponent* au niveau de la classe et les stéréotypes *MasterBusIf* et *SlaveBusIf* au niveau des ports. Ces deux derniers distinguent un port de communication maître d'un port esclave. Lors de la construction de la plateforme, le standard IP-XACT impose entre autres qu'un port maître requière une interface qui est fournie par le port esclave auquel il est connecté.

Depuis un tel modèle d'IP en ESL, on peut générer grâce à une chaîne d'outils développée en collaboration avec *STMicroelectronics* jusqu'à 80% du code d'implémentation SystemC, qui inclut les hiérarchies, les ports, les déclarations de registres et leurs types d'accès, les algorithmes de décodage de l'adressage mémoire, etc. Le travail de thèse de S.Revol [74] explique en détail les transformations successives qui partent de modèles ESL conformes à IP-XACT pour produire du code SystemC TLM (Transaction Level Modeling).

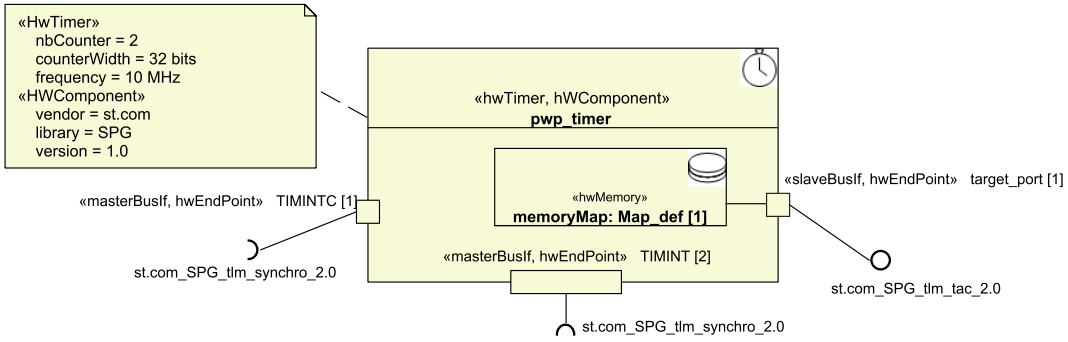


FIG. 3.51 – *pwp_Timer*

4.3 Bilan

Dans cette section, on a proposé un dispositif simple et efficace pour synchroniser les différentes disciplines de conception du matériel autour d'un même modèle UML de la plateforme matérielle. On utilise une multi-application de profils UML complémentaires qui ne crée pas d'incohérence sémantique. Cette technique rarement utilisée auparavant nous a permis de répondre à un besoin pressant d'une communauté du matériel traditionnellement partagée en deux courants selon qu'on fasse de l'implémentation ou bien de l'allocation et de l'analyse.

Le long des sections de ce chapitre, on a étudié exclusivement le développement du matériel. On a fourni un langage et une méthodologie pour le modéliser, un outillage complet pour le simuler et un processus efficace pour l'implémenter. Dans le chapitre suivant, on contribura à la prise en charge du matériel ainsi développé, dans la conception de l'application (logiciel) temps-réel embarquée (sur ce matériel).

CHAPITRE 4

Prise en charge du matériel dans la conception du système temps-réel

Si grâce au chapitre précédent, on est capable de modéliser la plateforme matérielle en identifiant une à une ses ressources. Nous allons définir dans ce chapitre, un langage de modélisation permettant d'identifier une à une les entités logicielles de l'application temps-réel. On consacrera une section à la spécification d'un métamodèle qui décrit la structure d'une application temps-réel modélisée par la méthodologie ACCORD/UML.

Une fois que notre co-développement logiciel/matériel est achevé, nous obtenons le modèle de l'application temps-réel en ACCORD et le modèle de la plateforme matérielle en HRM, nous entamons donc l'allocation. Nous allons fournir les règles d'allocation qui régissent les placements des entités logicielles sur les ressources matérielles. Nous développerons également quelques mécanismes d'adéquation pour adapter les configurations d'allocation à priori inadéquates.

Les règles d'allocation spécifiées, nous habilitent à établir la faisabilité structurelle de l'allocation. Afin de valider un système hybride fraîchement intégré, il faut ensuite enchaîner les phases d'analyses, de simulations et de tests en prenant compte des propriétés non-fonctionnelles requises.

Pour illustrer et appliquer les contributions de ce travail de thèse qui interviennent à différents niveaux du flot de conception, nous allons développer, comme cas d'étude, une chenille de robots unicycles qui roulent sans glisser sur un plan horizontal. Il s'agit d'un système temps-réel embarqué multi-tâches distribué répétitif et paramétrable.

1 Conception et modélisation de l'application temps-réel

Dans la perspective de prendre en charge les aspects matériels dans la conception des applications temps-réel, on développera tout au long de ce chapitre des mécanismes d'allocation et d'adéquation entre logiciel et matériel. L'allocation sera essentiellement un dispositif qui nous permettra de désigner pour chaque entité logicielle, un support d'exécution sous forme d'une ressource matérielle (ou d'un agencement de plusieurs ressources, voir section 2). Il est donc trivial que l'allocation ne devient possible que si elle est précédée par une identification des entités logicielles d'un côté et des ressources matérielles de l'autre. On a d'abord fourni dans le chapitre précédent un langage de modélisation HRM et une méthodologie qui habilite à concevoir et modéliser une plateforme matérielle et à identifier chacune de ses ressources d'exécution. De même, on devra fournir un langage qui nous permettra de modéliser le logiciel ou du moins en modéliser une abstraction afin d'identifier les entités logicielles à allouer.

Dans notre cas, la spécification de l'application temps-réel est soumise à la méthodologie ACCORD/UML développée au sein de notre équipe et qu'on a introduite dans la section 2.4.1 du chapitre 2. Cette méthodologie regroupe une cinquantaine de règles de conception qui guide le développeur depuis le cahier de charge jusqu'au modèle UML final et exécutable. ACCORD/UML définit un modèle d'exécution spécifique dit ACCORD qui régit avec déterminisme le parallélisme, la communication et le partage de ressources (logicielles) et prend en charge les contraintes de temps. Ensuite, tout modèle d'application temps-réel ainsi obtenu est déployé sur l'infrastructure ACCORD (voir section 2.4.2 chapitre 2) qui implémente ce modèle d'exécution. On consacrera cette section à la définition d'un métamodèle qui décrit la structure d'une application temps-réel modélisée grâce à la méthodologie ACCORD/UML et qui permet d'identifier de manière précise chacune de ses entités (objets) logicielles.

1.1 Contexte

La modélisation d'une application temps-réel avec ACCORD/UML repose principalement sur trois vues partielles, complémentaires et cohérentes d'un même modèle global :

Modèle structurel : Il définit l'architecture générale de l'application en termes de classes, de propriétés de classes, et de relations entre classes. Il est réalisé essentiellement par un diagramme de classes UML.

Modèle comportemental : Il décrit le comportement de chaque classe dans deux machines états-transitions de UML. La première spécifie son cycle de vie (vue protocole) et la seconde spécifie ses réactions à son environnement (vue déclenchement).

Modèle d'interactions : Il décrit des scénarios d'exécution ainsi que les différents échanges entre instances de l'application effectuant une tâche donnée.

On considère que le niveau d'abstraction du modèle structurel suffit à l'élaboration de l'allocation (voir section 2). On se limitera donc dans le cadre de ce travail aux aspects structurels de l'application temps-réel et on ne traitera que les quinze règles de conception d'ACCORD/UML le concernant. On ne traitera pas non plus les entités structurelles spécifiques à l'infrastructure (noyau) ACCORD comme les concepts *Zone* et *LoaderPRM* car on considère qu'ils relèvent de l'implémentation.

Parmi les chapitres du standard MARTE, on retrouve RTEMoCC (RTE Model of Computation & Communication) qui reprend en partie le modèle d'exécution ACCORD et beaucoup de ses concepts, notamment le concept d'objet temps-réel, d'objet passif protégé, de RTF (Real-Time Feature) etc. Cependant comme RTEMoCC était voué à la standardisation, il est resté générique sur certains points et détaillé sur d'autres. Prenons deux exemples :

- Les concepts d'interface active et interface passive qui désignent les points d'interaction du système avec les acteurs extérieurs, ne sont pas repris par RTEMoCC.
- Le modèle d'exécution ACCORD est basé exclusivement sur une politique d'ordonnancement EDF (Earliest Deadline First), mais RTEMoCC en propose plusieurs pour une meilleure généricité.

Nous avons voulu créer un métamodèle à l'image de RTEMoCC mais plus fidèle à la méthodologie ACCORD/UML et qui respecte à la lettre son modèle d'exécution et ses règles de conception.

Pendant la réalisation du métamodèle HRM lors du chapitre précédent, on a utilisé le mécanisme d'extension d'UML via des profils. Bien que ce mécanisme soit standardisé par l'OMG et considéré par ce dernier comme le seul moyen d'étendre UML efficacement, il reste très contraignant. Il autorise à étendre les métaclasses d'UML par des stéréotypes, mais il ne donne point accès aux méta-associations, on est dès lors obligé d'utiliser des règles OCL au niveau modèle ou de passer par des astuces telles celles expliquées pendant l'étape 7 de notre méthodologie de modélisation du matériel en section 2 du chapitre 3. Le système d'extension d'UML via des profils garantit une moindre modification portant principalement sur la sémantique des concepts d'UML et est éventuellement paramétrée par des `tag definitions` supplémentaires. On a proposé en section 1.4 du chapitre 2 deux autres techniques d'extension d'UML, qui considère le métamodèle d'UML comme une simple librairie à importer et/ou modifier il s'agit des extensions *lightweight* et *heavyweight*.

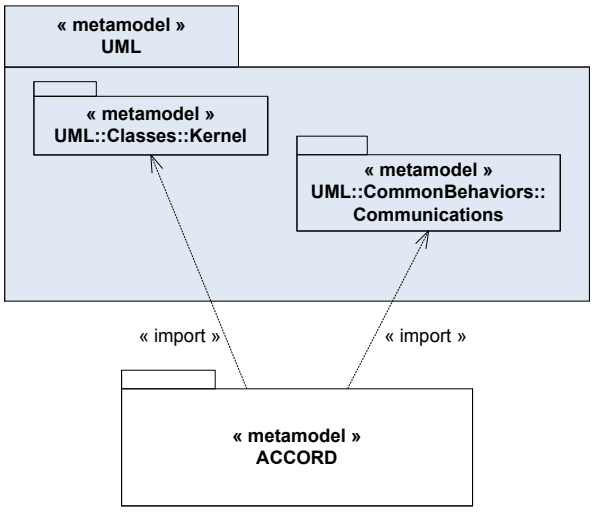


FIG. 4.1 – Extension *lightweight*

Afin de promouvoir ces alternatives, on adoptera la technique *lightweight* dans la réalisation du métamodèle de la structure ACCORD. Cette technique importe en lecture seule le métamodèle d'UML, permet d'hériter de ses métaclasses et de spécialiser

et/ou surcharger ses méta-associations. Comme le montre la figure 4.1, on importe précisément deux paquetages du métamodèle UML. Il s'agit de `UML::Classes::Kernel` et `UML::CommonBehaviors::Communications`.

1.2 Métamodèle ACCORD

En bref, on développera par le biais d'une extension *lightweight* d'UML, un métamodèle qui reprend fidèlement le modèle structurel d'ACCORD et qui permet d'identifier les entités logicielles de l'application temps-réel. Tout au long de la présentation de ce métamodèle, on notifiera les règles de modélisation de la méthodologie ACCORD/UML qu'il remplit. On respectera pour cela la numérotation des règles appliquée dans la thèse de S.Gerard [33].

1.2.1 Structure générale

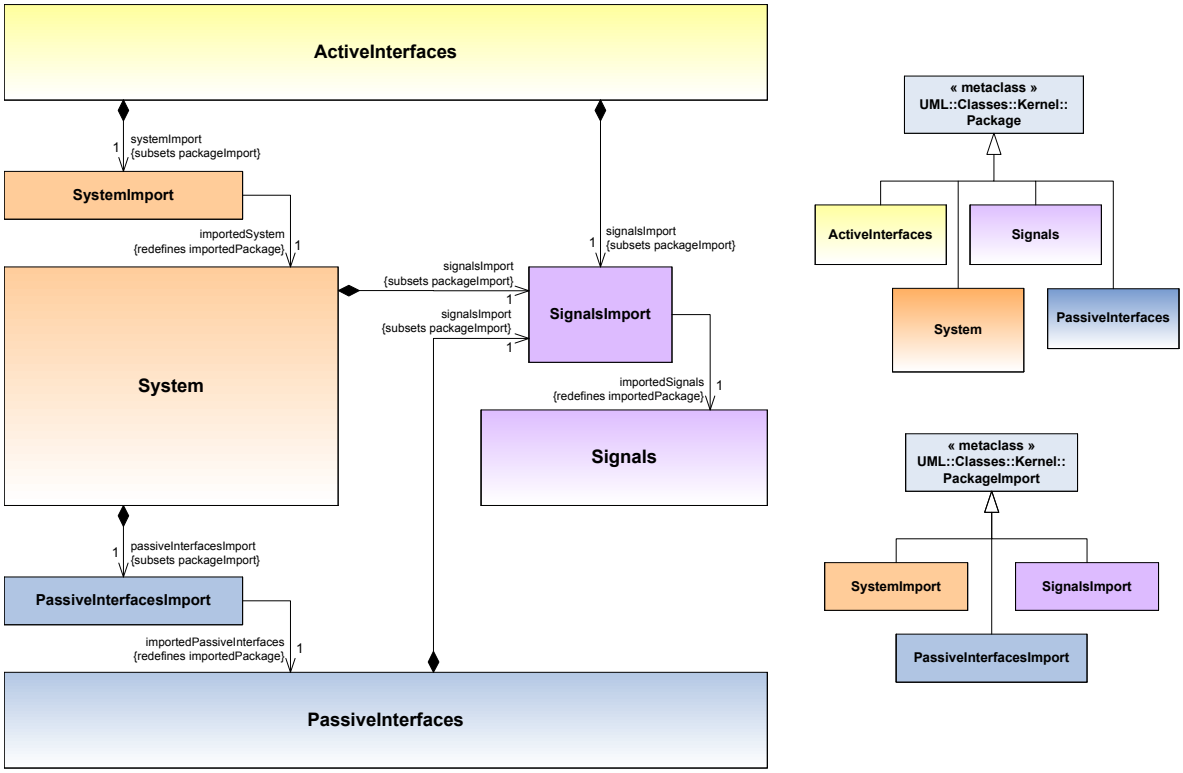


FIG. 4.2 – Structure générale du modèle de l'application

Pour accroître la réutilisabilité et l'efficacité des modèles construits, la méthodologie ACCORD/UML applique une architecture logicielle générique qui sépare le coeur du système de ses interfaces avec l'environnement. On distingue en tout quatre paquetages :

- ***ActiveInterfaces*** regroupe les points d'interaction qui stimulent le système.
- ***System*** est le modèle même de l'application.
- ***Signals*** contient les signaux échangés dans l'application.
- ***PassiveInterfaces*** regroupe les points d'interaction à partir desquels le système contrôle son environnement.

ACCORD/UML définit clairement la structure et l'orientation des dépendances entre ces paquetages. En effet, tous les paquetages importent le paquetage `Signals`. `ActiveInterfaces` importe également `System` et `System` importe en plus `PassiveInterfaces`. Seul `Signals` n'importe aucun paquetage.

Dans le métamodèle illustré par la figure 4.2 on a exploité les trois mécanismes d'extension offerts par la méthode *lightweight*. Rappelons qu'on se place au niveau métamodèle (M2) et qu'on importe la superstructure UML [15] par un import MOF (M3) [9].

1. **Spécialisation des métaclasses** : Par de simples héritages (MOF), on étend des métaclasses importées d'UML. Les quatre paquetages héritent du concept `Package` provenant du noyau UML, et les trois dépendances héritent du concept `PackageImport`.
2. **Spécialisation des méta-associations** : Grâce au dispositif `subsets` (MOF) on spécialise les méta-associations d'UML. On a spécialisé à chaque fois le rôle `packageImport` qui associe en UML un `Package` (source) à un `PackageImport`.
3. **Surcharge des méta-associations** : Grâce au dispositif `redefines` (MOF) on redéfinit les méta-associations d'UML. On a redéfini à chaque fois le rôle `importedPackage` qui associe en UML un `PackageImport` à un `Package` (cible).

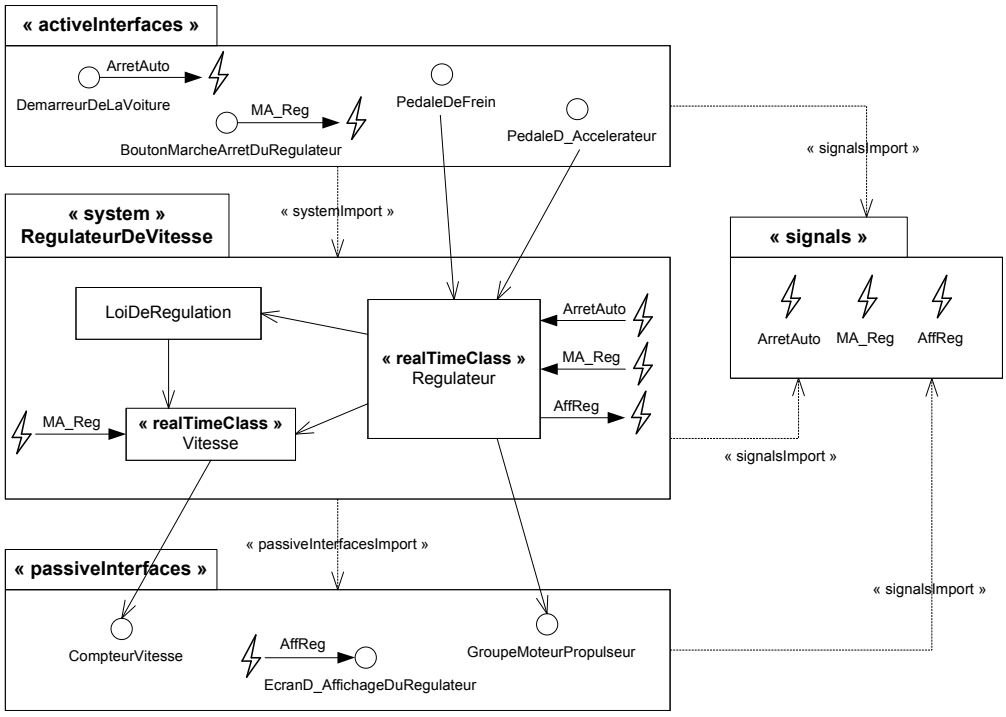


FIG. 4.3 – Régulateur de vitesse ACCORD

La figure 4.3 est un exemple représentatif d'un modèle structurel d'une application temps-réel ACCORD. Il s'agit du régulateur de vitesse [33] qu'on utilisera le long de cette section pour fournir des exemples aux concepts ACCORD introduits. On y retrouve clairement l'architecture ACCORD/UML séparant les quatre paquetages et on y voit également les imports spécialisés qui les lient.

