



HAL
open science

MuVArch : une approche de méta-modélisation pour la représentation multi-vues des architectures hétérogènes embarqués

Amani Khecharem

► **To cite this version:**

Amani Khecharem. MuVArch : une approche de méta-modélisation pour la représentation multi-vues des architectures hétérogènes embarqués. Autre [cs.OH]. Université Nice Sophia Antipolis, 2016. Français. NNT : 2016NICE4019 . tel-01361473

HAL Id: tel-01361473

<https://theses.hal.science/tel-01361473>

Submitted on 7 Sep 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ NICE SOPHIA-ANTIPOLIS
ÉCOLE DOCTORALE STIC
SCIENCES ET TECHNOLOGIES DE L'INFORMATION
ET DE LA COMMUNICATION

THÈSE

pour l'obtention du grade de

DOCTEUR EN SCIENCES

Spécialité INFORMATIQUE

de l'Université Nice Sophia-Antipolis

présentée et soutenue par

AMANI KHECHAREM

le 3 Mai 2016

MU_VARCH :
**Une approche de méta-modélisation pour la
représentation multi-vues des architectures
hétérogènes embarqués**

Thèse dirigée par: Robert DE SIMONE
au sein de l'équipe commune AOSTE
INRIA Sophia-Méditerranée et UMR 7271: I3S (CNRS et Université Nice
Sophia-Antipolis)

Jury :

| | | | |
|-----------------------|------------------------|-------|---------------------------|
| M. Guy GOGNIAT | Professeur | UBS | <i>Rapporteur</i> |
| M. Jean-Pierre TALPIN | Directeur de recherche | INRIA | <i>Rapporteur</i> |
| M. Abdoulaye GAMATIE | Directeur de recherche | CNRS | <i>Examineur</i> |
| M. Michel AUGUIN | Directeur de recherche | CNRS | <i>Examineur</i> |
| M. Robert DE SIMONE | Directeur de recherche | INRIA | <i>Directeur de Thèse</i> |

Remerciements

Je souhaite tout d'abord remercier grandement mon directeur de thèse, M. Robert DE SIMONE, pour la confiance qu'il m'a témoignée en acceptant de diriger ma thèse et de m'avoir accueilli au sein de l'équipe AOSTE.

Je remercie vivement M. Guy GOGNIAT d'avoir accepté évaluer mon travail de thèse. Je suis également très honoré que M. Michel AUGUIN et M. Abdoulaye GAMATIÉ aient accepté de faire partie de mon jury.

Je souhaite aussi remercier tous les membres passés et présents de l'équipe AOSTE-INRIA ; et plus particulièrement Julien DEANTONI (pour son aide et sa manière directe d'expliquer les choses), Carlos Ernesto GÓMEZ CÁRDENAS (pour ses précieux conseils au début de ma thèse) et Patricia LACHAUME (pour sa joie de vivre et sa sollicitude).

Je tiens à remercier également toutes mes amies, ou plutôt mes soeurs, avec qui j'ai partagé ma vie en France et je me suis sentie toujours avec ma deuxième famille.

Je remercie toutes les personnes qui me sont chères et qui tiennent une grande place dans mon cœur : mes frères, mes grands parents, mes oncles, mes tantes et mes cousin(e)s, ma belle famille et tous ceux que j'aime et qui m'aiment, qui malgré la distance ont toujours été présents pour moi.

Je remercie tout spécialement mon cher mari pour son soutien sans faille et ses encouragements. Je le remercie pour avoir toujours trouvé les mots justes dans les moments difficiles et m'avoir apporté le réconfort dont j'avais besoin.

Mes remerciements finaux et non les moindres vont aux plus chères personnes dans ma vie, mes parents, qui m'ont tout enseigné, ont suivi et soutenu mon évolution depuis mes premiers pas et qui ont su me donner toutes les chances pour réussir. Leur amour et leurs perpétuels encouragements ont été la source et la force d'aboutissement de ce travail qui est en grande partie le leur, qu'il leur soit dédié.

Enfin, une pensée émue à mon cher grand père maternel et à ma chère tante qui auraient pu être fières de moi s'ils étaient toujours de ce monde.

Amani KHECHAREM
Avril 2016, Sophia Antipolis

Résumé

Nous avons défini et réalisé avec l'approche MuVarch un environnement de (meta-) modélisation orientée vers la représentation multi-vues des architectures embarquées hétérogènes (de type "smartphone" par exemple). En plus de la vue architecturale de base, support de toutes les autres, on considère les vues "performance", "consommation", "température", ainsi que la vue fonctionnelle "applicative" pour fournir des scénarios comportementaux de fonctionnement de la plate-forme. Il était important de savoir décrire en MuVarch comment les vues se raccrochent à la vue de base architecturale, et comment elle se relient également entre elles (relation entre consommation énergétique et température par exemple). L'objectif ultime est d'utiliser ce framework multi-vues et les différentes informations apportées par chacune, pour savoir supporter des politiques alternatives de mapping/allocation des tâches applicatives sur les ressources de l'architecture (la définition de ces politiques restant extérieure à nos travaux de thèse). La représentation adéquate de cette relation d'allocation forme donc un des aspects importants de nos travaux.

Mots-clés : Ingénierie Dirigée par les Modèles (IDM), Modélisation, Modélisation multi-vues, Langage de Modélisation Spécifique à un Domaine (DSML), Architecture *big.LITTLE*.