1.2.2 Signaux

Métamodèle

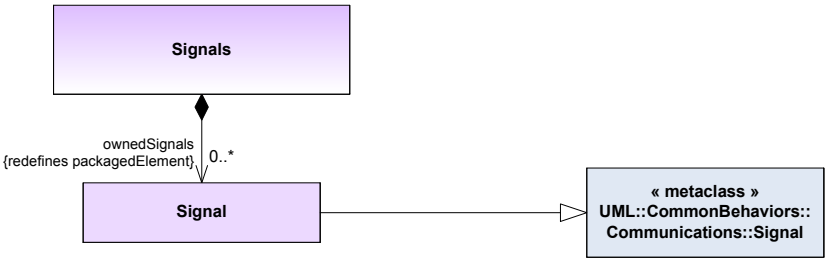


FIG. 4.4 – Signaux

Les objets de l'application peuvent communiquer de manière asynchrone par envoi de signal. Tous les signaux échangés doivent être spécifiés dans le paquetage `Signals`. En surchargeant son (méta-)rôle `packagedElement` de UML, on impose au paquetage `Signals` de ne contenir que des signaux ACCORD (règle de modélisation 4). Notons que le système d'extension via des profils ne nous permettait point d'exprimer une telle règle, seule une contrainte OCL qu'on vérifiait au niveau modèle nous aidait à pallier à cela.

Description des concepts

➢ ***Signal*** est un cas particulier du signal UML. Il s'agit, dans le modèle d'exécution ACCORD, d'une communication asynchrone de type diffusion. L'objet émetteur ne connait pas les objets cibles qui réceptionnent le signal et ces derniers ne connaissent pas non plus l'émetteur (règle 8). Notons que même si le concept ACCORD de `Signal` porte le même nom que la métaclasse UML qu'il spécialise, on ne peut confondre les deux car ils appartiennent à deux domaines (`Namespace`) différents.
Exemple : *ArretAuto*, *MA_Reg* et *AffReg*.

1.2.3 Interfaces passives

Métamodèle

La méthodologie ACCORD/UML structure l'interfaçage du système avec son environnement. Elle classifie les points d'interaction du système en interfaces actives (actionneurs) et interfaces passives (capteurs) (règle 2). Si un point d'interaction est à la fois actif et passif, il faut le décomposer en deux interfaces, une active et une passive (règle 3). En figure 4.5, on constate que le paquetage `PassiveInterfaces` ne peut contenir que des interfaces passives et qu'une `PassiveInterface` peut recevoir des signaux ACCORD.

Description des concepts

➢ ***PassiveInterface*** est un point d'interaction du système qui est uniquement destinataire de messages émis par ce système. En particulier, `PassiveInterface` peut recevoir les signaux (ACCORD) émis par les objets du système et déclarés auparavant dans le paquetage

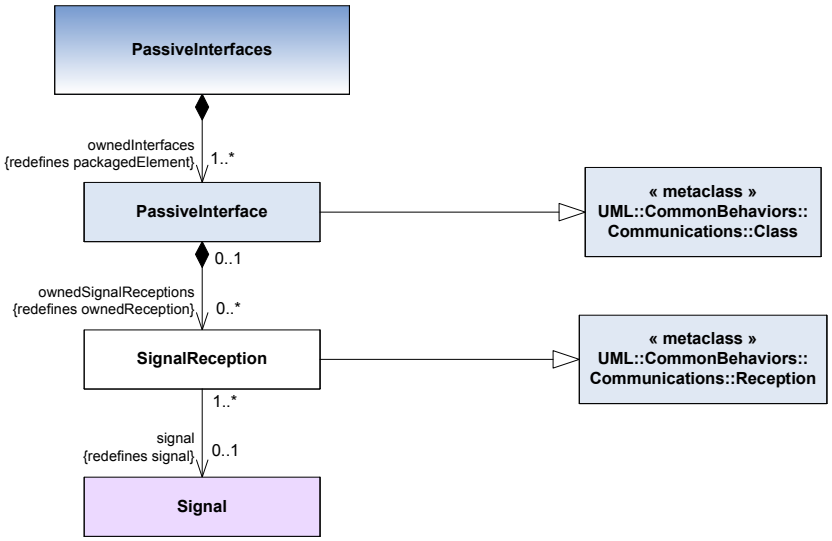


FIG. 4.5 – Interfaces passives

`Signals` (règle 11). En d'autres termes, il s'agit d'une sortie du système, d'un actionneur.
Exemple : *EcranD_AffichageDuRegulateur*, *CompteurVitesse* et *GroupeMoteurPropulseur*.

➢ ***SignalReception*** dénote la sensibilité à un `Signal`. Du point de vue sémantique, `SignalReception` ne diffère que légèrement du concept `Reception` d'UML qu'elle spécialise, car si cette dernière est sensible à n'importe quel `UML::Signal`, `SignalReception` n'est sensible qu'à un `ACCORD::Signal`. Contrairement aux extensions *heavyweight*, l'extension *lightweight* importe le métamodèle UML en lecture seule, on ne peut donc modifier `UML::Reception` pour la rendre uniquement sensible aux signaux ACCORD, on est donc obligé de créer une nouvelle métaclasse `SignalReception` avec une sorte de duplication par héritage.
Exemple : *EcranD_AffichageDuRegulateur* est sensible au signal *AffReg*.

1.2.4 Interfaces actives

Métamodèle

De manière générale, les objets d'une application ACCORD communiquent par envoi de messages. Dans ACCORD un envoi de message peut prendre deux formes soit un appel d'opération, soit un envoi de signal. Une *ActiveInterface* peut user des deux moyens pour stimuler le système (voir figure 4.6). Elle peut envoyer des signaux ACCORD énumérés par le méta-rôle (`tag definition`) `emittedSignals`. Notons ici que dans UML, rien ne permet au niveau structurel de déclarer qu'une classe est susceptible d'émettre un signal donné, on peut cependant le vérifier au niveau comportemental si la classe a une opération dont l'activité contient une action de type `SendAction` qui référence ce signal.

L'autre moyen de communication est l'appel d'opération, mais celui-ci exige un lien structurel (`UML::Association`) orienté et navigable de l'émetteur vers le récepteur. Si une `ActiveInterface` appelle des services de classes temps-réel qu'elle énumère par le biais du méta-rôle `/calledClasses`, ce rôle doit être dérivé des associations du modèle de l'application qui la lient aux `RealTimeClass`(s) (réceptrices et dont on verra la sémantique

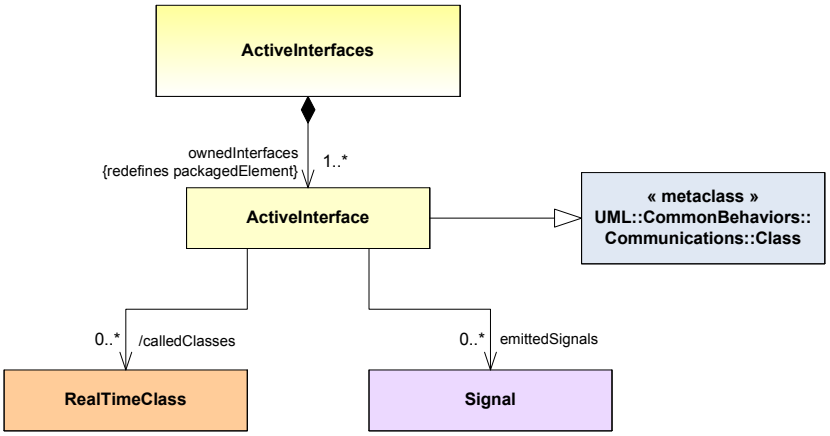


FIG. 4.6 – Interfaces actives

par la suite). A titre d'exemple, on fournit une règle OCL qui implémente cette dérivation :

```
context ActiveInterface
 inv: self.calledClasses->asSet() =
  uml::Association.allInstances()->iterate(a:uml::Association; acc:Set(uml::Type)=Set{} |
   if ((a.memberEnd->exists(p:uml::Property | p.type=self.base_class))
    and
     (a.memberEnd->exists(p:uml::Property | p.type.getAppliedStereotypes()
      ->exists(name = 'RealTimeClass'))))
   then
    acc->union(a.memberEnd.type->select(t:uml::Type | t.getAppliedStereotypes()
     ->exists(name = 'RealTimeClass'))->asSet())
   else
    acc
   endif)
```

Au niveau modèle, on itère sur toute association qui lie l'`ActiveInterface` à une `RealTimeClass` et on vérifie que cette `RealTimeClass` est bien renseignée dans la `tag defintion calledClasses`.

Description des concepts

➢ ***ActiveInterface*** est un point d'interaction qui est uniquement source de messages en direction du système. Une `ActiveInterface` peut émettre des signaux (`emittedSignals`) déclarés auparavant dans le paquetage `Signals` (règle 11). Comme elle peut stimuler des classes temps-réel du système (`/calledClasses`) en appelant leurs opérations. Autrement dit, il s'agit d'une entrée du système, d'un capteur.
Exemple : *DemarreurDeLaVoiture* émet le signal *ArretAuto*, *BoutonMarcheArretDuRegulateur* émet le signal *MA_Reg*, quand *PedaleDeFrein* et *PedaleD_Accelerateur* appellent directement la classe temps-réel *Regulateur*.

1.2.5 Système

Métamodèle

System est le coeur de l'application temps-réel, il se compose principalement d'un ensemble de classes UML dites `AccordClass`(s). Le modèle d'exécution ACCORD

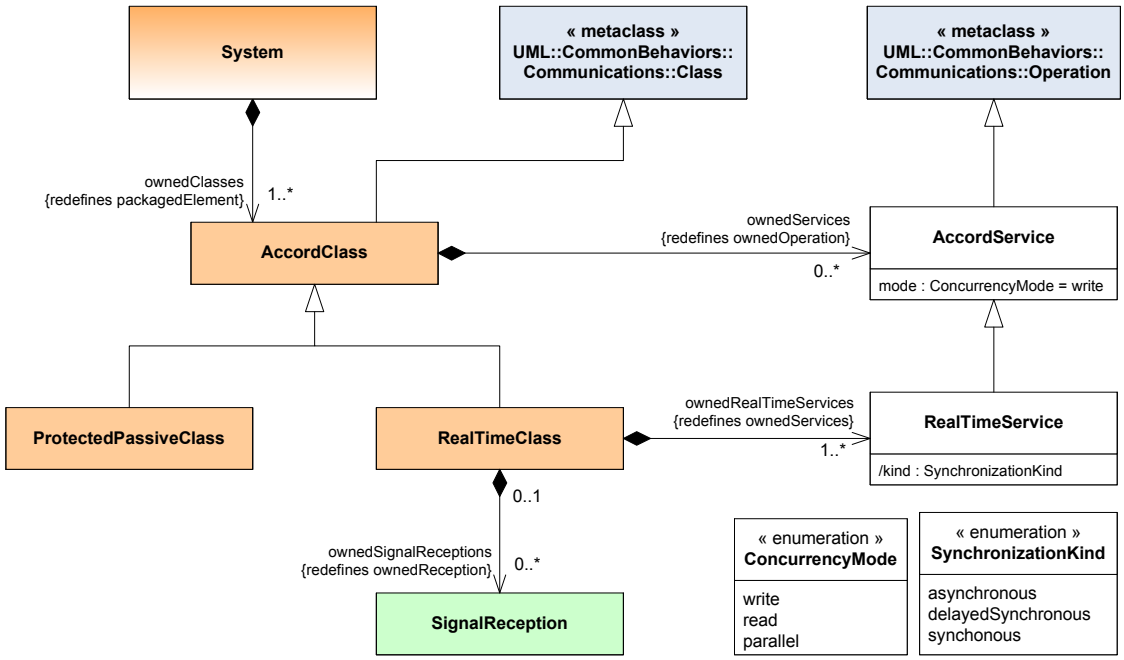


FIG. 4.7 – Système

distingue deux cas particuliers de classes : `RealTimeClass` et `ProtectedPassiveClass`. `RealTimeClass` est une classe active autonome munie d'une boite aux lettres et d'une structure de contrôle interne qui lui permet de gérer parallèlement plusieurs réceptions de signaux et plusieurs appels de services. `ProtectedPassiveClass` est par contre une classe passive qui risque d'être utilisée parallèlement par plusieurs classes temps-réel, elle implémente donc une protection contre les accès concurrents. Quand une classe n'est ni l'une ni l'autre, elle est considérée comme une simple `AccordClass` c-à-d qu'elle est passive et qu'elle n'est utilisée que par une seule autre classe du système.

Toute Classe ACCORD offre des services, un `AccordService` est une opération qu'on qualifie par son mode de concurrence (`ConcurrencyMode`). Un cas particulier de services sont les `RealTimeService`(s) exclusivement opérés par les classes temps-réel et qu'on caractérise par leur type de synchronisation (`SynchronizationKind`).

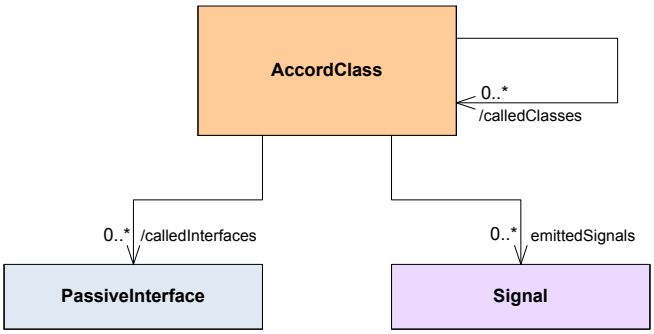


FIG. 4.8 – Classes ACCORD

On exprime dans la figure 4.8, le fait qu'une Classe ACCORD puisse éventuellement émettre des signaux (`emittedSignals`) et que de par ses associations on puisse

dériver les autres classes ACCORD (`/calledClasses`) et les interfaces passives (`/calledInterfaces`) dont elle appellent les opérations. Deux contraintes OCL semblables à celle fournie précédemment sont alors annexées au métamodèle.

Description des concepts

≻ ***AccordClass*** est un concept générique qui dénote toute classe du système possédant éventuellement des services. Une AccordClass peut émettre des signaux comme elle peut appeler les opérations des interfaces passives ou les services des autres classes du système auxquelles elle est associée. Par défaut, une `AccordClass` est passive et ses services ne sont sollicités que par une seule autre classe. En effet, quand elle est active, elle doit être stéréotypée en `RealTimeClass` et quand elle est passive mais référencée par plusieurs autres classes, elle doit être stéréotypée en `ProtectedPassiveClass`.
Exemple : *LoiDeRegulation* est typiquement une `AccordClass`. Elle appelle la classe temps-réel *Vitesse*.

≻ ***RealTimeClass*** est un concept central dans l'approche ACCORD. Elle dénote une classe active et autonome, qui est constituée d'une boite aux lettres servant de receptacle aux requêtes et d'un contrôleur de concurrence et d'état qui arbitre l'exécution des services temps-réel en fonction de l'état de la classe et des contraintes de concurrence. Seule une `RealTimeClass` peut gérer des appels de services asynchrones ou des réceptions de signaux (règle 12). Elle est ainsi la seule à pouvoir recevoir des stimulus depuis son environnement.
Exemple : *Regulateur* est une classe temps-réel recevant les signaux *ArretAuto* et *MA_Reg* et les appels de services temps-réel provenant des interfaces actives *PedaleDeFrein* et *PedaleD_Accelerateur*. *Regulateur* peut émettre le signal *AffReg*, contrôle l'interface passive *GroupeMoteurPropulseur* auquel il est associé et appelle les opérations des classes *LoiDeRegulation* et *Vitesse*.

≻ ***ProtectedPassiveClass*** est une classe passive, qui contrairement à une classe temps-réel n'est pas autonome. Elle est également protégée car ces services peuvent être sollicités en parallèle par plusieurs classes temps-réel et par conséquent sa protection s'impose. On exploite pour ce faire le dispositif `ConcurrencyMode`.

≻ ***ConcurrencyMode*** est un type énuméré qui spécifie les trois modes de concurrence d'un service ACCORD quelconque (règle 13) :
– **write** : si l'opération peut utiliser les attributs ou rôles de la classe en écriture.
– **read** : si l'opération utilise les attributs et les rôles de la classe en lecture seule
– **parallel** : si l'opération n'utilise ni rôles ni attributs.
Cette taxinomie couvre trivialement tous les cas.

≻ ***AccordService*** est une opération UML quelconque caractérisée par son mode de concurrence. Ce dernier est pris en compte pour garder la cohérence des données. Signalons que le modèle d'exécution ACCORD applique une politique nR1W (n Readers 1 Writer).

≻ ***SynchronizationKind*** est un type énuméré qui caractérise le type de synchronisation engendré par l'appel d'un service temps-réel (règle 14). La valeur est déduite de la signature de l'opération en question :
– **asynchronous** : quand l'opération n'a ni valeur de retour, ni paramètre de sortie (*out* ou *inout*)

– **delayedSynchronous** : quand l'opération n'a pas de valeur de retour mais possède au moins un paramètre de sortie. La synchronisation est différée.
– **synchronous** : quand l'opération possède une valeur de retour. L'appel est bloquant.

Cette taxinomie couvre trivialement tous les cas.

➢ ***RealTimeService*** est un cas particulier d'un service ACCORD. Il s'agit d'une opération publique appartenant à une classe temps-réel. Cette dernière prend en compte son type de synchronisation afin d'ordonnancer les exécutions de requêtes en évitant d'éventuels inter-blocages. Signalons qu'un service non temps-réel (`AccordService`) est par essence de type synchrone car il appartient à une classe passive, il utilise de ce fait le fil d'exécution de la classe active qui l'appelle.

1.3 Conclusion

Dans cette section, on a utilisé une extension *lightweight* d'UML pour créer un métamodèle qui nous permet de décrire succinctement la structure d'une application temps-réel conçue grâce à la méthodologie ACCORD/UML. Ce métamodèle nous habilite à identifier les entités logicielles de l'application temps-réel ACCORD grâce notamment à une dizaine de stéréotypes. Il permet également d'identifier les dépendances qui existent entre ces différentes entités (ex : appel de service, envoi de signal). La prise en charge de ces deux aspects (entités + dépendances) est indispensable à l'élaboration d'une allocation de notre application sur du matériel.

2 Allocation logiciel/matériel

L'allocation est l'opération qui fait correspondre les entités logicielles aux ressources matérielles. Également dit placement, Elle consiste à placer chaque entité logicielle sur une ressource ou une collaboration de ressources matérielles qui supporteront son exécution. L'allocation est l'étape phare du processus d'intégration qui vient unifier les deux flots de conception logiciel et matériel. Elle peut être réduite à une simple formalité si la communication entre les flots parallèles logiciel et matériel fût parfaite, comme elle peut remettre en cause tout le développement réalisé en cas d'inadéquation entre les deux composantes, elle engendre ainsi une lourde itération coûteuse en temps et en argent. L'allocation est donc une étape ultime, test et verdict.

On a défini dans le chapitre 3 en section 1, le langage de modélisation HRM dont le premier cas d'utilisation énuméré en page 48, est la modélisation à un haut niveau d'abstraction de l'architecture matérielle pour influencer la conception du logiciel et entamer l'allocation. Parallèlement, on a défini dans la section précédente un métamodèle ACCORD qui nous permet de décrire, toujours à un haut niveau d'abstraction, la structure (entités et dépendances) d'une application temps-réel conçue grâce à la méthodologie ACCORD/UML. A partir de ces deux abstractions du logiciel et du matériel, on définira du point de vue structurel un ensemble de règles d'allocation depuis lesquelles on pourra déterminer et explorer différentes configurations de placement logiciel/matériel.

2.1 Contexte

Avant de fournir la définition des règles d'allocation, commençons par déterminer notre contexte et l'ensemble de postulats sur lesquels nous nous basons.

Nature et mécanisme d'allocation

En plus d'être principalement structurelle, l'allocation que nous signifions ici, est complètement statique dans le sens où elle est établie à l'avance et figée avant toute exécution du système. En effet, le caractère temps-réel et critique des systèmes embarqués considérés, nous oblige à garantir les temps d'exécution et les espaces de mémoire nécessaires et à éliminer les incertitudes liées à la dynamique. Il est par exemple interdit de migrer dynamiquement une tâche d'une ressource matérielle à une autre ou de créer des données durant l'exécution. Pour exprimer ce placement statique, nous utiliserons un mécanisme basique que nous retrouvons dans SysML et MARTE, il s'agit de la relation unidirectionnelle `«allocate»` qui lie une source logicielle à sa cible matérielle (voir figure 4.9).

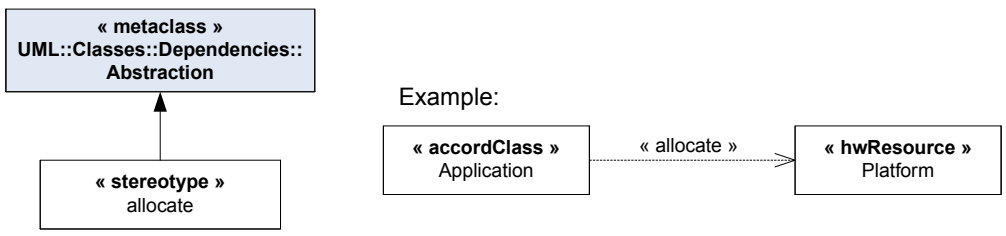


FIG. 4.9 – Mécanisme d'allocation

Nature de l'application temps-réel

Dans notre cas, le modèle de l'application temps-réel est conçu grâce à la méthodologie ACCORD/UML. Il doit être consistant et conforme par sa structure au métamodèle ACCORD qu'on a défini précédemment. L'instanciation des classes ACCORD doit être également statique pour les mêmes raisons avancées ci-dessus. Par souci de rigueur, on ne peut instancier dynamiquement un objet d'une classe car cela peut entrainer entre autres de l'indéterminisme dans le comportement de la mémoire. On en déduit que quand les classes ACCORD ne sont pas des classes singleton, ce qui n'est pas souvent le cas, leur multiplicité est de valeur $n > 1$ finie et prédéterminée statiquement et que l'instanciation des n objets se fait exclusivement au démarrage de l'application. Dans ce schéma particulier à caractère structurel et statique, on se permet de confondre les concepts classe et objet. En effet, cette assimilation est triviale dans le cas d'une classe singleton et si la classe est de multiplicité finie $n > 1$, il suffira de la dupliquer en n classes singleton de multiplicité 1 pour se ramener au premier cas.

Nature de l'architecture matérielle

L'architecture matérielle doit être à son tour consistante et conforme au métamodèle HRM. Elle doit être de préférence conçue grâce à la méthodologie de modélisation du matériel définie en section 2 du chapitre 3. On suppose que la plateforme matérielle est distribuée en un ensemble de noeuds de calcul. Sachant qu'on ne peut déployer l'infrastructure ACCORD que sur un système d'exploitation (ex : Linux, VxWorks), on introduit le concept de `Node`[1] qu'on considère comme une collaboration de ressources matérielles capables de supporter l'exécution d'un système d'exploitation et par conséquent celle de l'infrastructure ACCORD. Un noeud de calcul peut être mono-processeur ou SMP (multi-processeurs à mémoire partagée). On le représente par une `Collaboration` UML illustrée en figure 4.10. La collaboration `Node` est donc composée de n processeurs (*CPU*) connectés au même bus système (*FSB*) et partageant via ce dernier m mémoires (*SharedMemory*). Un noeud mono-processeur correspond au cas particulier où $n = 1$.

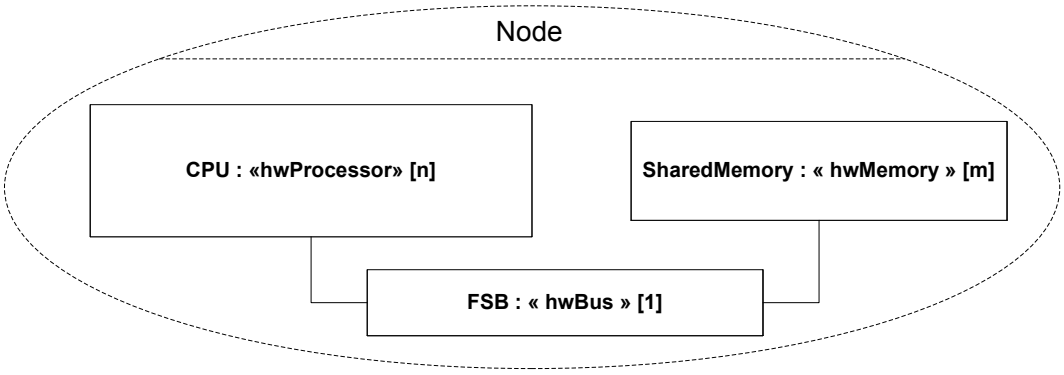


FIG. 4.10 – Noeud matériel

Une plateforme matérielle contient un ensemble $N \subset R$ de noeuds de cardinalité $k \geq 1$

[1]Le concept `Node` existe dans UML avec une sémantique large, il s'agit d'une quelconque cible matérielle de calcul.

tel que R est l'ensemble de toutes les ressources de la plateforme (voir définition page 92). Chaque noeud est conforme au patron `Node`. On dit que deux ressources et en particulier deux noeuds $node1$ et $node2$ sont connectés et on note ($node1 \rightleftharpoons node2$), si et seulement si ils jouent les rôles de `r1` et `r2` dans une occurrence de la collaboration `Connection` illustrée dans la figure 4.11. On y trouve les ressources `r1` et `r2` connectées via une ressource stéréotypée `HwMedia`.

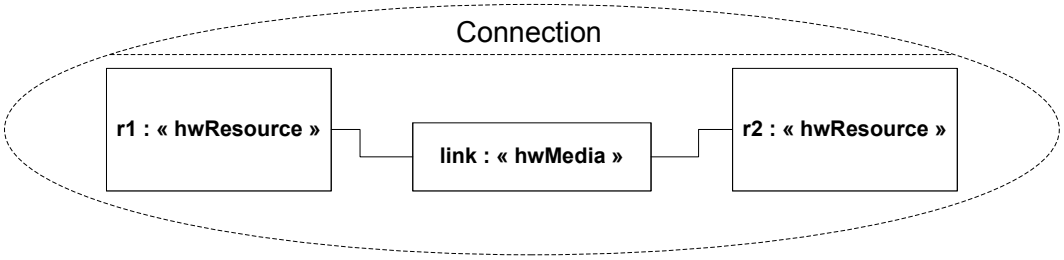


FIG. 4.11 – Ressources connectées

On définit la relation «*communique avec*» que nous notons $\leftrightarrow$ comme la clôture reflexive de $\rightleftharpoons$ tel que $\leftrightarrow = \rightleftharpoons \cup Id_R$, puisqu'on considère qu'une ressource communique aussi avec elle même. Notons que Id_R est la relation identité sur l'ensemble des ressources R.

Remarque : Dans les figures précédentes (4.10,4.11) ainsi que dans plusieurs figures de cette section d'allocation, on spécifie parfois que le stéréotype sans nommer le type auquel il est appliqué (ex : CPU :«hwProcessor» en figure 4.10), on signifie de cette manière (graphiquement) une contrainte sur la nature du type. Si on veut rester entièrement conforme à UML, une solution serait de définir des règles OCL équivalentes au niveau de chaque rôle de la collaboration, vérifiant que les types jouant ce rôle dans le modèle appliquent bien le stéréotype en question.

2.2 Adaptation de l'infrastructure ACCORD

L'infrastructure ACCORD est une couche logicielle (intergiciel) qui s'intercale entre l'application et le système d'exploitation. Elle a deux rôles principaux, d'un côté elle implémente le modèle d'exécution ACCORD en prenant le contrôle du système d'exploitation et de l'autre elle abstrait l'application temps-réel ACCORD/UML de toute dépendance vers un système d'exploitation ou un matériel particuliers. A l'origine, le modèle d'exécution ACCORD et son implémentation (infrastructure ACCORD) ciblaient l'exécution sur une architecture mono-noeud et mono-processeur c-à-d que ni les plateformes distribuées (plusieurs noeuds de calcul) ni les noeuds SMP (plusieurs processeurs à mémoire partagée dans un même noeud) n'étaient supportés. On a donc remédié à cela par deux mises à jour majeures.

– Adaptation aux noeuds SMP : Le modèle d'exécution ACCORD est basé sur un contrôleur global d'ordonnancement appliquant une politique EDF. Dans le cas d'un déploiement sur un mono-processeur, ce contrôleur lance une tâche à la fois et commence par la plus urgente c-à-d celle qui a la deadline la plus proche. Dans le cas du déploiement sur un noeud SMP (`Node`) avec n processeurs, il a suffi de modifier dans ce contrôleur le système de lancement des tâches afin de garantir, en permanence,

l'exécution des n tâches les plus urgentes à condition qu'elles soient non concurrentes.

– Adaptation aux plateformes distribuées : L'infrastructure ACCORD implémente un dispositif `Channel` qui permet d'échanger des messages (envoi de signal ou appel d'opération) entre différentes zones de l'infrastructure ACCORD[2]. Il a fallu étendre le dispositif `Channel` pour implémenter des échanges de messages entre noeuds via des `sockets`. En bref, il est maintenant possible de communiquer entre différentes instances de l'infrastructure ACCORD déployées sur des noeuds différents avec des systèmes d'exploitation différents, à condition que ces noeuds soient connectés entre eux selon le schéma expliqué précédemment.

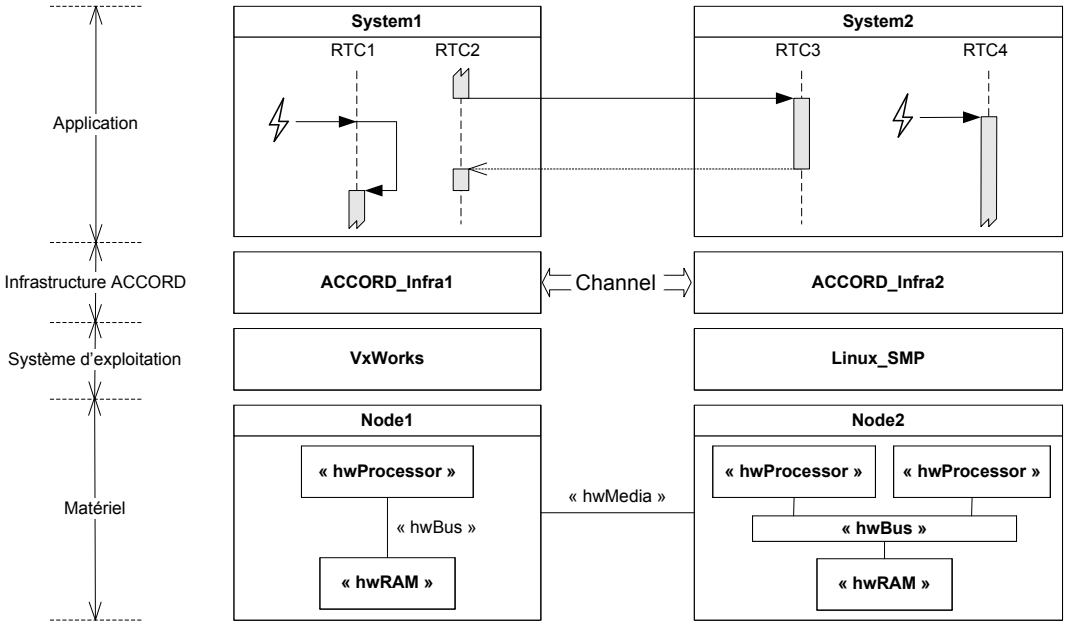


FIG. 4.12 – Exemple de système distribué hétérogène

Ainsi adaptée, l'infrastructure ACCORD peut être aisément déployée sur les architectures matérielles qu'on prend en charge dans le contexte de cette section. En effet, grâce à ces deux mises à jour, il est désormais possible de développer en ACCORD/UML des systèmes temps-réel distribués. La figure 4.12 représente un exemple de ce type de systèmes. La plateforme matérielle y est distribuée et hétérogène puisqu'elle se compose de deux noeuds, l'un est mono-processeur et l'autre a deux processeurs. On déploie dessus séparément deux instances de l'infrastructure ACCORD, puis on sépare l'application en deux parties, on alloue les deux classes temps-réel *RTC1* et *RTC2* sur *Node1* et *RTC3* et *RTC4* sur *Node2*. On constate qu'un appel synchrone de service entre *RTC2* et *RTC3* s'opère via le `Channel` et que grâce à ses deux processeurs, *Node2* peut maintenir simultanément deux fils d'exécution contrairement à *Node1*.

2.3 Elaboration de l'allocation

Après avoir déterminé le contexte dans lequel on évolue, on propose d'entamer l'allocation structurelle de l'application temps-réel sur la plateforme matérielle. Pour ce faire,

[2]Une zone peut être assimilée à un `process` Linux, elle a donc son propre espace mémoire

on va procèder en trois étapes :

1. Dans un premier temps, on ne prendra pas en compte les dépendances qu'il peut y avoir entre les entités logicielles et on considèrera qu'elles sont toutes indépendantes. On donnera, dans ce schéma, les règles d'allocation qui pour chaque concept ACCORD désignent les ressources matérielles qui peuvent l'accueillir.
2. Ensuite, on prendra en compte les dépendances qui lient, entre elles, les classes de l'application. On spécifiera de manière formelle les contraintes que le matériel doit satisfaire pour supporter ces dépendances indispensables à l'exécution du logiciel.
3. On fournira finalement des mécanismes d'adéquation du logiciel qui assoupliront les contraintes d'allocation générées précédemment par ces dépendances intra-logiciel.

2.3.1 Règles d'allocation

Une opération d'allocation consiste à lier par une relation `«allocate»`, une classe de l'application à une ressource du matériel (voir figure 4.9). Une règle d'allocation se veut plus générique et constructive, elle consiste à lier un concept ACCORD à un concept HRM. On dénote par une règle d'allocation qu'un type d'entités logicielles (ex : `realTimeClass`, `activeInterface`) devra être déployé sur tel type de ressources matérielles (ex : `HwProcessor`, `HwSensor`). En bref, Toute opération d'allocation doit appliquer une des règles d'allocation.