Abstract

We introduced and realized with our MuVarch approach an heterogeneous (meta)modeling environment for multi-view representation of heterogeneous embedded architectures (of "smartphone" type for instance). In addition to the backbone architectural view supporting others, we considered performance, power, and thermal view. We introduced also the functional applicative view, to provide typical use cases for the architecture. It was important to describe in MuVarch our various views would connect to the basic one, and how they would mutually relate together as well (how temperature depends on power consumption for instance). The global objective was to let the framework consider alternative mapping/allocation strategies for applicative tasks on architectural resources (although the definition of such strategies themselves was out of the scope). The appropriate form of such an allocation relation, which may be quite involved, was thus an important aspect of this thesis.

Keywords: Model Driven Engineering (MDE), Modeling, Multi-view Modeling, Domain Specific Modeling Language (DSML), *big.LITTLE* architecture

Table des matières

| | | |
|----------|---|-----------|
| 1 | Introduction | 2 |
| 1.1 | Contexte des travaux | 2 |
| 1.2 | Projet ANR HOPE | 4 |
| 1.3 | Contribution de la thèse | 5 |
| 1.4 | Contenu du mémoire | 8 |
| | | |
| 2 | Modélisation pour la Conception des Systèmes Embarqués Hétérogènes | 10 |
| 2.1 | Introduction | 11 |
| 2.2 | Généralités | 12 |
| 2.2.1 | Système Embarqué | 12 |
| 2.2.2 | Ingénierie Système et cycles de conception | 14 |
| 2.2.3 | Autour de l'Adéquation Algorithme-Architecture et du <i>Platform-Based Design</i> | 16 |
| 2.3 | Formalismes utilisés pour la modélisation du domaine | 18 |
| 2.3.1 | Modèles abstraits de l'application | 18 |
| 2.3.1.1 | Graphes de tâches | 19 |
| 2.3.1.2 | Machines à état fini (FSM) | 20 |
| 2.3.1.3 | Graphes flot de données et <i>Process Network</i> | 20 |
| 2.3.2 | Modélisation d'architecture | 22 |
| 2.3.2.1 | Langages de description des architectures | 23 |
| 2.3.2.2 | Caractéristiques des architectures | 24 |
| 2.3.3 | Méthodes d'optimisation de l'allocation | 28 |
| 2.3.4 | Systèmes de conception conjointe | 29 |
| 2.4 | Utilisation de l'IDM dans notre cadre | 33 |
| 2.4.1 | MetaObject Facility MOF | 34 |
| 2.4.2 | Unified Modeling Language UML | 34 |
| 2.4.2.1 | SysML | 35 |
| 2.4.2.2 | MARTE | 36 |
| 2.4.3 | Langage dédié DSL | 36 |
| 2.4.4 | Cas spécial de la modélisation multi-vues | 38 |
| 2.5 | Conclusion | 40 |

| | | |
|----------|--|-----------|
| 3 | MuVArch : une Approche Multi-Vues pour la Modélisation des Systèmes Embar- qués Hétérogènes | 42 |
| 3.1 | Introduction | 43 |
| 3.2 | Principes de notre modélisation | 44 |
| 3.3 | Porte vues : Vue architecturale de base | 49 |
| 3.4 | Autres vues spécialisées | 51 |
| 3.4.1 | Aspects communs des autres vues | 51 |
| 3.4.2 | Spécialisation des vues non-fonctionnelles | 54 |
| 3.4.2.1 | Vue performance | 54 |
| 3.4.2.2 | Vue puissance | 55 |
| 3.4.2.3 | Vue thermique | 56 |
| 3.4.3 | Spécialisation de la vue applicative | 57 |
| 3.5 | Relations entre vues : Abstraction et Association | 58 |
| 3.5.1 | Association | 58 |
| 3.5.2 | Abstraction | 60 |
| 3.6 | Contrôleur général de MUARCH et Relation d'Allocation | 61 |
| 3.6.1 | Contraintes issues des relations de MUARCH | 62 |
| 3.6.2 | Relation d'Allocation effective | 63 |
| 3.6.3 | Représentation alternative d'allocation comme FSM | 66 |
| 3.7 | Conclusion | 68 |
| 4 | Application et Validation : MuVArch framework | 70 |
| 4.1 | Introduction | 71 |
| 4.2 | Atelier logiciel pour l'instrumentation de MUARCH | 71 |
| 4.3 | Présentation de l'architecture <i>big.LITTLE</i> | 74 |
| 4.4 | Modélisation de notre cas d'étude avec MUARCH | 77 |
| 4.4.1 | Modélisation de la Porte Vues (<i>Backbone View</i>) | 80 |
| 4.4.2 | Modélisation de la vue Performance | 81 |
| 4.4.3 | Modélisation de la vue Puissance | 82 |
| 4.4.4 | Modélisation de la vue Thermique | 83 |
| 4.4.5 | Modélisation de la vue Application | 84 |
| 4.4.6 | Prise en compte du contrôleur général en Sirius | 85 |
| 4.5 | Connexion avec un solveur SMT existant | 87 |
| 4.5.1 | Génération du code pour le Solver SMT | 87 |
| 4.5.2 | Calcul des contraintes et de l'ordonnancement | 90 |
| 4.5.3 | Prise en compte de l'ordonnancement par le contrôleur de MUARCH | 91 |
| 4.6 | Conclusion | 96 |
| 5 | Bilan et Perspectives | 98 |
| 5.1 | Bilan | 98 |

| | |
|--|------------|
| 5.2 Perspectives | 100 |
| A Exécution des modèles hétérogènes pour des analyses thermique | 102 |
| A.1 Contexte des analyses | 103 |
| A.2 Co-Simulation conjointe | 104 |
| A.3 Étude de cas : Contrôleur thermique d'un processeur | 107 |
| Table des Figures | 111 |
| Liste des Tableaux | 114 |
| Abréviations | 116 |
| Bibliographie | 126 |