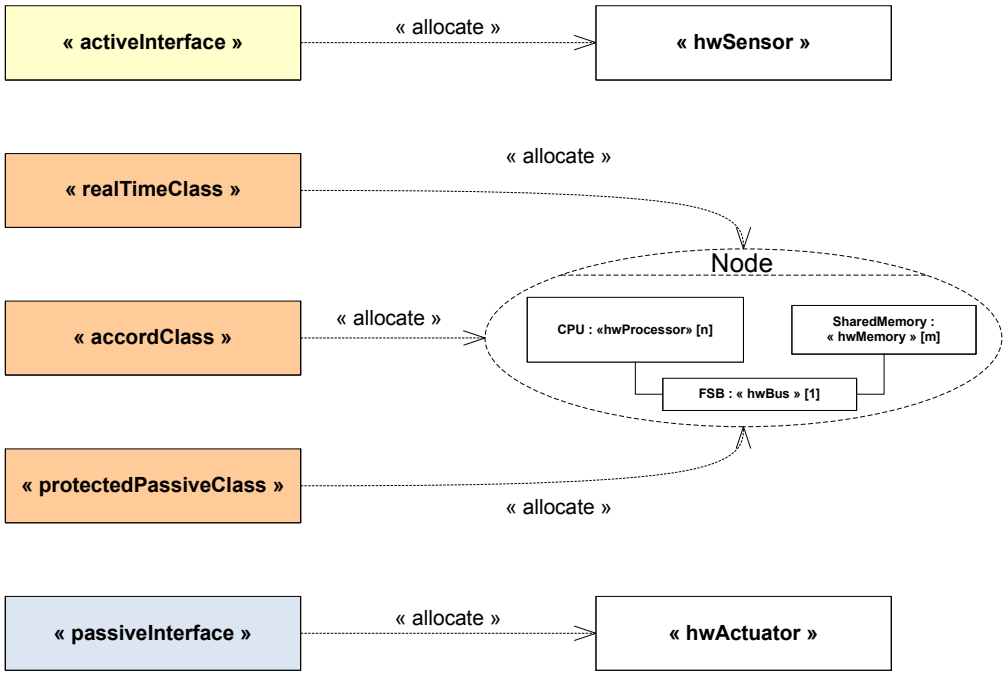


FIG. 4.13 – Règles d'allocation

La figure 4.13 rassemble toutes les règles d'allocation. Dans ACCORD, on distingue cinq types de classes c-à-d cinq stéréotypes applicables à des classes, ils sont représentés en couleur dans la figure 4.13. La définition dans ACCORD de `ActiveInterface` et `PassiveInterface` nous impose de les allouer respectivement à `HwSensor` et `HwActuator`. `RealTimeClass` est une entité logicielle active, elle est allouée sur un noeud

entier car elle a besoin d'une ressource de calcul pour exécuter ses services et d'une mémoire pour stocker ses données. `AccordClass` et `ProtectedPassiveClass` sont cependant des classes passives qui n'ont pas de fils d'exécution propres à eux[3], elle sont donc considérés comme de simples données, néanmoins, elles sont également allouées à des noeuds où des classes actives les manipulent.

2.3.2 Contraintes d'allocation

Les règles d'allocation précédentes allouent dans l'absolu les classes de l'application sans prendre en compte leurs dépendances. Ces dépendances se répercutent en contraintes matérielles indispensables au fonctionnement. En bref, si deux entités logicielles échangent des messages, on ne peut les allouer que sur un même noeud ou sur deux noeuds qui sont connectés physiquement. Dans la figure 4.14, on montre un exemple représentatif d'un système hétérogère. D'un côté, on a le logiciel en ACCORD/UML et de l'autre, on a le matériel en HRM. On fournit dans la figure un schéma d'allocation envisageable puisqu'il vérifie trois contraintes :

- La dépendance entre *RTC1* et *RTC2* est réalisable car *Node1* et *Node2* communiquent matériellement.
- L'envoi de signal entre *AI* et *RTC1* est également envisageable car *Sensor* et *Node1* sont connectés.
- *RTC2* et *AC* sont allouées à un même noeud. On rappelle que *AC* est une classe passive dont les services s'exécutent sur le fil d'exécution de l'appelant en l'occurrence *RTC2*. Par conséquent, *RTC2* et *AC* doivent appartenir au même noeud.

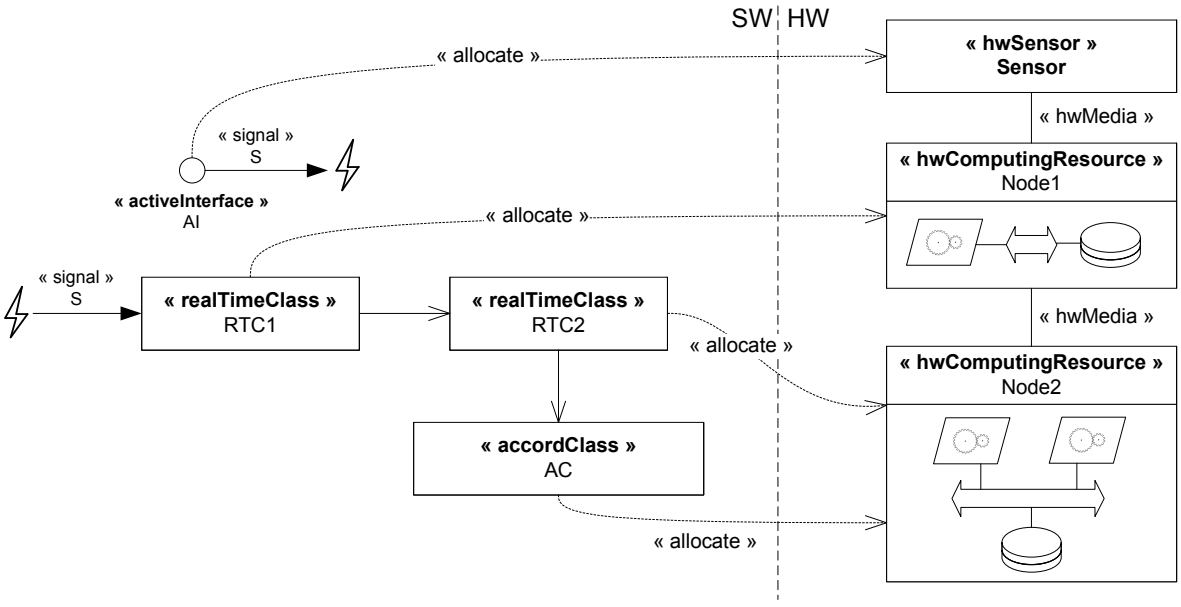


FIG. 4.14 – Exemple d'allocation de classes dépendantes

Une dépendance entre classes correspond à un éventuel envoi de message. Un envoi de message dans ACCORD correspond à son tour à un appel d'opération ou un envoi de signal. On va énumérer toutes les dépendances possibles dans le cadre d'une application

[3] leurs opérations s'exécutent sur le fil d'exécution de l'appelant

ACCORD, et on les organisera en deux parties, la première pour les appels d'opérations et la seconde pour les diffusions de signaux. On spécifiera formellement la contrainte matérielle induite pour chaque type de dépendance. Pour une meilleure lisiblité, on formulera ces contraintes en conditions algébriques que nous jugeons plus simples à lire que les formules OCL équivalentes.

On note C l'ensemble des classes de l'application temps réel ACCORD avec AI, AC, RTC, PPC et PI des sous-ensembles de C tel que :
$AI = \{c \in C \mid c.isStereotyped(ActiveInterface)\}$
$AC = \{c \in C \mid c.isStereotyped(AccordClass)\}$
$RTC = \{c \in C \mid c.isStereotyped(RealTimeClass)\}$
$PPC = \{c \in C \mid c.isStereotyped(ProtectedPassiveClass)\}$
$PI = \{c \in C \mid c.isStereotyped(PassiveInterface)\}$
La définition des stéréotypes ACCORD est telle que leur application sur une classe est obligatoire et exclusive. On en déduit que (AI, AC, RTC, PPC, PI) est une partition de C. Ensuite, on note $allocatedTo()$ l'opération qui va de C vers R et qui associe à toute classe $c \in C$, la ressource matérielle $c.allocatedTo() \in R$ sur laquelle c est allouée.

Appels d'opérations

Commençons avec les envois de messages par appels d'opérations. On peut appeler les services de classes temps-réel, de classes passives ou d'interfaces passives.

[C1]. **Appel *de* ActiveInterface, RealTimeClass, AccordClass ou ProtectedPassiveClass *vers* RealTimeClass** :
$(\forall\, c \in AI \cup RTC \cup AC \cup PPC)(\forall\, rtc \in RTC)$,

$$rtc \in c.calledClasses() \Longrightarrow c.allocatedTo() \leftrightarrow rtc.allocatedTo()$$

Tout appel d'opération vers une classe temps-réel nécessite que la ressource matérielle allouée à l'appelant communique avec celle allouée à la classe temps-réel appelée. La relation «*communique avec*» notée $\leftrightarrow$ est définie en page 128, alors que l'opération $calledClasses()$ (dans la prémisse), rend les valeurs du rôle homonyme en figures 4.6 et 4.8.

[C2]. **Appel *de* RealTimeClass, AccordClass ou ProtectedPassiveClass *vers* AccordClass ou ProtectedPassiveClass** :
$(\forall\, c \in RTC \cup AC \cup PPC)(\forall\, pc \in AC \cup PPC)$,

$$pc \in c.calledClasses() \Longrightarrow c.allocatedTo() = pc.allocatedTo()$$

Quand une classe appelle un service d'une classe passive, cette dernière doit appartenir au même espace d'adressage. Ainsi, la classe appelant peut accéder aux données et exécuter les opérations de la classe passive. Par conséquent, la classe appelant et la classe passive appelée doivent faire partie du même noeud.

[C3]. **Appel *de* RealTimeClass, AccordClass ou ProtectedPassiveClass *vers* PassiveInterface** :
$(\forall\, c \in RTC \cup AC \cup PPC)(\forall\, pi \in PI)$,

$$pi \in c.calledInterfaces() \Longrightarrow c.allocatedTo() \leftrightarrow pi.allocatedTo()$$

Toute classe du système peut contrôler une interface passive via un appel d'opération. La seule réserve est que cette dernière soit allouée à une sortie matérielle (`HwActuator`) qui puisse communiquer avec le noeud accueillant la classe appelant.

Envois de signaux

Selon le modèle d'exécution ACCORD, seules les classes temps-réel et les interfaces passives peuvent recevoir des signaux.

[C4]. **Envoi *de* ActiveInterface, RealTimeClass, AccordClass ou ProtectedPassiveClass *vers* RealTimeClass** :
$(\forall\, c \in AI \cup RTC \cup AC \cup PPC)(\forall\, rtc \in RTC)$,

$$(c.emittedSignals() \cap \{s \mid (\exists\, r \in rtc.ownedSignalReceptions()), s = r.signal()\} \neq \phi) \\ \Longrightarrow c.allocatedTo() \leftrightarrow rtc.allocatedTo()$$

Pour qu'un envoi de signal soit intercepté par une classe temps réel qui lui est sensible, il faut vérifier que la ressource allouée à l'entité logicielle diffusant le signal, communique avec le noeud alloué à la classe temps-réel recevant ce signal. Par la prémisse, on cherche s'il existe un signal émis auquel la classe temps-réel a une `SignalReception` (voir figures 4.5 et 4.7).

[C5]. **Envoi *de* RealTimeClass, AccordClass ou ProtectedPassiveClass *vers* PassiveInterface** :
$(\forall\, c \in RTC \cup AC \cup PPC)(\forall\, pi \in PI)$,

$$(c.emittedSignals() \cap \{s \mid (\exists\, r \in pi.ownedSignalReceptions()), s = r.signal()\} \neq \phi) \\ \Longrightarrow c.allocatedTo() \leftrightarrow pi.allocatedTo()$$

Une classe du système peut stimuler une interface passive via un envoi de signal. Toutefois, il faut que cette dernière soit allouée à une sortie matérielle (`HwActuator`) communiquant avec le noeud qui accueille la classe diffusant le signal.

2.3.3 Adéquation de l'allocation

Grâce aux règles d'allocation définies, nous pouvons générer un grand nombre de configurations de placement logiciel/matériel en explorant toutes les combinaisons. Imaginons que mis à part les interfaces, on a n classes (temps-réel ou passive) dans la partie système de l'application ACCORD et k noeuds dans notre architecture matérielle, on pourra donc explorer jusqu'à k^n configurations. Cependant, beaucoup de configurations seront automatiquement éliminées si elles ne valident pas les cinq contraintes d'allocation définies. Notre objectif est de fournir des mécanismes d'adéquation du modèle de l'application temps-réel qui pourront rendre réalisable toute configuration à priori non-valide. Ces mécanismes d'adaptation ne remettent pas en cause l'architecture logicielle établie, il s'agit de modifications post-développement logiciel qui sont simples à mettre en oeuvre et parfaitement automatisables.

On propose, en tout, deux adaptations du modèle d'application qui viennent satisfaire une à une les contraintes d'allocation. Nous les nommons mécanisme de *classe relai* et mécanisme d'*activation de classe*.

Classe relai

Sur les cinq contraintes d'allocation, les contraintes C1 et C4 concernent l'envoi de message (appel d'opération et envoi de signal) vers une classe temps-réel, et les contraintes C3 et C5 concernent l'envoi de message vers une interface passive. Ces quatre contraintes exigent que les entités logicielles $e1, e2 \in C$ respectivement source et cible du message, soient allouées à deux ressources de la plateforme qui communiquent du point de vue matériel, c-à-d selon la collaboration illustrée en figure 4.11. On note $e1.allocateTo() \leftrightarrow e2.allocatedTo()$.

Quand une architecture matérielle est sous forme d'étoile dans le sens où il existe un réseau commun à tous les noeuds et interfaces, et qu'ils peuvent communiquer directement de l'un à l'autre, alors ces contraintes de communication matérielle sont trivialement vérifiées. Alors que si l'architecture matérielle est plutôt sous forme de chaîne, ces contraintes deviennent restrictives. On retrouve une telle architecture dans l'exemple précédent de la figure 4.14 où l'interface active *AI* envoie des signaux vers la classe temps-réel *RTC1*. *AI* est impérativement allouée à *Sensor*, on ne peut donc allouer *RTC1* sur *Node2* (à la place de *Node1*) car il n'y a pas de communication physique directe entre *Sensor* et *Node2*. On propose de remédier à cela via une classe relai qu'on crée puis on alloue sur *Node1*.

La figure 4.15 illustre l'élaboration de l'adéquation du logiciel via le mécanisme de classe relai. Au départ on a deux classes temps-réel dont l'une appelle l'autre et trois noeuds connectés en chaîne. Avec nos règles d'allocation on peut avoir $3^2 = 9$ configurations de placement. Toutes les configurations vérifient les contraintes d'allocation sauf deux, celles qui allouent séparément *RTC1* et *RTC2* sur *Node1* et *Node3* car ces deux noeuds ne communiquent pas directement. L'astuce est dès lors de créer une classe relai *Relay* qui s'intercale entre les deux classes temps-réel et qu'on alloue sur *Node2*. Cette classe est très facile à implémenter puisqu'elle a pour seul rôle la transmission des appels d'opération de l'une vers l'autre (de *RTC1* vers *RTC2*).

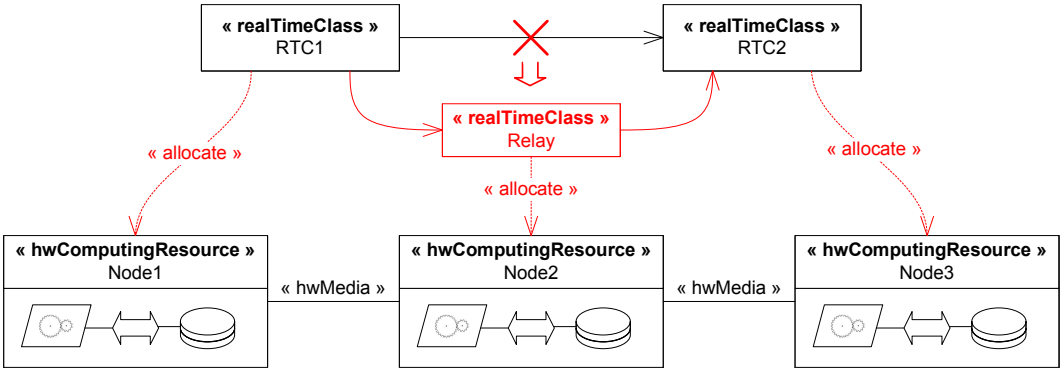


FIG. 4.15 – Mécanisme classe relai

Le mécanisme de classe relai peut être appliqué autant de fois que nécessaire pour lier deux entités logicielles échangeant des messages quand elles sont allouées à deux ressources éloignées dans la chaine des noeuds. On a, dans ce cas, une séquence de classes relai. Par conséquent, la condition de communication ($e1.allocateTo() \leftrightarrow e2.allocatedTo()$) se retrouve simplifiée à ($e1.allocateTo() \leftrightsquigarrow e2.allocatedTo()$) avec $\leftrightsquigarrow$ est la clôture transitive de $\leftrightarrow$ tel que ($\leftrightsquigarrow = \bigcup_{n\geq 1} \leftrightarrow^n$). La relation binaire $\leftrightsquigarrow$ dénote l'existence d'un chemin, entre deux ressources, à travers une chaîne de noeuds. Si on ajoute à cela que l'existence d'un tel

chemin est toujours vraie dans le cadre des architectures matérielles que nous considérons, puisque le cas contraire signifiera que la plateforme matérielle se compose de deux parties complètement indépendantes, la condition de communication n'en est donc plus une.

Pour conclure, on peut pallier aux restrictions engendrées par quatre contraintes d'allocation C1, C3, C4 et C5 avec un simple mécanisme d'adéquation logicielle dit classe relai. La classe relai est temps-réel, elle permet ainsi de prendre en compte des contraintes de temps lors de la communication. Toutefois, dans des cas précis, on peut imaginer d'autres solutions implémentées directement dans les systèmes d'exploitation accueillant l'infrastructure ACCORD, comme par exemple des tables de routage. Il est tout aussi évident qu'une solution matérielle via une liaison physique entre les ressources concernées, peut également résoudre le problème.

Activation de classe

Le second mécanisme d'adéquation, dit activation de classe, permettra de lever la restriction engendrée par la contrainte d'allocation C2 restante. Cette contrainte vérifie que l'appel de service d'une classe passive qu'elle soit une AccordClass ou une ProtectedPassiveClass, ne peut s'opérer que si cette classe appartient au noeud de l'appelant. Dans l'exemple illustré en figure 4.14, on voit que *RTC2* et *AC* appartiennent au même noeud *Node2*.

Quand la classe passive est de type AccordClass c-à-d qu'elle n'est pas sujette à accès multiples, il est normal qu'elle soit allouée au noeud de son unique appelant[4]. Par, ailleurs, quand la classe passive est de type ProtectedPassiveClass, cela signifie qu'elle est sujette à des accès multiples et concurrents. La figure 4.16 montre un tel cas de figure. La contrainte d'allocation C2 implique, dans cette configuration, que la classe passive *PPC* ainsi que les appelants *RTC2*, *RTC3*, *AC* et par transitivité *RTC1*, soient tous alloués au même noeud. Cette fois la restriction devient préjudiciable. En effet, il se peut qu'aucun noeud de l'architecture matérielle ne puisse supporter trois classes temps-réel actives du point de vue performance.

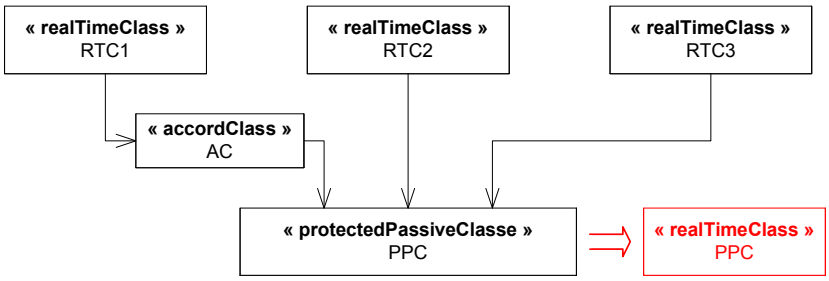


FIG. 4.16 – Mécanisme d'activation de classe

Le mécanisme d'activation de classe remédie à cela. Il consiste à transformer la classe passive protégée en classe temps-réel. Cette opération est très simple à réaliser puisqu'il suffit de retirer au niveau du modèle de l'application, le stéréotype `ProtectedPassiveClass` et appliquer le stéréotype `RealTimeClass`. Par cette transformation en classe temps-réel, la contrainte d'allocation C2 se trouve transformée en contrainte d'allocation C1 concernant

[4]On peut envisager de l'allouer ailleurs (sur un autre noeud) si cette classe passive occupe une grande taille mémoire non supportée par le noeud local (de l'appelant)

les appels de services temps-réel. C1 est trivialement plus souple et peut être éventuellement satisfaite par le mécanisme de classe relai introduit plus haut. Revenons à notre exemple en figure 4.16, on obtient après l'élaboration du mécanisme l'activation de classe, quatre classes temps-réel qu'on peut allouer séparément sur différents noeuds.

2.4 Conclusion

L'allocation est une première étape dans le processus de validation, elle succède aux diverses communications entre flots de conception et précède les phases d'analyse, de simulation et de test validant le système. Il existe deux manières d'utiliser l'ensemble des règles, des contraintes et des adéquations de l'allocation qu'on a définies : une constructive et une vérificatrice. La manière constructive consiste à utiliser ces mécanismes d'allocation afin de générer toutes les configurations possibles de placement logiciel/matériel en vue de les analyser, les tester et en élire une qui remplit tous les critères. Pendant que la façon vérificatrice, ne vient que valider, rendre adéquat ou réfuter un schéma d'allocation proposé par un architecte système ou pré-établi lors du développement conjoint logiciel/matériel.

3 Processus de validation

Pour compléter le processus de développement de systèmes embarqués hétérogènes, on expose sommairement notre vision du flot de validation qui suit l'intégration des deux composantes logicielle et matérielle. Ce processus est représenté en figure 4.17.

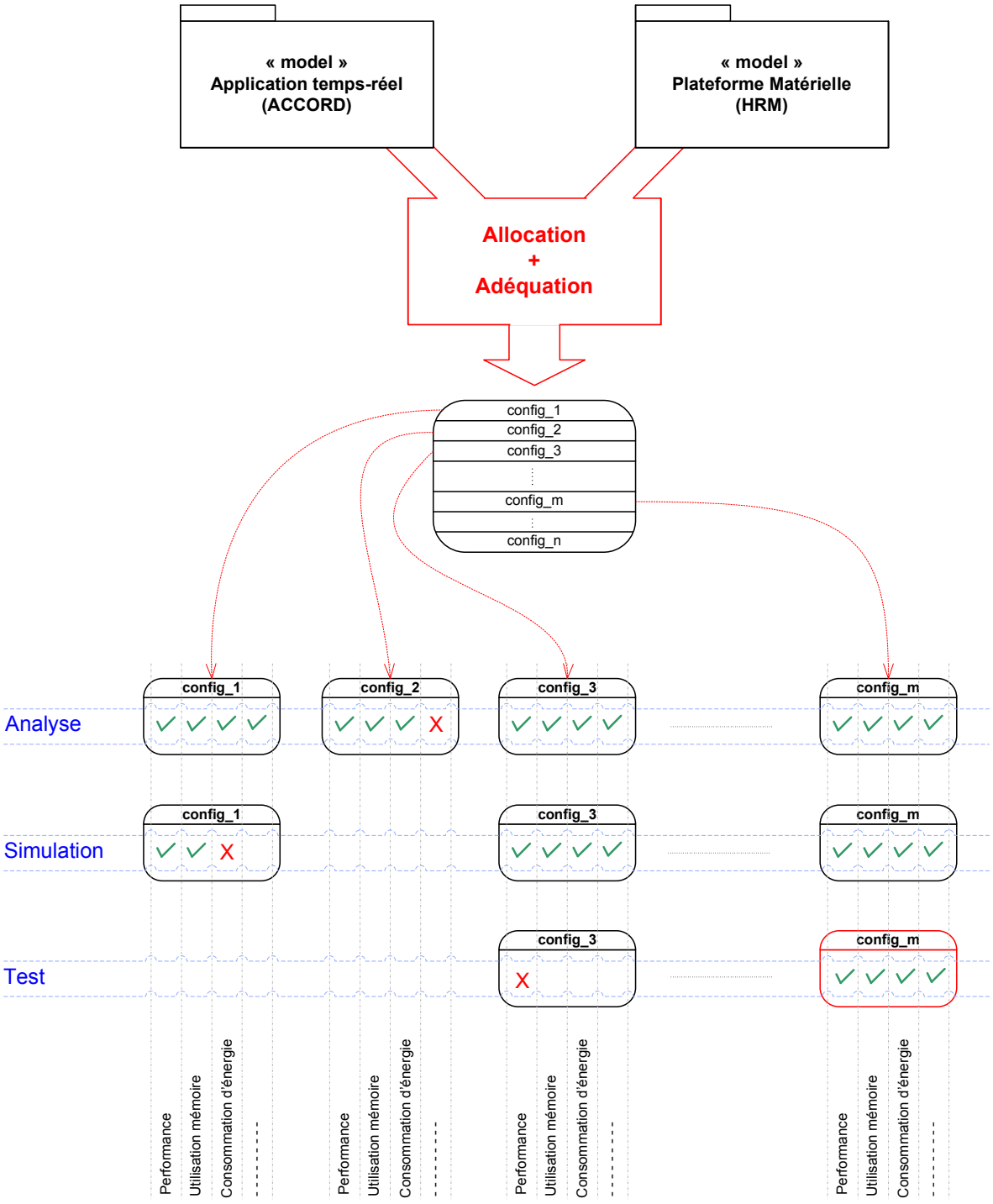


FIG. 4.17 – Processus de validation

3.1 Vue générale

Les règles d'allocation spécifiées dans la section précédente, nous habilitent à établir la faisabilité structurelle de l'allocation. Autrement dit, on détermine les configurations du système hybride qui sont réalisables et fonctionnelles, par opposition aux propriétés non-fonctionnelles, qu'une configuration doit également remplir avant d'être validée. Dans la communauté des systèmes embarqués, on distingue par fonctionnel et non-fonctionnel, service et qualité de service. Si grâce à l'allocation, on forme une configuration du système implémentant les services, il reste après à vérifier si son comportement est au niveau de la qualité requise. Parmi, les propriétés non-fonctionnelles, on trouve en premier la performance, la consommation d'énergie et l'utilisation mémoire. D'autres propriétés sont exigées selon le domaine d'application, telle la tolérance aux fautes pour les systèmes critiques. Dans notre cas :

- La performance qu'on dit aussi bon-fonctionnement, revient à vérifier que les délais (deadlines) annotés par le dispositif `RTF` au niveau des services temps-réel de notre application ACCORD, ne sont jamais dépassés lors de l'exécution.
- L'utilisation mémoire consiste à vérifier que l'usage mémoire de l'application temps-réel ACCORD ne dépasse, à aucun moment de l'exécution, la taille disponible et annotée avec HRM au niveau de la plateforme matérielle. Signalons que le caractère statique des applications ACCORD, nous garantit une occupation mémoire stable le long de l'exécution.
- La consommation d'énergie revient à vérifier l'autonomie du système intégré. Si on a décrit par le profil physique de HRM la description énergétique de la plateforme matérielle dans l'absolu, il faut mesurer maintenant son autonomie une fois sollicitée par l'application ACCORD.

3.2 Déroulement du processus

Afin de valider un système hybride fraîchement intégré, il faut enchaîner les phases d'analyses, de simulations et de tests. Cet ordre est très important, il prend en compte l'augmentation en coût, en temps et en précision pendant le processus de validation. Plus finement, chaque phase se compose de plusieurs étapes et chaque étape correspond à valider certaines contraintes non-fonctionnelles. Il faut donc prendre en compte les mêmes critères d'ordre pour organiser les étapes à l'intérieur de chaque phase, par exemple, il faudra organiser les différentes analyses des différentes propriétés non-fonctionnelles selon leur coût et le temps qu'elles nécessitent.

Si une étape de validation ne peut garantir qu'une configuration validée soit vraiment opérationnelle c-à-d qu'elle soit également validée par les étapes suivantes[5]. Il faut néanmoins qu'elle assure qu'une configuration rejetée est réellement inadaptée, en donnant comme justification un scénario d'exécution non-valide. Cela est une condition nécessaire à toute heuristique de validation. On dit qu'une étape de validation est efficace ou précise quand elle détecte un maximum de configurations non-valides et qu'elle les retire du processus de validation engendrant ainsi d'appréciables économies. Techniquement, on mesure l'efficacité d'une étape de validation, par le ratio du nombre de configurations

[5] sachant que les étapes de validation suivantes seront plus précises

validées par les étapes suivantes parmi celles validées par l'étape courante. Par conséquent, moins il y a de configurations validées par l'étape courante et rejetées par les étapes suivantes, plus l'étape est efficace.

Ouvrant le bal, on exige paradoxalement de l'analyse d'être rapide et très efficace. Par sa rapidité elle économisera du temps au flot de validation et par son efficacité elle économisera d'inutiles futures étapes de validation. Une liste exhaustive des analyses qu'on peut généralement opérer sur un système embarqué, est donnée en page 71. La simulation est la seconde phase de validation, elle permet de tester le système avant la disponibilité physique du matériel, elle se compose de deux étapes correspondant à deux niveaux de précision (voir section 2.3.2 du chapitre 2) : la simulation dite fonctionnelle est incapable de valider la performance mais peut valider d'autres propriétés non-fonctionnelles tel l'usage mémoire, alors que la simulation au cycle près est plus lente mais beaucoup plus précise, elle permet en conséquence de valider la performance du système. Finalement, la phase test est la dernière validation opérée, elle doit être exhaustive et nécessite du temps, il est donc très préjudiciable de ne détecter la non-validité d'une configuration que pendant la phase de test.

3.3 Scénarios

Deux scénarios se présentent à nous, un catastrophe et un optimum. Le scénario catastrophe correspond à un rejet de toutes les configurations disponibles par le processus de validation, cela engendre une itération coûteuse qui remet en cause tout le développement des deux composantes logiciel/matériel. Le scénario optimum ou idéal correspond quant à lui, a une validation d'une configuration par toutes les phases du processus, c'est par exemple le cas de *config_m* dans la figure 4.17. On peut éventuellement s'arrêter à cette première configuration opérationnelle, comme on peut continuer pour en trouver d'autres. Il faudra ensuite choisir la configuration la plus optimale par rapport à une propriété non-fonctionnelle donnée, on peut choisir, par exemple, la configuration la plus performante ou bien la plus autonome.

En conclusion, on a couvert à ce stade toutes les étapes de développement d'un système embarqué : la conception du matériel (chapitre 3), le développement du logiciel (section 1 de ce chapitre), l'allocation (section 2 de ce chapitre) et enfin la validation dans cette section. Nous proposerons dans la prochaine et dernière section de ce travail, un cas d'étude de synthèse pour mettre en valeur l'utilité de nos contributions dans le cadre d'un flot complet de développement de systèmes embarqués complexes, et cela depuis la spécification jusqu'à l'implémentation valide finale. Nous allons surtout montrer par ce cas d'étude, comment une forte communication des intentions de conception entre les deux flots logiciel et matériel est la clé d'une intégration et validation réussies du système.

4 Cas d'étude : co-développement logiciel/matériel d'une chenille de robots

Pour illustrer et appliquer les contributions de ce travail de thèse qui interviennent à différents niveaux du flot de conception, il nous faut développer entièrement un cas d'étude qui doit réunir plusieurs critères :

≻ ***Temps-réel*** : Le système doit être contraint à des délais d'exécution.

≻ ***Embarqué*** : Le système doit être hétérogène avec deux composantes une logicielle et une matérielle.

≻ ***Multi-tâches*** : Le système doit réaliser plusieurs opérations en parallèle.

≻ ***Distribué*** : L'architecture matérielle/logicielle est distribuée sur plusieurs noeuds.

≻ ***Répétitif*** : Le système doit avoir ou contenir une structure répétitive.

≻ ***Paramétrable*** : On signifie par ce critère de qualité, la capacité à intervenir via des paramètres sur l'architecture du système ou la complexité des tâches, dans le but d'expérimenter, plusieurs configurations avec un même cas d'étude.

Nous avons pensé à une chenille de robots unicycles qui roulent sans glisser sur un plan horizontal. Le scénario de fonctionnement est le suivant :

1. Initialement, n robots unicycles identiques sont à l'arrêt à n différentes positions. Comme on a vu en section 4 du chapitre 2, un robot unicycle est constitué de deux roues fixes, de même axe, motrices, et indépendantes. La position d'un robot est déterminée par le triplet (x, y, θ) où (x, y) sont les coordonnées du milieu de l'essieu du robot et θ est l'angle d'orientation mesuré entre l'axe longitudinal du robot et l'axe des abcisses $\vec{x}$.
2. A $t = 0$, l'utilisateur choisit un robot Rob_1 parmi les n robots, auquel il communique les positions des $(n-1)$ autres robots. On suppose que chaque robot connait sa position.
3. Rob_1 doit trouver parmi les autres robots celui qui par sa position est le plus proche. On peut imaginer d'autres heuristiques de choix, vu qu'on est clairement face au problème classique du voyageur de commerce.
4. Une fois le prochain robot Rob_2 déterminé, Rob_1 doit calculer une trajectoire qui doit le mener jusqu'à s'arrêter derrière Rob_2 en s'alignant à lui (même angles d'orientation $\theta_1 = \theta_2$). Rob_1 doit opérer les deux calculs précédents dans un délai T_{start}.
5. A $t = T_{start}$, le robot Rob_1 entame sa trajectoire. Il doit atteindre sa position finale en s'arrêtant derrière Rob_2 à $t = T_{start} + T_{traj}$. Le temps du trajet est donc T_{traj}.
6. A $t = T_{start} + T_{traj}$, l'opération mécanique de sceller les deux robots commence, il s'agit d'accrocher Rob_1 au milieu de l'essieu de Rob_2. Cette liaison est pivot et forme un angle $(\theta_2 - \theta_1) \in]-\frac{\pi}{2}, \frac{\pi}{2}[$. La jonction nécessite un temps $T_{link} < T_{start}$. Dès qu'elle est établie, un signal SIG_{link} est envoyé à Rob_1.
7. En parallèle avec l'opération mécanique précédente, Rob_1 doit choisir le prochain robot Rob_3 à joindre et calculer une trajectoire pour mener la chenille formée de Rob_1 et Rob_2 jusqu'à s'arrêter derrière Rob_3 en ligne droite. Le calcul doit prendre moins de T_{start}.
8. A $t = 2T_{start} + T_{traj}$, la chenille ($Rob_1$,$Rob_2$) entame son mouvement vers Rob_3 qu'elle atteint en $t = 2(T_{start} + T_{traj})$, soit un temps de trajet toujours de T_{traj}.

9. L'opération de jonction entre Rob_2 et Rob_3 commence puis dure $T_{link} < T_{start}$. A sa fin, le signal SIG_{link} est envoyé à Rob_1.
10. ...On itère ainsi jusqu'à ramasser les n robots à $t = (n-1)(T_{start} + T_{traj}) + T_{link}$.

La figure 4.18 illustre le fonctionnement d'une chenille de robots[6]. On y voit la chenille aux moments des jonctions qui s'allonge au fur et à mesure. On peut aussi distinguer la position initiale de chaque robot et sa trajectoire lors des mouvements de la chenille.

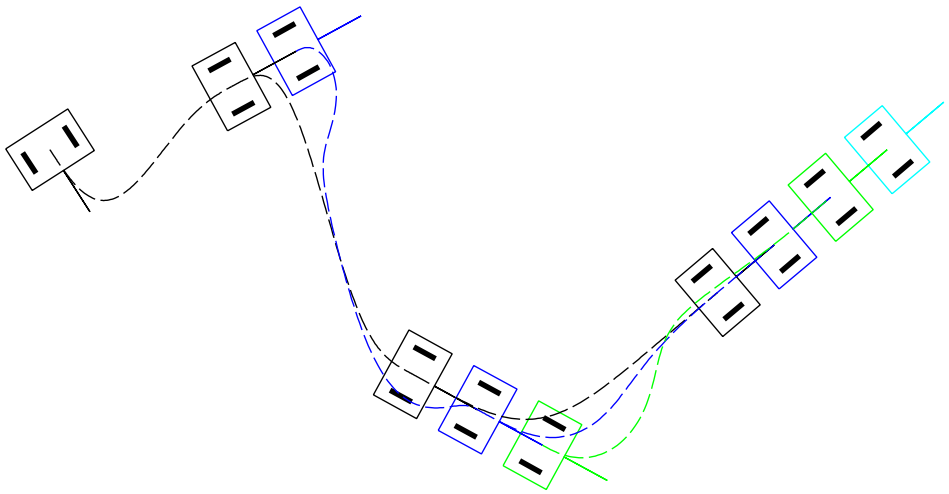


FIG. 4.18 – Fonctionnement de la chenille de robots

Globalement, on distingue deux états de fonctionnement de la chenille correspondant à deux tâches :

- La chenille est en mouvement quand $t \in [i.T_{start} + (i-1).T_{traj}, i.(T_{start} + T_{traj})]$ avec $i \in [1, n-1]$. Pendant une durée T_{traj}, elle se compose de i robots et doit commander et asservir les $2i$ roues motrices indépendemment.
- La chenille est à l'arrêt quand $t \in [(i-1).(T_{start}+T_{traj}), i.T_{start}+(i-1).T_{traj}]$. Pendant une durée T_{start}, elle doit calculer la prochaine trajectoire vers le $(i+1)$-éme robot à joindre.