INTRODUCTION

Sommaire

| | | |
|------------|---------------------------------|----------|
| 1.1 | Contexte des travaux | 2 |
| 1.2 | Projet ANR HOPE | 4 |
| 1.3 | Contribution de la thèse | 5 |
| 1.4 | Contenu du mémoire | 8 |

1.1 Contexte des travaux

A partir de l'année 1958, l'électronique a connu une évolution importante suite à la réalisation du tout premier circuit intégré par *Jack Kilby* [1] chez *Texas Instruments*. Depuis, ces circuits évoluent d'une manière très rapide. Un circuit intégré peut contenir aujourd'hui quelques millions de transistors, voire quelques dizaines de millions comme l'illustre la figure 1.1. Cette évolution n'est pas prête de s'arrêter, au vu de l'étude délivrée par la SIA¹ [2]. Aujourd'hui, cette course à la miniaturisation a permis aux concepteurs de concevoir des systèmes de plus en plus complexes, possédant des performances et fonctionnalités extraordinaires. Désormais, il est possible d'intégrer un système complet sur une seule puce. On parle alors de Système sur puce ou systèmes mono-puces (SoC, pour *System-On-Chip*).

1. SIA : Semiconductor Industry Association

La complexité des SoCs actuels augmente avec l'émergence de nouvelles applications télécommunications et multimédia avec des exigences fonctionnelles de plus en plus sévères (dépendance de données) et non fonctionnelles (temps, énergie, coût...). Les SoCs sont connectés ainsi à tous les domaines notamment dans les systèmes embarqués ou les systèmes cyber-physiques. De tels systèmes sont dits hétérogènes. Le téléphone mobile est un exemple de tel système ; il est toujours de plus en plus complexe et est également sensé non seulement avoir une petite taille et être léger, mais aussi fournir une batterie de longue durée.

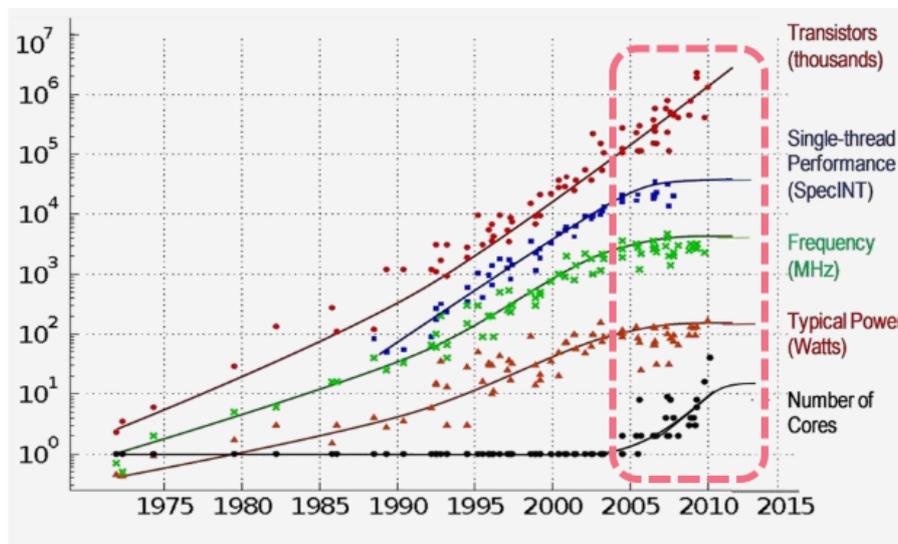


FIGURE 1.1: Évolution proportionnelle du nombre de cœurs par rapport à l'énergie, la fréquence et la performance pour les SoCs [3]

La figure 1.1 illustre l'évolution proportionnelle de nombre de cœurs par rapport à la densité de consommation d'énergie, de fréquence et de performance pour les systèmes sur puce modernes. Notons qu'une dissipation de puissance et les effets thermiques associés sont ainsi les principaux freins à une augmentation des performances embarquées dans un circuit. La maîtrise de la complexité croissante des exigences de ces systèmes est l'un des enjeux majeurs des concepteurs. En effet, définir une architecture logicielle/matérielle de tels systèmes ayant le niveau de performance exigé par la vue applicative du système avec une ou des stratégies de gestion de la consommation peut s'avérer complexe et, plus ces stratégies sont optimisées, plus les chances d'introduire des erreurs de conception sont grandes. Cependant, la conception de ces systèmes ne peut être implémentée par des méthodes classiques où la conception du matériel précède celle du logiciel. Ces méthodes

sont incompatibles avec les contraintes de temps de mise sur le marché actuelles. La conception des SoCs peut être décrite en utilisant différents langages de programmation et selon différents niveaux d'abstraction : algorithmique, architectural, transfert de registre, logique ou physique (cités du plus haut vers le plus bas niveau d'abstraction).