Il est clair qu'au fur et à mesure que la chenille s'allonge, les tâches de calcul deviennent plus longues et plus complexes, mais les contraintes de temps, en l'occurrence T_{start} et T_{traj}, restent inchangées. Par ce cas d'étude, on testera une montée en charge incrémentale.

Pour le flot de conception de la chenille, nous adopterons le processus en losange qu'on a introduit en section 2.2 chapitre 2 (voir figure 2.11). On commencera par la spécification fonctionnelle, puis on développera parallèlement le matériel et le logiciel, et finalement on établira l'allocation et la validation par simulation. Le long de ces étapes nous mettrons en valeur les contributions suivantes :

- Modélisation du matériel avec le langage HRM et application de la méthodologie de modélisation (sections 1 et 2 du chapitre 3).
- Application simultanée des profils HRM et RSM (Repetitive Structure Modeling, section 3 du chapitre 2) pour étendre la modélisation du matériel aux structures répétitives.
- Simulation du matériel avec l'outillage fourni en section 3 du chapitre 3.

[6]Cette figure est une prise d'écran d'une implémentation de la chenille de robots sous Scilab [75].

– Prise en charge du matériel dans la conception de l'application temps-réel ACCORD et modélisation d'une abstraction de cette application (section 1 du chapitre 4).
– Allocation et adéquation entre logiciel et matériel (section 2 du chapitre 4).

4.1 Spécification fonctionnelle

Les fonctionnalités de la chenille se résument principalement à deux tâches : la planification de trajectoire et le suivi de trajectoire. Mais commençons d'abord par la modélisation cinématique de la chenille et l'étude de sa commandabilité. Pour la modélisation et la commande de la chenille de robots, nous nous sommes inspiré de la voiture à n chariots [68, 66]. Il s'agit d'un système plat, parfaitement commandable sous la contrainte de roulement sans glissement.

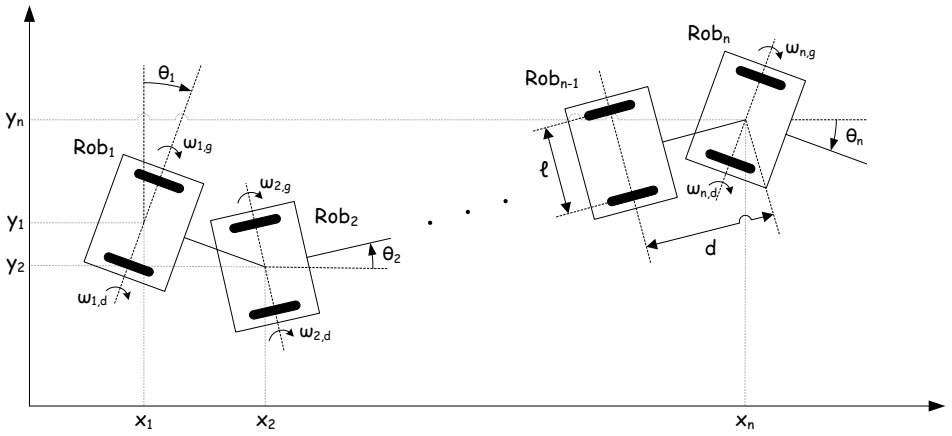


FIG. 4.19 – Modélisation de la chenille de robots

Considérons n le nombre courant de robots qui forment la chenille. Comme le montre la figure 4.19, l'état de la chenille est entièrement situé grâce à $3n$ variables sous la forme d'un vecteur $(x_i, y_i, \theta_i)_{1\leq i\leq n}$. Si on a besoin, à première vue, d'autant de variables, c'est qu'on ne prend pas en compte les liaisons pivots entre robots. En considérant cela, on obtient les $(2n-2)$ équations suivantes pour $2 \leq i \leq n$:

$$x_i = x_1 + d.\sum_{j=1}^{i-1}\cos(\theta_j)$$
$$y_i = y_1 + d.\sum_{j=1}^{i-1}\sin(\theta_j)$$

On déduit, qu'il nous suffit de connaître les $(n+2)$ variables (x_1, y_1) et $(\theta_i)_{1\leq i\leq n}$ pour retrouver les $(2n-2)$ restantes $(x_i, y_i)_{2\leq i\leq n}$. Considérons aussi que tous les robots roulent sans glissement, cela se traduit par n équations :

$$tan(\theta_i) = \frac{\dot{y_i}}{\dot{x_i}}, 1 \leq i \leq n$$

Un simple raisonnement par récurrence [68], permet de prouver que le système est entièrement spécifié par (x_1, y_1) et leurs dérivées jusqu'à l'ordre n. La chenille de robots est donc un système plat (voir section 4 chapitre 2) et les coordonnées (x_1, y_1) de Rob_1 sont une sortie plate. Il nous suffit alors de commander la chenille en fonction des composantes de cette sortie plate puis de reconstituer automatiquement les commandes de chaque roue de la chenille.

Planification de trajectoire

Il s'agit de planifier une trajectoire que la chenille doit suivre pour aller d'une position initiale $(x_{init}, y_{init}, \theta_{init})$ à une position finale $(x_{final}, y_{final}, \theta_{final})$ en un temps T_{traj}. Les trajectoires de la chenille sont toujours d'arrêt à arrêt. En plus, la chenille est toujours alignée à l'arrêt c-à-d que tous les robots ont le même angle d'orientation $(\theta_i = \theta)_{1 \leq i \leq n}$. Par conséquent, les positions initiale et finale de Rob_1 suffisent pour déterminer l'état initial et final de toute la chenille. Mathématiquement, on cherche une courbe $y = f(x)$ avec les conditions suivantes :

$$x(0) = x_{init}; \quad \dot{x}(0) = 0; \quad x(T_{traj}) = x_{final}; \quad \dot{x}(T_{traj}) = 0;$$
$$f(x_{init}) = y_{init}; \quad \frac{df}{dx}(x_{init}) = \tan(\theta_{init}); \quad f(x_{final}) = y_{final}; \quad \frac{df}{dx}(x_{final}) = \tan(\theta_{final});$$
$$\frac{d^i f}{dx^i}(x_{init}) = \frac{d^i f}{dx^i}(x_{final}) = 0, (2 \leq i \leq n)$$

Nous utiliserons la théorie de l'interpolation [66], pour trouver $x(t)$ sous forme d'un polynôme de t, puis $y(x)$ sous forme d'un polynôme de x. Sachant que $x(t)$ vérifie 4 conditions (voir ci-dessus), on obtient donc un polynôme de degré 3 :

$$x(t) = x_{init} + (x_{final} - x_{init}).(\frac{t}{T_{traj}})^2.(3 - 2(\frac{t}{T_{traj}}))$$

Le polynôme $y(x)$ doit, quant à lui, vérifier $2n + 2$ conditions, il est donc de degré $2n + 1$:

$$y(x) = \sum_{j=0}^{2n+1} a_j x^j$$

tel que $(a_j)_{0 \leq j \leq 2n+1}$ est solution de :

$$\begin{pmatrix} 1 & x_{init} & x_{init}^2 & x_{init}^3 & \dots & x_{init}^{2n+1} \\ 1 & x_{final} & x_{final}^2 & x_{final}^3 & \dots & x_{final}^{2n+1} \\ 0 & 1 & 2x_{init} & 3x_{init}^2 & \dots & (2n+1)x_{init}^{2n} \\ 0 & 1 & 2x_{final} & 3x_{final}^2 & \dots & (2n+1)x_{final}^{2n} \\ 0 & 0 & 2 & 6x_{init} & \dots & (2n+1)(2n)x_{init}^{2n-1} \\ \vdots & \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & 0 & \dots & \frac{(2n+1)!}{(n+1)!}x_{init}^{n+1} \\ 0 & 0 & 0 & 0 & \dots & \frac{(2n+1)!}{(n+1)!}x_{final}^{n+1} \end{pmatrix} \cdot \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{2n+1} \end{pmatrix} = \begin{pmatrix} y_{init} \\ y_{final} \\ \tan(\theta_{init}) \\ \tan(\theta_{final}) \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \end{pmatrix}$$

Pour planifier une trajectoire, il faut donc inverser une matrice carrée de taille $(2n + 2)^2$.

Commande et suivi de trajectoire

Maintenant qu'on a calculé une trajectoire pour la sortie plate (x_1, y_1) de Rob_1, il faut générer la commande des $2n$ roues motrices de la chenille $(\omega_{i,g}, \omega_{i,d})_{1 \leq i \leq n}$. Normalement, seules deux commandes (locomotive) suffisent à asservir toute la chenille, d'où la dimension de la sortie plate. Il nous faut donc commander toutes les roues en harmonie pour avoir le même mouvement et garantir un roulement sans glissement, exactement comme si on avait une locomotive.

On considère que $y = y(x)$ et on se place dans le repère de Frenet [68], on aboutit aux équations récursives suivantes :

$$\tan(\theta_{i+1}-\theta_i) = d.k_i, (1 \leq i \leq n-1);$$
$$ds_1 = \sqrt{1+(dy/dx)^2}\, dx; \quad k_1 = \frac{d^2y/dx^2}{(1+(dy/dx)^2)^{3/2}};$$
$$ds_{i+1} = \sqrt{1+d^2{k_i}^2}\, ds_i, (1 \leq i \leq n-1);$$
$$k_{i+1} = \frac{1}{\sqrt{1+d^2{k_i}^2}}(k_i + \frac{d}{1+d^2{k_i}^2}\frac{dk_i}{ds_i}), (1 \leq i \leq n-1).$$

s_i et k_i sont respectivement l'abscisse curviligne et la courbure du robot Rob_i de la chenille le long de la trajectoire[7]. Dans un premier temps, on considère que la commande est sous la forme $(u_i, \dot{\theta}_i)_{1\leq i\leq n}$ où u_i est la vitesse curviligne du robot Rob_i et $\dot{\theta}_i$ est sa vitesse de rotation. On obtient les commandes récursives suivantes :

$$u_1 = \sqrt{1+(dy/dx)^2}\, \dot{x}(t); \quad \dot{\theta}_1 = \frac{d^2y/dx^2}{(1+(dy/dx)^2)}\, \dot{x}(t)$$
$$u_{i+1} = \sqrt{1+d^2{k_i}^2}\, u_i, (1 \leq i \leq n-1);$$
$$\dot{\theta}_{i+1} = \dot{\theta}_i + \frac{d}{1+d^2{k_i}^2}\frac{dk_i}{ds_i}u_i, (1 \leq i \leq n-1).$$

Pour retrouver finalement les commandes $(\omega_{i,g}, \omega_{i,d})_{1\leq i\leq n}$, on utilise les équations suivantes (voir section 4 du chapitre 2) :

$$\omega_{i,g} = \frac{2u_i - \ell\dot{\theta}_i}{2r}, (1 \leq i \leq n); \quad \omega_{i,d} = \frac{2u_i + \ell\dot{\theta}_i}{2r}, (1 \leq i \leq n).$$

r est le rayon commun des roues.

Une fois la commande $(\omega_{i,g}, \omega_{i,d})$ concernant Rob_i est calculée, ce dernier devra asservir en vitesse ses deux moteurs de roues. Pour simplifier notre cas d'étude, on supposera d'abord que les robots ne sont pas limités en vitesse, ensuite, on supposera qu'on dispose de régulateurs spécialisés (HwASIC) implémentant le PID (régulateur Proportionnel Intégral Dérivé) adéquat, cela nous évitera d'implémenter $2n$ régulateurs de vitesse de manière logicielle via des objets temps-réel qu'on intégrera dans notre application. Enfin, on considère qu'un moteur de roue est suffisamment précis, et que le temps d'établissement de la vitesse commandée est infiniment petit par rapport à la cadence de la chenille.

Bilan

Notre planification et suivi de trajectoire sont dites en *boucle ouverte*, puisque nos calculs ne prennent pas en compte et ne corrigent pas d'éventuels écarts par rapport à la trajectoire de référence. On doit alors connaître parfaitement la dynamique de notre chenille qui ne doit subir, par ailleurs, aucune perturbation de son environnement qui soit capable de la dévier de la trajectoire calculée. Remarquons qu'en plus d'être temps-réel embarqué multi-tâches distribué et répétitif, notre cas d'étude est bien paramétrable, notamment par le biais de :

- n : nombre de robots à joindre.
- T_{start} : temps d'une planification de trajectoire.
- T_{traj} : temps d'un mouvement de chenille.
- nb_{traj} : nombre de périodes d'échantillonage d'une trajectoire, une commande est donc calculée pour chaque période (T_{traj}/nb_{traj}). Globalement, plus nb_{traj} est élevé, plus le mouvement de la chenille est fluide et précis.

[7] Il ne faut pas confondre ici le d de dérivation et la distance d qui sépare deux robots de la chenille.

On peut intervenir librement et de manière indépendante sur ces paramètres pour configurer l'architecture de l'application et ses besoins en calcul, en performance, en taille mémoire et en charge de communication.

4.2 Développement et modélisation du matériel

Au lancement, un seul robot est désigné pour entamer la formation de la chenille. Ce premier robot Rob_1 doit être capable à lui seul de décider du second robot Rob_2 à joindre, de planifier sa trajectoire vers Rob_2, puis de s'auto-commander le long de cette trajectoire. Par conséquent, Rob_1 doit contenir une plateforme d'exécution entière et une source d'énergie pour accomplir toutes ces tâches de manière autonome. En plus, comme le choix de Rob_1 au départ est aléatoire, tous les robots doivent contenir une telle plateforme d'exécution. L'architecture matérielle de la chenille de robots est alors clairement **distribuée**.

Les robots unicycles formant la chenille sont identiques à tous les niveaux. Ils ont les mêmes dimensions, le même type de roues et de moteurs, et surtout ils embarquent le même matériel informatique. L'architecture matérielle de la chenille de robots a donc une **structure répétitive**.

La spécification fonctionnelle a révélé de fortes dépendances entre les calculs des commandes des différents robots. Matériellement, cela se traduit par des connexions entre les plateformes de robots. La structure physique de la chenille ne permet de lier que deux robots qui se suivent. La distribution matérielle de la chenille est ainsi sous forme de **chaîne**.

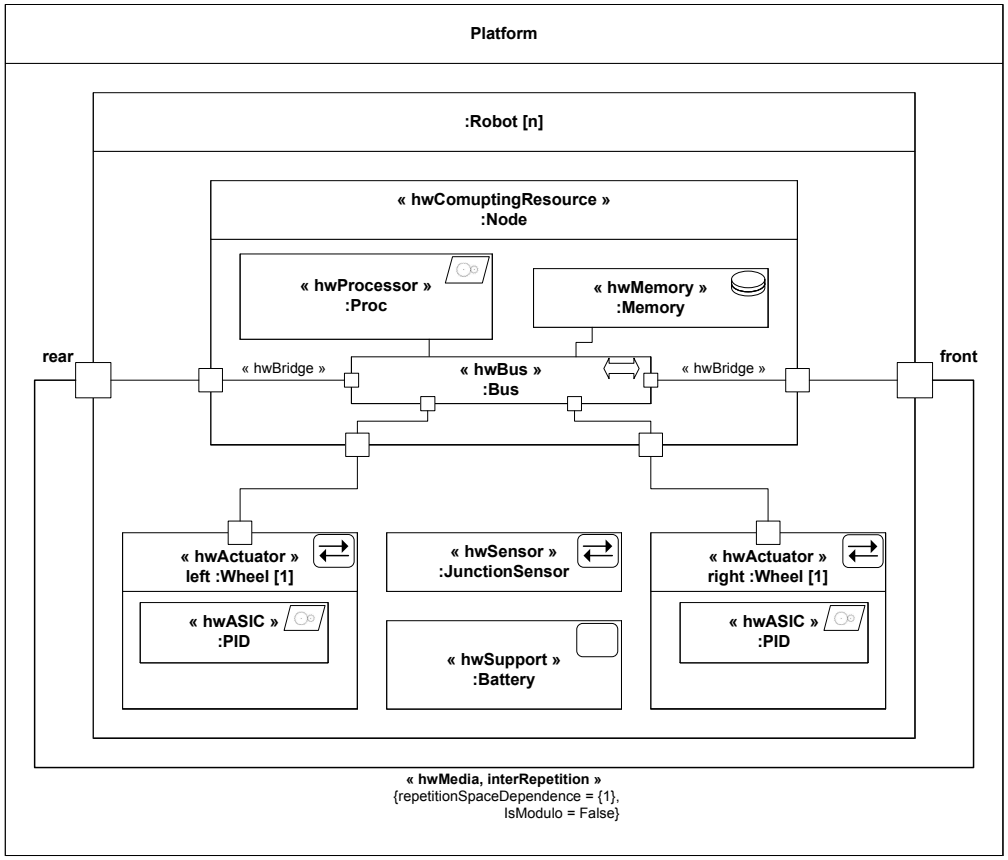


FIG. 4.20 – L'architecture matérielle de la chenille

Dans la figure 4.20, on a modélisé méthodologiquement la plateforme matérielle de la chenille. Il s'agit d'une répétition en chaîne de n robots, où chaque robot est connecté via un `HwMedia` au robot qui le devance et à celui qui le suit. On exprime ces connexions grâce au dispositif `interRepetition` du profil RSM de MARTE. On a là une double application de stéréotypes provenant de deux profils différents en l'occurence HRM et RSM. Un robot se compose d'un noeud de calcul avec processeur, bus et mémoire, deux actionneurs contenant chacun un `HwASIC` implémentant le PID d'asservissement du moteur de roue, un détecteur de sa jonction à la chenille et une batterie pour sa propre autonomie.

Ici, on ne représente qu'une vue logique du matériel, il serait intéressant d'en modéliser les deux vues physiques, l'une avec `HwPower` pour la modélisation des aspects énergitiques et l'autre avec `HwLayout` pour la modélisation des formes, des dimensions et de la diposition des composants à l'intérieur des robot.

4.3 Conception de l'application temps-réel

L'aspect récursif des équations qu'on a élaboré lors de la spécification fonctionnelle se traduit par un calcul séquentiel des commandes des différents robots. En effet, il est obligatoire de calculer les commandes, robot par robot dans l'ordre de notre indexation. Autrement dit, le calcul des commandes de Rob_i nécessite le calcul en amont des commandes des robots précédents $(Rob_j)_{j<i}$. D'un autre côté, l'architecture matérielle développée ci-dessus est distribuée, et seul un calcul parallèle pourra en tirer partie. Face à ce dilemme, la solution est simple et classique, il s'agit d'utiliser une architecture logicielle en flot de données sous forme d'un *pipeline*.

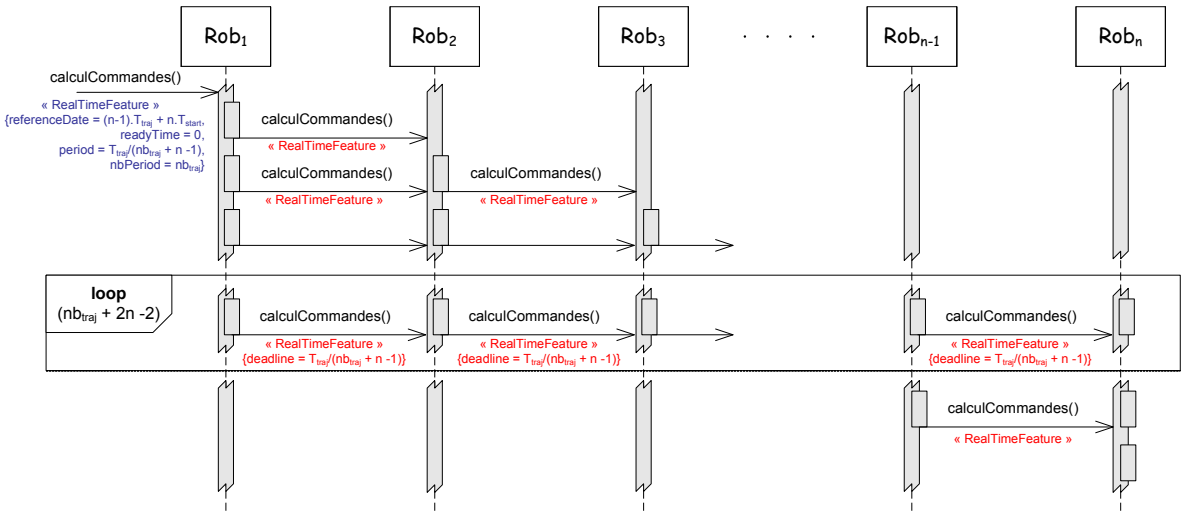


FIG. 4.21 – Pipeline de commande et de suivi de trajectoire

La figure 4.21 est un diagramme de séquence UML qui décrit le fonctionnement en *pipeline* des calculs de commandes lors du suivi de trajectoire. A $t = n.T_{start} + (n-1).T_{traj}$, la trajectoire est planifiée pour la n-ième fois, la chenille entame son mouvement et elle mesure n robots $(Rob_i)_{1 \leq i \leq n}$. Tout commence par Rob_1 qui exécute la méthode *calculCommandes()* avec une `RealTimeFeature` (contrainte de temps) périodique qui relance cette méthode nb_{traj} fois à une période de $T_{traj}/(nb_{traj} + n - 1)$. On rappelle que nb_{traj} est le nombre

de commandes calculées à interval égal lors d'une même trajectoire, il exprime la fréquence d'échantillonnage. On augmente ici nb_{traj} de $(n-1)$ pour couvrir le début du *pipeline*. Notons que cette augmentation implique des délais plus courts et donc des besoins en performance plus importants. La méthode *calculCommandes()* implémentée par tous les robots, calcule comme son nom l'indique les commandes des roues dans un délai qui ne doit jamais dépasser $T_{traj}/(nb_{traj}+n-1)$. A la fin de son exécution cette méthode envoie un message au robot suivant où elle transmet les commandes calculées localement $(u_i, \dot{\theta}_i, k_i, s_i)$ et appelle la méthode *calculCommandes()* de Rob_{i+1} avec une `RealTimeFeature` portant un délai de $T_{traj}/(nb_{traj}+n-1)$.

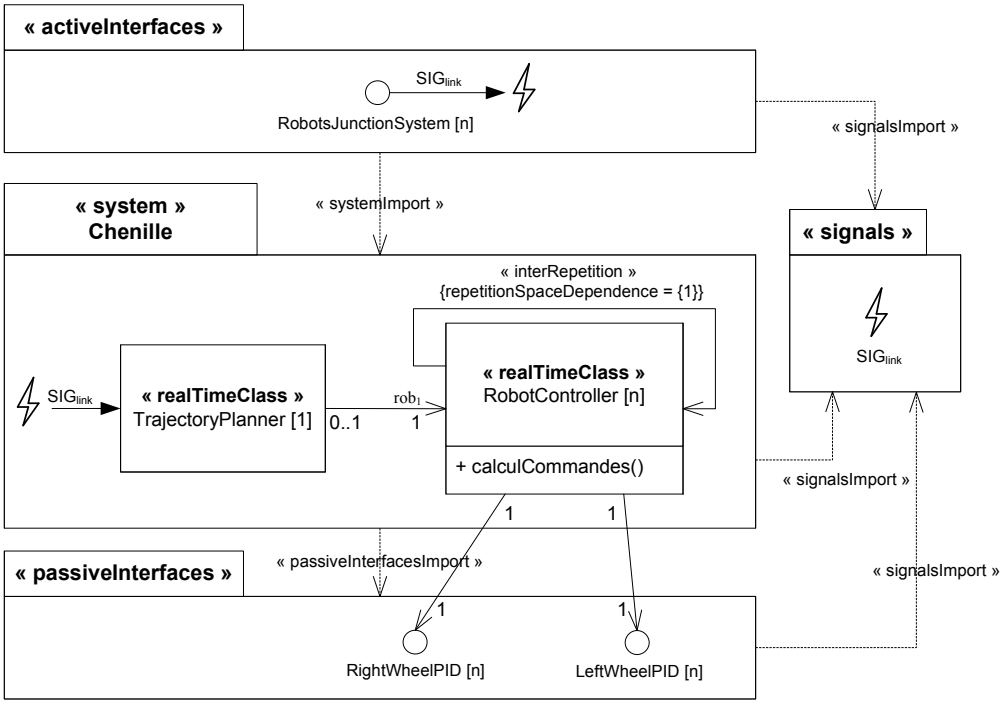


FIG. 4.22 – Architecture logicielle de la chenille

Nous modélisons dans la figure 4.22 une abstraction de la structure de notre application temps-réel ACCORD. L'application a comme seule interface active, le système de jonction qui est embarqué sur chaque robot et qui envoie le signal SIG_{link} quand celui-ci est attaché à la chenille. Le système se compose d'une classe temps-réel *TrajectoryPlanner* sensible au signal SIG_{link}. Cette classe référence le premier contrôleur de robot dans la chenille. Il y a n instances de la classe temps-réel *RobotController* et chacune contrôle deux interfaces passives : une roue droite et une roue gauche.

4.4 Allocation et adéquation logiciel/matériel

Quand la spécification fonctionnelle est claire, et quand l'architecture matérielle est prise en charge lors du développement logiciel, l'allocation est souvent intuitive. En effet, une forte communication entre les flots de conception logiciel et matériel, mène à une structure commune qui facilite le placement un à un des entités logicielles sur les ressources matérielles. Dans le cas précis de la chenille, la prise en charge de l'architecture matérielle qui est distribuée répétitive et sous forme de chaîne, nous a mené à une structure logicielle qui

présente les mêmes caractéristiques c-à-d distribuée en plusieurs objets, répétitive puisque ces objets instancient la même classe temps-réel, et finalement sous forme de chaîne en considérant la disposition des liens entre ces objets.

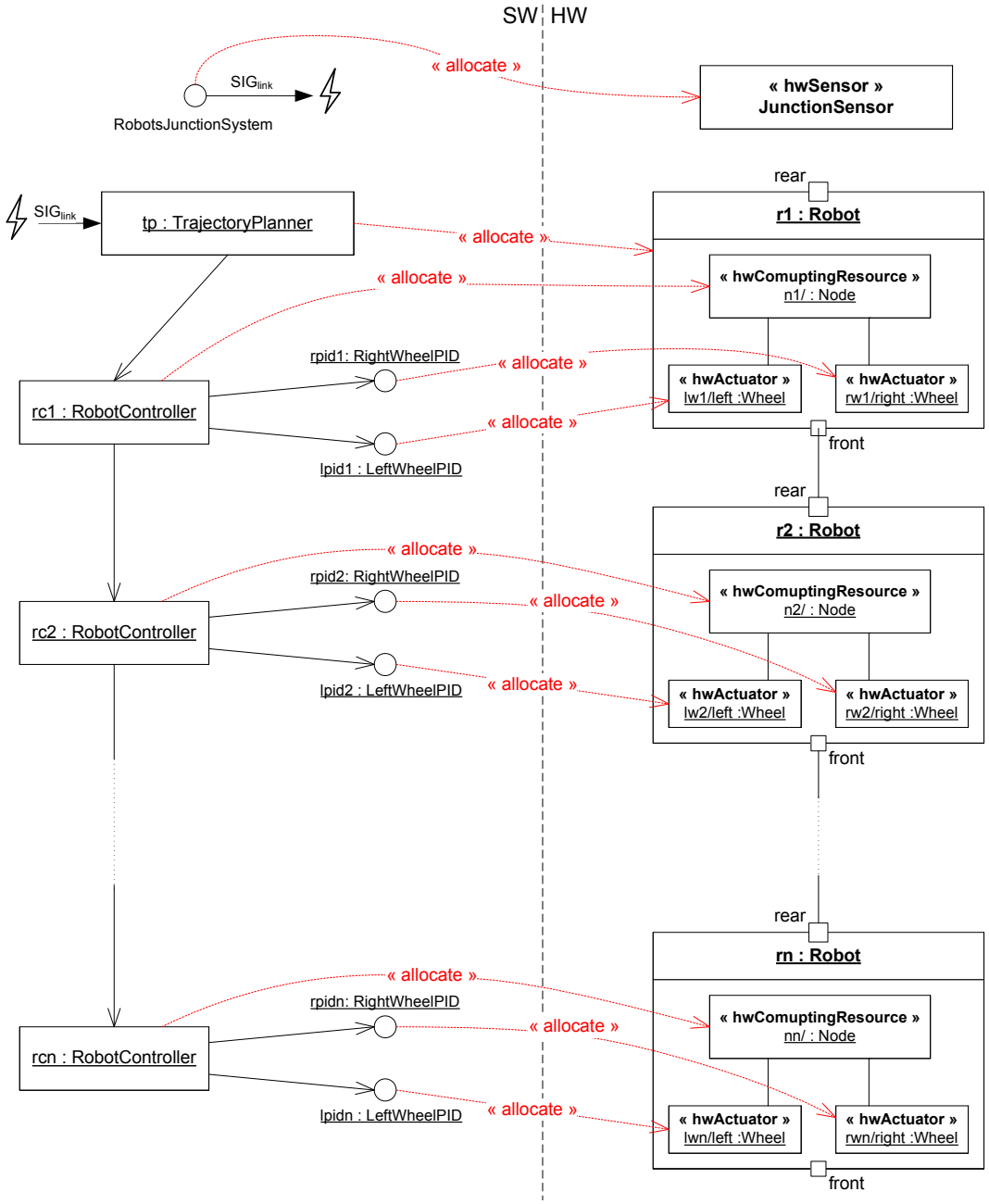


FIG. 4.23 – Allocation logiciel/matériel de la chenille

La figure 4.23 montre un schéma d'allocation qui aboutit à la configuration la plus naturelle du système. En effet, on a alloué le planificateur de trajectoire sur la plateforme du premier robot. Ensuite, on a alloué respectivement chaque contrôleur et les deux roues qu'il commande sur le noeud et les actionneurs du robot correspondant dans la plateforme matérielle de la chenille. En bref, on a alloué chaque objet dans la chaîne logicielle à la ressource qui occupe la même position dans la chaîne matérielle de la chenille. Aussi, on a alloué les n interfaces actives de l'application aux n capteurs de jonction embarqués sur les

n robots de la plateforme.

Dans cette configuration, toutes les contraintes d'allocation qu'on a définies en section 2 sont vérifiées sauf une de type C4. Il s'agit de celle qui concerne l'envoi du signal SIG_{link} d'une interface active *RobotsJunctionSystem* vers la classe temps-réel *TrajectoryPlanner*. En effet, le capteur matériel *JunctionSensor* du n-ième robot est susceptible d'envoyer le signal SIG_{link} à $r1$, et il n'y a aucun lien de communication physique direct entre ces deux ressources matérielles. On applique donc, à $(n-1)$ reprises, notre mécanisme d'adéquation : Classe relai. On illustre cette adéquation en figure 4.24.

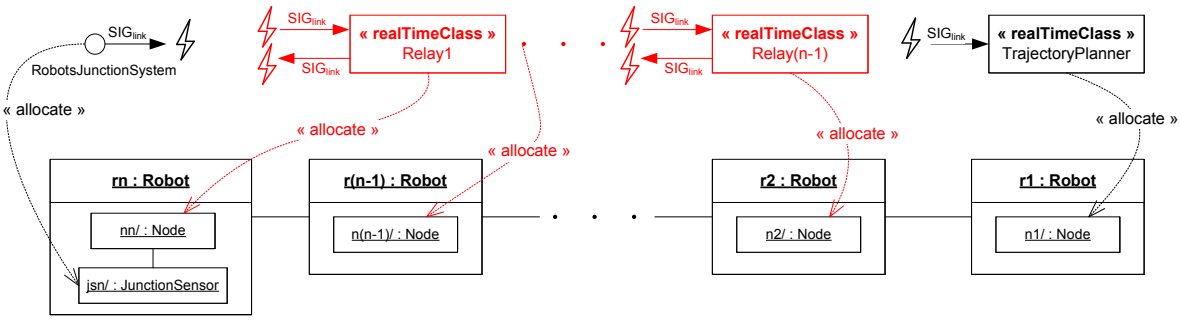


FIG. 4.24 – Adéquation par classes relai

On a ajouté à l'application ACCORD, $(n-1)$ classes relai, chacune d'elles est une classe temps-réel qui a la simple tâche de réémettre le signal SIG_{link} dès qu'elle le reçoit, dans un délai inférieur à $\frac{T_{start}-T_{link}}{n}$. On exprime évidemment cette contrainte de temps grâce au dispositif `RealTimeFeature`.

4.5 Validation par simulation

A ce stade, inutile de rappeler les avantages que procure la simulation et ses différents niveaux de précision, dans l'exploration architecturale et la validation du système. On va simuler la configuration logiciel/matériel obtenue ci-dessus grâce à notre outillage de simulation fourni en section 3 du chapitre 3. Plus précisément, on a choisi dans notre librairie (UML/HRM) de composants *Simics*, la carte *Ebony* qui représente une *(IBM/AMCC PPC440GP Board)*. La figure 3.45 en page 101 donne le modèle détaillé de cette carte. On a utilisé n cartes *Ebony* pour simuler les n robots de la chenille. Effectivement, l'architecture matérielle de la carte *Ebony* est très similaire à l'architecture d'un robot dans la plateforme matérielle de la chenille (voir figure 4.20). La carte *Ebony* comprend un processeur *PowerPC PPC440GP*, un bus système et différentes mémoires. On retrouve donc la partie *Node* de la plateforme matérielle du robot. Ensuite, la carte Ebony a deux interfaces *Ethernet* qu'on utilisera pour simuler les deux ports *front* et *rear* du robot. Rappelons ici que Simics permet grâce à son langage de commandes, de brancher à chaud une liaison *Ethernet*, on a ainsi simulé les jonctions entre robots lors du fonctionnement de la chenille. Aussi la carte *Ebony* a deux ports série qu'on peut utiliser pour s'interfacer aux encodeurs des moteurs de roue. Enfin, cette carte a une horloge temps-réel et des emplacements *PCI* pour étendre la plateforme si on imagine de nouvelles fonctionnalités à la chenille.

Sur chaque carte *Ebony*, on a utilisé le *Linux 2.4* fourni par *Simics*, sur lequel on a porté une instance de l'infrastructure ACCORD. Cette plateforme de simulation ne nous a nécessité

aucune modification ou adaptation du code C++ de l'application temps-réel ACCORD ou de celui de l'infrastructure ACCORD.

4.6 Bilan

On considère la spécification de ce cas d'étude comme une contribution en soi. En effet, la chenille de robots est un système embarqué temps-réel complexe, dont les composantes logicielle et matérielle sont structurellement configurables. Cette flexibilité originale permet de tester par le même cas d'étude plusieurs configurations en agissant simplement sur des paramètres de la chenille comme le nombre de robots ou la fréquence d'échantillonnage. On peut notamment tester la montée en charge incrémental des besoins en performance, en mémoire et en débits de communication lors de l'élongation progressive de la chenille. Nous pensons n'avoir exploité qu'un petit pourcentage des possibilités qu'offre ce cas d'étude. Néanmoins nous avons réussi à expliciter les différentes contributions de cette thèse qui vont de la modélisation du matériel en HRM et l'abstraction du logiciel ACCORD, jusqu'à l'allocation, l'adéquation et la simulation.