Face à cette complexité des systèmes et leurs perpétuelles progressions, les concepteurs doivent faire preuve d'une grande réactivité et doivent ainsi améliorer leurs techniques de conception d'une manière continue dans un délai de mise sur le marché (*time to market*) réduit. Dans ce contexte, une méthodologie de conception haut niveau s'avère la solution la plus efficace pour faciliter le travail des concepteurs. Au niveau système par exemple, le concepteur peut traiter des applications hautes performances avec une maîtrise totale des technologies sans se soucier des détails techniques du niveau physique.

1.2 Projet ANR HOPE

Les travaux présentés dans ce manuscrit s'inscrivent dans le cadre d'un projet ANR ² (Agence Nationale de la Recherche), au nom de HOPE ³ (Hierarchically Organized Power/Energy management). Celui ci est un projet collaboratif avec les partenaires industriels Docea-Power ⁴, Magillem ⁵, Synopsys ⁶, Intel ⁷ et la Plateforme Conception ARCSIS ⁸ et les partenaires académiques INRIA (l'équipe AOSTE ⁹) et LEAT ¹⁰. A l'origine, le projet HOPE comprenait également comme partenaire majeur la société Texas Instruments ¹¹, qui a malheureusement fermé ses activités basées en France près de Sophia-Antipolis. Cette société portait en partie le projet en proposant des schémas typiques d'applications à un très haut degré d'abstraction sous la forme de scénarios d'utilisation d'un smartphone sur un jour (entre deux recharges de batterie), aussi qu'une description

2. www.agence-nationale-recherche.fr/

3. <http://anr-hope.unice.fr>

4. <http://www.doceapower.com/>

5. <http://www.magillem.com/>

6. <http://www.synopsys.com/home.aspx>

7. <http://www.intel.fr/content/www/fr/fr/homepage.html>

8. <http://www.pf-conception.org/>

9. <https://team.inria.fr/aoste/>

10. <http://leat.unice.fr/>

11. www.ti.com/fr

des mécanismes de gestion d'énergie de leur processeur embarqué OMAP, en particulier dans ses aspects hiérarchiques.

Le projet HOPE a pour objectif d'introduire une méthodologie complète de modélisation d'architecture *power* de systèmes sur puce. Les travaux de ce projet portent principalement sur :

- La modélisation abstraite et les techniques associées pour explorer des solutions offrant de bons compromis performance/énergie(puissance)/température. Ce point consistera le contexte des travaux de ce mémoire.
- La validation au niveau TLM de la ou des solutions précédentes en considérant le système complet : architectures matérielle, logicielle et énergie(puissance).
- La proposition d'une approche de hiérarchisation du *power management*.

1.3 Contribution de la thèse

Il existe actuellement une activité importante autour des principes de co-simulation des systèmes cyber-physiques (d'une part multi-physiques, avec des composants distincts modélisés dans des domaines continus différents, mais également avec une modélisation discrète des comportements de contrôleurs et processeurs numériques). L'objet de la présente thèse est triple :

- D'une part, étudier les capacités et les besoins en **co-modélisation**, en amont de la co-simulation, afin de prendre en compte globalement la cohérence des définitions du système (et éventuellement effectuer des analyses sur la conjonction d'objets mathématiques) ;
- D'autre part, considérer le cas dans cette co-modélisation où des modèles distincts peuvent, non pas représenter forcément des composants distincts interagissant, mais possiblement des **aspects différents d'un même composant (ses "vues")**. Dans ce dernier cas, le couplage entre les différentes vues-modèles qui interviennent pour décrire un composant sera encore plus fort, s'appuyant sur des éléments structurels de représentation communs, ce qui rendra d'autant plus impératif le besoin de co-modélisation cohérente et intégré.

- Enfin, combiner dans un même formalisme de représentation une étape où les modèles peuvent être exploités afin d'étudier par analyse fonctionnelle les différents options possibles pour optimiser efficacement l'allocation des parties applicatives sur les ressources architecturales, puis dans une seconde étape (toujours dans le même environnement de représentation) permettre de produire des scénarios de simulation effectives pour évaluer dynamiquement les allocations proposées. On voit que cette approche réclame idéalement de connecter nos modèles, d'une part avec des outils dédiés d'analyse (mapping, ordonnancement) pour calculer les allocations, d'autre part de simulation hybride pour évaluer plus finement les différents aspects non-fonctionnelles sur nos vues.

Nos travaux décrits dans ce mémoire proposent une approche, nommée *MU*ARCH et résumée dans la figure 1.2, de conception dirigée par les modèles pour la conception des systèmes embarqués à haut niveau destinée à l'analyse (performance, énergie, ordonnancement et placement), et également à la simulation qui exploite ensuite ce premier niveau d'abstraction. Nous utilisons les technologies de l'Ingénierie Des Modèles (IDM) tels que EMF et Sirius pour notre environnement de modélisation, en nous raccrochant à des environnements logiciels développés dans notre équipe d'accueil comme *TIMESQUARE* [4]¹² ou des environnement CAO tels Synopsys MCO¹³ ou *Scilab*¹⁴.

Les caractéristiques (et aussi les parti-pris) de *MU*ARCH sont les suivants : sur une vue architecturale de base, qui représente la structure en termes de composants du système, nous “plaquons” les vues de domaines plus spécifiques, extra fonctionnels (performance/fréquence d'horloge, puissance/tension, température,...). Ces vues sont décrites de manière aussi similaires entre elles que possible, et nous décrivons ensuite également la vue fonctionnelle (les abstractions d'applications venant exercer la plate-forme) également selon les mêmes principes. Les trois questions principales que se posent alors à nous sont :

1. comment “raccrocher” les différentes vues à la vue de base afin de définir et de conserver les bonnes informations qui seront utiles et nécessaires à notre objectif ultime, l'étude du mapping/allocation entre les tâches et fonctions applicatives et

12. <http://timesquare.inria.fr/>

13. <http://www.synopsys.com/Prototyping/ArchitectureDesign/Pages/platform-architect.aspx>

14. <http://www.scilab.org/fr>

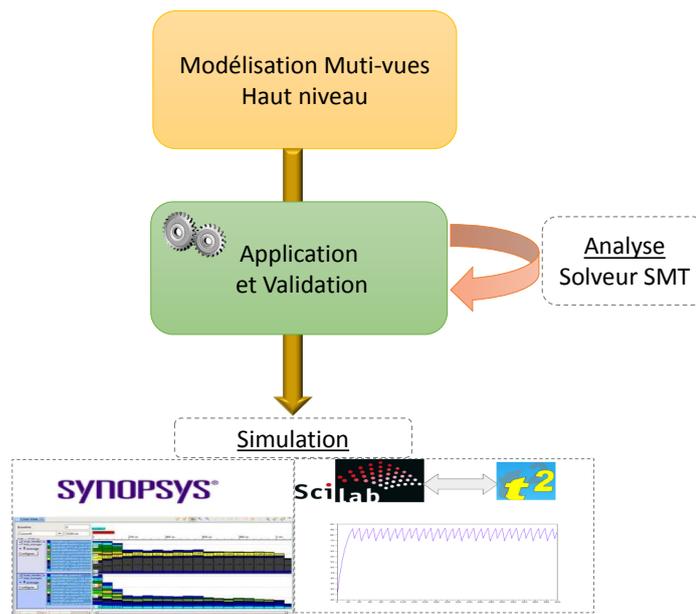


FIGURE 1.2: Illustration générale des contributions

les ressources architecturales qui optimisent (ou du moins respectent) les contraintes des vues extra-fonctionnelles ;

2. comment exprimer (ou du moins représenter) les relations existantes entre ces vues non-fonctionnelles (par exemple la température dépend de la consommation électrique, qui elle-même varie suivant la fréquence d'horloge,...) ;
3. comment définir et représenter la relation d'allocation elle-même, entre cette fois la vue fonctionnelle des applications et la vue architecturale de base. Ce troisième point est d'une importance critique pour notre travail : cette relation d'allocation est bien souvent traitée d'une manière simpliste (voire désinvolte) dans la littérature MDA (comme une simple flèche tâche \rightarrow IP bloc), ce qui dénote sans doute un déficit d'utilisation de leur propre formalisme par ses promoteurs. En réalité cette relation incarne, au premier niveau mentionné plus haut, toute l'algorithmique de l'ordonnancement et de la distribution de tâches. Et une fois calculé ce résultat (ce qui, encore une fois, n'est pas du ressort de cette thèse), la relation d'allocation peut avoir à représenter une solution où les diverses instances d'une tâche peuvent se voir allouer sur divers composants de calcul, où les valeurs des vues non fonctionnelles peuvent également varier selon ces instances, et où les allocations de communications

et de stockage de données devraient être également traitées (faute de temps nous ne traiterons pas ce dernier point dans cette thèse).

1.4 Contenu du mémoire

Ce mémoire est organisé en trois chapitres :

- Dans le chapitre 2 intitulé “**Modélisation pour la conception des systèmes embarqués hétérogènes**”, nous introduisons les notions et expressions clés utilisées dans ce manuscrit. Nous présentons, au début, les méthodologies de conception des systèmes embarqués hétérogènes. Ensuite, nous décrivons les principaux modèles pour la description des applications et architectures embarqués tout en détaillant l’analyse des besoins (performance, énergie et ordonnancement) auxquels devra répondre notre approche. Enfin, nous détaillons l’IDM, pilier de notre approche, ainsi que des approches relatives telles que UML et ses profils. Nous nous intéressons spécialement pour la modélisation multi-vues.
- Dans le chapitre 3 intitulé “**MuVArch : Une approche multi-vues pour la modélisation des systèmes embarqués hétérogènes**”, nous décrivons l’approche MUARCH pour la modélisation multi-vues de système que nous avons développée. Après avoir introduit les concepts fondamentaux sur lesquels s’appuie notre approche, nous détaillons les différents éléments qui la composent, à savoir la vue de base architectural, les différents vues (performance, puissance, thermique et application) qui s’y accrochent ainsi que leurs interconnexions. Particulièrement, nous montrons que la relation d’allocation des tâches applicatives vers les ressources architecturales, se faisant dans le contrôleur général du système, s’avère plus complexe et présente le comportement du système après calcul d’ordonnancement par un solveur SMT tiers. Nous obtenons donc finalement deux besoins de représentation au niveau de ce contrôleur : en amont, il nous faut extraire de toutes les relations entre vues (intra-vue et inter-vues) une expression des contraintes qui viennent alimenter un solveur pour calculer une allocation ;

en aval, il faudra représenter en retour dans l’environnement de modélisation ce résultat d’allocation.