CHAPITRE 5

Conclusion

1 Résumé

Ce travail de thèse porte sur la co-modélisation logiciel/matériel des systèmes embarqués temps-réel, et plus précisément, sur la prise en charge du matériel dans la modélisation du logiciel. Il nous a fallu, de ce fait, marier plusieurs domaines de compétence dont l'ingénierie dirigée par les modèles, la conception d'applications temps-réel (méthodologies, analyses, noyaux temps-réel…), le développement du matériel embarqué (modélisation, simulation, implantation…), l'intégration logiciel/matériel (allocation, adéquation, validation…), etc. Nous avons donc longtemps exploré toutes ces disciplines pour accumuler l'état de l'art nécessaire. Nous nous sommes vite rendu compte du grand nombre de problématiques non résolues et de pratiques de conception dépassées ou insuffisantes vu le rythme d'évolution imposé par l'industrie. Après un positionnement en chapitre 2 vis à vis de l'existant, nous avons introduit au fur et à mesure l'ensemble de nos contributions en accompagnant le processus de développement des systèmes embarqués temps-réel.

Afin d'améliorer le développement du matériel et de faciliter la communication des décisions architecturales au flot logiciel, nous avons adopté l'ingénierie dirigée par les modèles dans la conception, simulation et implantation de la plateforme matérielle :

- Définition d'un langage de modélisation HRM (Hardware Resource Model) pour la description de plateformes matérielles sous différentes vues et à différents niveaux de détail. HRM fait partie du profil MARTE standardisé à l'OMG.
- Conception d'une méthodologie de modélisation du matériel en HRM pour assister tout utilisateur novice dans la construction de modèles de plateformes.
- Développement d'un outillage complet et automatisé basé sur Simics pour la simulation des plateformes matérielles ainsi modélisées.
- Unification entre HRM et le standard d'implantation du matériel IP-XACT dans un même processus de développement.

Du côté logiciel, l'application temps-réel est développée, dans notre cas, par la méthodologie ACCORD/UML. Grâce aux contributions précédentes, UML devient un langage commun entre le matériel et le logiciel. Ainsi, La prise en charge du matériel est facilitée :

- Réalisation d'un métamodèle pour la description structurelle des applications temps-réel modélisées par la méthodologie ACCORD/UML.
- Spécification des règles et des contraintes d'allocation qui régissent les placements des entités logicielles sur les ressources matérielles.
- Développement de mécanismes d'adéquation pour adapter des configurations fraîchement allouées.
- Description du processus de validation des configurations d'allocation avec prise en charge des propriétés non-fonctionnelles.

Enfin, pour illustrer l'agencement de toutes ces contributions dans le cadre d'un même processus de développement, nous avons développé une chenille de robots unicycles qui roulent sans glisser sur un plan horizontal. Il s'agit d'un système qui est à la fois temps-réel, embarqué, multi-tâches, distribué, répétitif et paramétrable.

2 Perspective

Mis à part des améliorations et des extensions qui peuvent concerner plusieurs de nos contributions, une perspective de recherche nous semble intéressante. Il s'agit de l'adéquation pré-allocation par opposition à l'adéquation post-allocation que nous proposons dans ce travail de thèse. En effet, lors du développement de la chenille de robots, nous avons clairement constaté qu'une communication des intentions de conception dès les premières étapes de développement rend l'allocation possible et intuitive. Il serait donc intéressant que nous travaillions sur une méthodologie de développement conjoint logiciel/matériel centrée sur une communication efficace entre les deux flots de conception. Il faudra commencer par étudier les besoins du développement logiciel en termes d'informations sur le matériel, et vice-versa. Par exemple, nous pourrions communiquer au développeur du matériel les besoins logiciels en termes de mémoire, de puissance de calcul, d'entrées/sorties, etc. Dans l'autre sens, nous transmettrons à l'informaticien les détails concernant le nombre d'unités de calculs, les débits de communication, etc. Nous devrions commencer d'abord par de simples systèmes embarqués mono-tâche et mono-CPU et établir des règles d'adéquation selon les besoins en performance et en mémoire, puis se diriger progressivement vers des architectures plus complexes, à l'image de la chenille de robots.

Table des figures

Bibliographie

[1] Ian Philips. When less means more ; and more, the-same In *IEEE Second International Symposium on Industrial Embedded Systems SIES'2007*, 2007.

[2] Bran Selic. Model-Driven Development : Its Essence and Opportunities. *IEEE*, pages 313–319, ISORC 2006.

[3] Extensible Markup Language (XML). *http ://www.w3.org/XML/*.

[4] *L'ingénierie dirigée par les modèles, au-delà du MDA*. Lavoisier, Hermes-Science, 2006.

[5] Object Managment Group (OMG). *http ://www.omg.org/*.

[6] OMG Architecture Board. Model Driven Architecture (MDA Guide Version 1.0.1). Technical Report 2003-06-01, OMG, 2003.

[7] Frédéric Thomas, Safouan Taha, Ansgar Radermacher et Sébastien Gérard. Motif pour la métamodélisation : Plateforme d'exécution. dans 2eme Journée sur l'Ingénierie Dirigée par les Modèles, pages 235–238, 2006.

[8] Joao Paulo A. Almeida, Remco M. Dijkman, Marten van Sinderen et Luis Ferreira Pires. Platform-Independent Modelling in MDA : Supporting Abstract Platforms. In *MDAFA*, volume 3599 of *Lecture Notes in Computer Science*. Springer, 2005.

[9] Object Management Group, Inc. Meta-Object Facility (MOF) 2.0 Specification., 2004-10-15.

[10] Ole-Johan Dahl and Kristen Nygaard. Simula : an algol-based simulation language. *Commun. ACM*, 9(9) :671–678, 1966.

[11] Grady Booch. Object oriented design. *IEEE Transactions on Software Engineering*, 12(12) :211–221, 1986.

[12] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-oriented modeling and design*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[13] I. Jacobson, M.Christerson, P. Jonsson, and G. Overgaard. *Object-Oriented Software Engineering : A Use Case Driven Approach*. Addison-Wesley, 1992.

[14] Object Management Group, Inc. OMG UML Infrastructure, V2.1.2. Technical Report formal/2007-11-04, OMG, 2007.

[15] Object Management Group, Inc. OMG UML Superstructure, V2.1.2. Technical Report formal/2007-11-02, OMG, 2007.

[16] Fuentes-Fernández, Lidia and Vallecillo-Moreno, Antonio . An Introduction to UML Profiles. *The European Journal for the Informatics Professional*, 5(2), April 2004.

[17] A. Abouzahra, Jean Bézivin, Marcos Didonet Del Fabro, and Frédéric Jouault. A Practical Approach to Bridging Domain Specific Languages with UML profiles. In *OOPSLA Workshop on Best Practices for Model Driven Software Development*, 2005.

[18] Object Management Group, Inc., editor. *UML 2 OCL (Final Adopted specification)*. `http://www.omg.org/cgi-bin/doc?ptc/2003-10-14`, October 2003.

[19] James Bruck and Kenn Hussey. Customizing UML : Which Technique is Right for You ? *International Business Machines Corp.*, 2007.

[20] ATLAS team. ATLAS Transformation Language (ATL). *http ://www.eclipse.org/gmt/atl/*, 2006.

[21] West Team. ModTransf. *http ://www2.lifl.fr/west/modtransf/*, 2005.

[22] Object Management Group, Inc. MOF Query / Views / Transformations. Technical Report ptc/2007-07-07, OMG, 2007.

[23] Object Management Group, Inc. MOF 2.0 XMI Mapping Specification, Version 2.1. Technical Report 2005-09-01, OMG.

[24] CEA LIST. Papyrus UML2 Tool. *http ://www.papyrusuml.org*, 2007.

[25] Obeo. Acceleo Generator. *http ://www.acceleo.org*, 2007.

[26] CEA LIST. Laboratoire d'Ingénierie des Systèmes Embarqués. *http ://www-list.cea.fr/*.

[27] Object Management Group, Inc. Diagram Interchange 2.0. Technical Report formal/2006-04-04, OMG.

[28] IBM. Eclipse - Eclipse Modeling Framework. *http ://www.eclipse.org/modeling/emf/*.

[29] MDT. OCL (Object Constraint Language) Eclipse plugin. *http ://www.eclipse.org/modeling/mdt/?project=ocl#ocl*.

[30] M2T. JET (Java Emitter Templates). *http ://www.eclipse.org/modeling/m2t/?project=jet#jet*.

[31] Alberto Ferrari and Alberto Sangiovanni-Vincentelli. System Design : Traditional Concepts and New Paradigms. In *ICCD '99 : Proceedings of the 1999 IEEE International Conference on Computer Design*, page 2, Washington, DC, USA, 1999. IEEE Computer Society.

[32] Object Management Group, Inc. OMG Systems Modeling Language (OMG SysML), v1.0. Technical Report formal/2007-09-01, OMG, 2007.

[33] Sebastion GERARD. *Modélisation UML exécutable pour les systèmes embarqués de l'automobile*. PhD thesis, 2000.

[34] Marisa López-Vallejo and Juan Carlos López. On the hardware-software partitioning problem : System modeling and partitioning techniques. *ACM Trans. Des. Autom. Electron. Syst.*, 8(3) :269–297, 2003.

[35] C. Schulz-Key and T. Kuhn and W. Rosenstiel. A Framework for System-Level Partitioning of Object-Oriented Specifications. In *Proceedings of the tenth workshop on Synthesis and System Integration of Mixed Technologies (SASIMI'2001)*, Nara, Japan, 2001.

[36] R. Damaševičius and V. Štuikys. Application of the object-oriented principles for hardware and embedded system design. *Integr. VLSI J.*, 38(2) :309–339, 2004.

[37] Object Management Group, Inc. UML profile for System on Chip (SoC), Version 1.0.1. Technical Report formal/06-08-01, OMG, 2006.

[38] E. Riccobene and P. Scandurra and A. Rosti and S. Bocchio. A model-driven design environment for embedded systems. In *DAC '06 : Proceedings of the 43rd annual conference on Design automation*, pages 915–918, New York, NY, USA, 2006. ACM.

[39] Open SystemC Initiative. *http ://www.systemc.org*.

[40] Object Management Group, Inc. UML profile for Schedulability, Performance and Time (SPT), Version 1.1. Technical Report formal/05-01-02, OMG, 2005.

[41] Avionics Architecture Description Language Standards Document (AADL). *http ://www.aadl.info*.

[42] AUTomotive Open System ARchitecture (AUTOSAR). *http ://www.autosar.org*.

[43] Graphical Array Specification for Parallel and Distributed Computing (GASPARD). *http ://www2.lifl.fr/west/gaspard/*.

[44] Yann TANGUY et Sebastien GERARD. Profil ACCORD/UML pour la modélisation d'architecture matérielle. Technical Report 2003-06-20, CEA-LIST.

[45] Amol Bakshi and Vaibhav Mathur and Sumit Mohanty and Victor K. Prasanna and Cauligi S. Raghavendra and Mitali Singh and Aditya Agrawal and James Davis and Brandon Eames and Akos Ledeczi and Sandeep Neema and Greg Nordstrom. MILAN : A Model Based Integrated Simulation Framework for Design of Embedded Systems. *ACM SIGPLAN Notices*, 36(8) :82–87, 2001.

[46] SimpleScalar Tool Set. *http ://www.simplescalar.com*.

[47] Yurcik, W., Wolffe, G. S., and M. A. Holliday. A survey of simulators used in computer organization/architecture courses. In *Summer Computer Simulation Conference (SCSC), Society for Computer Simulation*, 2001.

[48] Simics Platform. *http ://www.virtutech.com*.

[49] Jakob Engblom, Guillaume Girard, Bengt Werner. Testing embedded software using simulated hardware. In *Embedded Real-Time Software (ERTS 2006), Toulouse, France*, 2006.

[50] Simics Model Builder. System Modeling with DML. Technical Report January 2007, Virtutech.

[51] David Servat, Agnès Lanusse, Patrick Vanuxeem, Sébastien Gérard, François Terrier. Doing real-time with a simple Linux kernel. In *Proceedings of the Fifth Real-Time Linux Workshop*, Valencia, Spain, Nov, 9-11 2003.

[52] Wind River. VxWorks. *http ://www.windriver.com/vxworks/*.

[53] Sébastien Gérard and Francois Terrier. UML for real-time : which native concepts to use ? *UML for real : design of embedded real-time systems*, pages 17–51, 2003.

[54] D. Bras, P. Vanuxeem, P. Roux. *Projet ACCORD : Manuel Utilisateur*, 1997.

[55] Pierre Roux. Introduction du synchronisme retardé et du synchronisme fort dans les envois de messages à des objets actifs. Technical report, CEA - LIST.

[56] Object Management Group, Inc. UML Profile for MARTE, Beta 1. Technical Report ptc/07-08-04, OMG, 2007.

[57] *Model Driven Engineering for Distributed Embedded Real-Time Systems*, chapter Model Driven Architecture for Intensive Embedded Systems. ISTE, Hermes science and Lavoisier, 2005.

[58] Huascar ESPINOZA. *An Integrated Model-Driven Framework for Specifying and Analyzing Non-Functional Properties of Real-Time Systems*. PhD thesis, 2007.

[59] Pierre Boulet, Philippe Marquet, Eric Piel, Julien Taillard. Repetitive Allocation Modelling with MARTE. In *Proceedings of the FORUM on Specification & Design Languages*, Barcelona, Spain, September 18-20, 2007.

[60] Alain Demeure and Anne Lafage and Emmanuel Boutillon and Didier Rozzonelli and Jean-Claude Dufourd and Jean-Louis Marro. Array-OL : Proposition d'un Formalisme Tableau pour le Traitement de Signal Multi-Dimensionnel. In *Gretsi*, Juan-Les-Pins, France, September 1995.

[61] J.-P Laumond. *La robotique mobile*. Hermes-Science, 2001.

[62] Bernard BAYLE. *Robotique mobile (cours ENSPS)*. 2007-2008.

[63] F. W. Warner. *Foundations of Differentiable Manifolds and Lie Groups*. Graduate Texts in Mathematics. Springer-Verlag, 1983.

[64] Frédéric JEAN. *Géométrie différentielle appliquée (cours ENSTA AOT13)*. 2007-2008.

[65] H. Nijmeijer and A. J. Van der Shaft. *Nonlinear Dynamical Control Systems*. Springer Verlag, New York, 1990.

[66] J.Lévine. *Analyse et Commande des Systèmes Non Linéaires (cours ENSMP)*. mars 2004.

[67] M. Fliess, J. Lévine, P. Martin and P. Rouchon. Flatness and defect of non-linear systems : introductory theory and examples. *Internat. J. Control*, 61(6) :1327–1361, 1995.

[68] Rouchon, P., fliess, M., Lévine, J. and Martin, P. Flatness and motion planning :The car with n trailers. In *Proceedings of the European Control Conference*, pages 1518–1522, 1993.

[69] F. Thomas, S. TAHA, A. Radermacher and S. Gerard. Motifs pour la métamodélisation : Plateforme d'exécution. In *2e Journées sur l'Ingénierie Dirigée par les Modèles, Lille, France*, pages 235–238, 2006.

[70] Bran Selic. Modeling quality of service with UML : how quantity changes quality. pages 189–204, 2003.

[71] TriCore Architecture. *http ://www.infineon.com/tricore*.

[72] Conrad Bock. UML 2 Composition Model. *Journal of Object Technology*, 3(10) :47–74, 2004.

[73] The SPIRIT Consortium. *http ://www.spiritconsortium.org*.

[74] Sebastion REVOL. *Modélisation de système sur puce (SoC) matériel/logiciel en SystemC/TLM et approche pour la transformation de modèles UML vers TLM*. PhD thesis, 2008.

[75] Scilab : La plateforme open source de calcul scientifique . *http ://www.scilab.org*.

Modélisation conjointe logiciel/matériel de systèmes temps réel

Ce travail de thèse porte précisément sur la prise en charge du matériel embarqué dans la modélisation de l'application temps-réel.

Afin d'améliorer le développement du matériel et de faciliter la communication des décisions architecturales au flot logiciel, nous avons adopté l'ingénierie dirigée par les modèles dans la conception, simulation et implantation de la plateforme matérielle. En effet, nous avons défini un langage de modélisation HRM (Hardware Resource Model) pour la description de plateformes matérielles sous différentes vues et à différents niveaux de détail. Nous avons ensuite conçu une méthodologie de modélisation du matériel en HRM pour assister tout utilisateur dans la construction de modèles de plateformes. Nous avons également développé un outillage complet et automatisé pour la simulation des plateformes matérielles ainsi modélisées. Enfin, nous décrivons un processus d'unification entre HRM et le standard d'implantation du matériel IP-XACT.

Pour mieux prendre en charge le modèle de la plateforme matérielle dans la conception du système temps-réel, nous avons spécifiés des règles et des contraintes d'allocation qui régissent les placements des entités logicielles sur les ressources matérielles. Puis nous avons proposé des mécanismes d'adéquation pour adapter des configurations à priori inadéquates.

Pour finir et illustrer l'agencement de toutes ces contributions dans le cadre d'un même processus de développement, nous avons développé une chenille de robots unicycles qui roulent sans glisser sur un plan horizontal. Il s'agit d'un système qui est à la fois temps-réel, embarqué, multi-tâches, distribué, répétitif et paramétrable.

Hardware/software co-modeling of real-Time systems

This PhD work focuses on the hardware support when modeling real-time systems.

To improve the development of hardware and to communicate architectural intends to the software flow, we adopted the model driven engineering for design, simulation and implementation of hardware platforms. We have first defined a modeling language HRM (Hardware Resource Model) that describes hardware platforms with different views and at different levels of detail. Then, we developed a methodology based on HRM to help users in the construction of their platforms models. We have also developed automated tools for the simulation of these hardware models. Finally, we provide an efficient process of unification between HRM and the recent standard of hardware implementation IP-XACT.

As our purpose is to take into consideration the hardware properties during the system design, we have specified rules and constraints that govern allocation of software entities onto hardware resources. After that, we proposed mechanisms to adapt inadequate configurations.

Finally, we illustrate all these contributions within the same case study, which is a robots chain. It is real-time, embedded, multi-tasking, distributed, repetitive and configurable system.