- dans le chapitre 4 intitulé “**Application et Validation : MuVArch framework**”, nous présentons l’instrumentation que nous avons implémenté de notre approche sous la forme d’un framework. Ce chapitre comprend tout d’abord la chaîne d’instrumentation pour la conception du framework. Ensuite, nous illustrons la mise en oeuvre de notre approche MuVARCH sur un cas d’étude dont nous détaillons les différents aspects et son application dans notre framework ainsi que sa connexion avec un solveur SMT existant pour le calcul de l’ordonnement adéquat.

Enfin, nous concluons et nous formulons quelques perspectives concernant l’ensemble de nos travaux. Notons que le chapitre en annexe A représente le résultat des travaux avec une version plus ancienne de modélisation nommée PRISMSYS et développé par *Carlos Ernesto Gómez Cárdenas*, ancien doctorant dans l’équipe AOSTE, dans sa thèse [5].

MODÉLISATION POUR LA
CONCEPTION DES SYSTÈMES
EMBARQUÉS HÉTÉROGÈNES

Sommaire

| | | |
|------------|---|-----------|
| 2.1 | Introduction | 11 |
| 2.2 | Généralités | 12 |
| 2.2.1 | Système Embarqué | 12 |
| 2.2.2 | Ingénierie Système et cycles de conception | 14 |
| 2.2.3 | Autour de l'Adéquation Algorithme-Architecture et du <i>Platform-Based Design</i> | 16 |
| 2.3 | Formalismes utilisés pour la modélisation du domaine | 18 |
| 2.3.1 | Modèles abstraits de l'application | 18 |
| 2.3.1.1 | Graphes de tâches | 19 |
| 2.3.1.2 | Machines à état fini (FSM) | 20 |
| 2.3.1.3 | Graphes flot de données et <i>Process Network</i> | 20 |
| 2.3.2 | Modélisation d'architecture | 22 |
| 2.3.2.1 | Langages de description des architectures | 23 |
| 2.3.2.2 | Caractéristiques des architectures | 24 |
| 2.3.3 | Méthodes d'optimisation de l'allocation | 28 |
| 2.3.4 | Systèmes de conception conjointe | 29 |

| | |
|--|-----------|
| 2.4 Utilisation de l’IDM dans notre cadre | 33 |
| 2.4.1 MetaObject Facility MOF | 34 |
| 2.4.2 Unified Modeling Language UML | 34 |
| 2.4.2.1 SysML | 35 |
| 2.4.2.2 MARTE | 36 |
| 2.4.3 Langage dédié DSL | 36 |
| 2.4.4 Cas spécial de la modélisation multi-vues | 38 |
| 2.5 Conclusion | 40 |

2.1 Introduction

Notre démarche s’appuie sur un certain nombre de principes de modélisation, qui co-existent et sont souvent déjà utilisés individuellement, mais que nous visons à combiner pour notre approche. Dans un premier temps (sous-section 2.2.1), nous relevons certaines caractéristiques du domaine des systèmes embarqués, qui justifient leurs besoins en terme de représentation (applications et logiciels réactifs, architectures hétérogènes, modélisation de la physique de l’environnement, contraintes non-fonctionnelles comme le temps-réel, la consommation, la température, etc...). Nous introduisons ensuite (en sous-section 2.2.2) la thématique de l’Ingénierie Système (*System Engineering*) comme discipline visant justement à définir et manipuler des modèles, généralement en phase amont de spécification. L’approche dite MBSE (*Model-Based System Engineering*) forme le cadre de nos travaux (sous section 2.2.3), plus spécifiquement sous l’angle *Platform-Based Design (PBD)*, ou *Adequation Algorithm-Architecture (AAA)*, qui visent à apparier au mieux un modèle abstrait d’application avec un modèle également abstrait d’architecture, et ce sous des contraintes décrites par d’autres modèles (qualifiés de “*vues*”) exprimant les impératifs non fonctionnels (performance, consommation, température, etc...).

La section 2.3 est dédiée à l’étude plus détaillée des catégories de modèles identifiés en 2.2.1. Nous débutons (sous-section 2.3.1) par une description (non exhaustive car le domaine est très large) des principaux modèles utilisés pour la description abstraites d’applications embarquées, souvent de nature *data/signal streaming*. La sous-section 2.3.2

couvre les modèles d'architecture (ou encore plates-formes d'exécution) , souvent représentés de manière plus ad-hoc voire implicite ; et nous soulignons aussi des différents modèles de propriétés et contraintes non-fonctionnelles (environnement physiques, vues thermique, performance ou consommation du système,..) Ensuite, la section 2.3.3 traite des modèles d'allocation permettant en général de rapporter la distribution dans l'espace et l'ordonnement dans le temps des diverses parties applicatives sur l'architecture (ce qui est en général établi, soit manuellement par le concepteur, soit par le biais de techniques d'optimisation sous les contraintes des vues non-fonctionnelles). Enfin dans 2.3.4, nous présentons quelques outils et méthodes participant dans ce domaine.

La section dernière 2.4 est dédiée à l'étude des environnements et langages pour la méta-modélisation dans notre contexte, c'est-à-dire autorisant la définition et la combinaison de modèles pour les inclure dans une démarche d'ingénierie. Nous introduisons spécialement le cas spécial de la modélisation multi-vues proposée pour modéliser notre approche MU_VARCH.

2.2 Généralités

2.2.1 Système Embarqué

Suivant la définition de [6], nous appelons un **système** “*un ensemble, une collection organisée (possédant une structure) d'objets reliés ou branchés (en interrelation) les uns aux autres, de façon à former une entité ou un tout remplissant une ou plusieurs fonctions. Un système est un produit artificiel de l'esprit des hommes*”. Cet ensemble forme donc un tout cohérent, intégré pour assurer une ou plusieurs fonctions correspondant aux besoins du système par rapport à son environnement.

Particulièrement, un système est dit **embarqué** lorsqu'il est intégré dans un système plus large avec lequel il est connecté et pour lequel il réalise une ou plusieurs fonctions particulières telles que le contrôle, l'impression, la communication, la surveillance, etc. Un système embarqué comprend des logiciels et des matériels conjointement et spécifiquement

conçus pour assurer ces fonctionnalités. Par suite, les systèmes embarqués sont essentiellement **hétérogènes** [7]. Nous pouvons distinguer les systèmes embarqués en différents types selon les contraintes auxquels ils peuvent être exposés. Nous identifions par exemple les systèmes temps réel, les systèmes distribués et mobiles et les systèmes cyber-physique. Nous nous intéressons par la suite aux derniers systèmes.

Un **Système Cyber-Physique CPS** (CPS pour *Cyber Physical System*) [8] [9] représente un système d'éléments de calcul qui collaborent dans le but d'assurer la coordination, la surveillance, et la commande de processus physiques. Il s'agit donc d'un système embarqué de nature hétérogène basé sur des systèmes dynamiques discrets pour les composants numériques et des équations différentielles pour les composants physiques. Contrairement à un système embarqué classique qui met l'accent plus sur les éléments de calcul, pas sur le lien entre les éléments de calcul et physiques, un CPS est un système profondément combiné entre le calcul et le système physique. L'amélioration de la liaison entre ces deux éléments en utilisant les progrès de la science et de l'ingénierie va stimuler l'utilisation des CPS. Plusieurs applications de l'utilisation des CPS tels que l'aéronautique, l'automobile, les processus chimiques, de l'infrastructure civile, l'énergie, les soins de santé, la fabrication, le transport, le divertissement et les appareils électroménagers.

Pour plus clarifier ces définitions, prenons l'exemple d'un téléphone portable, tel que celui présenté dans la figure 2.1. Ce téléphone est un système relativement complexe car, même s'il n'est pas composé d'un grand nombre d'éléments, il est composé de nombreux sous systèmes (batterie, radio sans fil, micro, amplificateur, etc); en plus il mélange de nombreux domaines physiques tels que le traitement du signal, le logiciel, l'électronique, la gestion de l'énergie, les réseaux, etc. Pour la conception de ce téléphone portable, ce système sera divisé en sous système. Chaque sous système sera un point de vue spécifique pour les experts. Ainsi, un expert travaille sur la chaîne de traitement de signal, un autre s'intéresse à l'aspect énergétique et un autre expert se préoccupe des applications logicielles embarquées par exemple.

Le développement de méthodes et d'outils permettant la gestion, la réalisation et l'association de tous ces sous systèmes est l'objectif de l'ingénierie système [10], que l'on présentera en détails dans la section suivante.

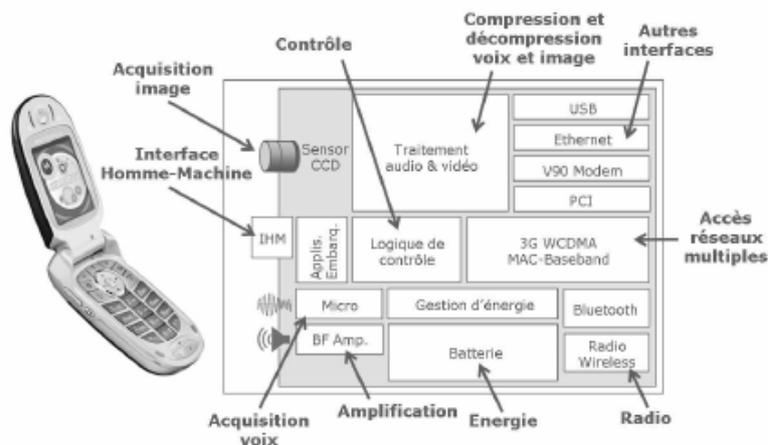


FIGURE 2.1: Exemple d'un système embarqué hétérogène : Téléphone portable

2.2.2 Ingénierie Système et cycles de conception

D'après [10], l'**ingénierie système** "*est une approche interdisciplinaire qui englobe tous les efforts techniques pour développer et vérifier un ensemble de solutions relatives aux systèmes, aux utilisateurs et aux processus dans un cycle de vie total et intégré pour satisfaire des besoins client.*" De ce fait, la démarche d'ingénierie système embrasse l'ensemble du cycle de vie du système en coordonnant des actions interdisciplinaires utiles pour ce faire. Il existe plusieurs méthodologies proposant des approches différentes de l'ingénierie système, notamment à travers des définitions différentes du **cycle de vie** du système tel que le cycle en cascade, le cycle en spirale et le cycle en V. Nous détaillons par suite le cycle en V.

Le cycle en V (ou parfois *ladder design cycle*), comme illustré dans la figure 2.2, comprend une partie descendante de raffinement en étapes depuis un cahier des charges très amont jusqu'à une réalisation détaillée, puis une remontée en vérifiant à rebours la correction des étapes, depuis les test unitaires jusqu'aux tests d'intégration et à la recette du produit final.

Les méthodologies de ces cycles de conception sous-tendent largement des formalismes et environnements de conception de l'ingénierie des modèles au niveau système, tels que UML [11]/ SysML[12]. L'enchaînement des étapes de raffinement y reste implicite,

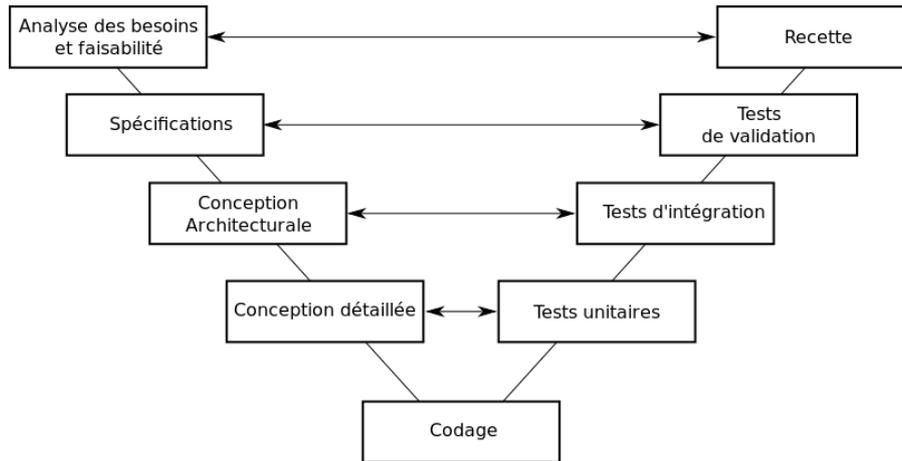


FIGURE 2.2: Modèle de cycle de développement en V

car elle doit surtout refléter l'organisation des tâches au sein de la structure de développement (souvent plusieurs sociétés avec des donneurs d'ordre et des sous-traitants). Le formalisme *ARCADIA* (*ARChitecture Analysis et Design Integrated Approach*)/*Melody Advance*¹ [13], développé par *Thales* et désormais disponible en projet Open Source sous *Eclipse PolarSys*², tente d'instrumenter ces liens de traçabilité au cours de raffinement successifs.

Les méthodologies de conception à base de cycle en *V/ladder* distinguent toujours, dans le processus de raffinement d'un système global, une phase de raffinement fonctionnel suivi d'un raffinement architectural. Dans le second temps, une description de l'architecture est donc nécessaire, sans que ce soit toujours explicite (ni que cette description soit toujours détaillée). Nous voulons noter ici que, bien que l'approche se présente comme une descente de raffinement (du système plutôt applicatif vers sa version allouée sur des ressources architecturales), la représentation de l'architecture elle-même reste du même niveau d'abstraction que celle du système, et que donc cette approche reste intégralement compatible avec nos travaux : en anticipant quelque peu sur des développements de la suite de ce chapitre, on verra que les termes de corrélation entre tâches et communications d'application vers les ressources de calcul et d'interconnexion de l'architecture ont l'effet de grouper et partitionner les premières vers des éléments uniques de la seconde, alors que les vues extra-fonctionnelles (performance, puissance, thermique) vont grouper

1. www.polarsys.org/capella/arcadia.html

2. <https://www.polarsys.org/capella/arcadia.html>

et partitionner les ressources architecturales vers des domaines (d’horloge, de puissance, de *floorplan* thermique).

2.2.3 Autour de l’Adéquation Algorithme-Architecture et du *Platform-Based Design*

La naissance de l’informatique (au-dessus des machines électroniques) vient de la possibilité d’exécuter modulairement des programmes multiples sur une même architecture matérielle. L’ajustement d’un algorithme logique (logiciel) sur une architecture d’exécution donnée est traditionnellement dénommé compilation. Dans le domaine de l’Ingénierie des Systèmes Dirigée par les Modèles (IDM) on parlera plutôt d’*Allocation*.

La plupart des formalismes (SysML [12], AADL [14], MARTE [15], ARCADIA [13]) représentent l’allocation comme une relation (flèche) pointant depuis une tâche ou une fonction d’un modèle de l’application vers une ressource (de calcul généralement) de l’architecture (figure 2.3 (a)). Bien qu’utile, cette forme nous semble trop naïve. Dans l’approche Adéquation Algorithme-Architecture (AAA) préconisé au sein de l’équipe de recherche AOSTE³, et souvent également nommée “Platform-Based Design” (PBD) dans la littérature, on reconnaît que, pour se définir, une telle allocation nécessite la connaissance a priori d’un modèle de l’architecture, défini à un niveau d’abstraction similaire de celui de la description de l’application, et qui sert à guider les choix de placement et d’ordonnancement inclus dans la notion d’allocation. D’autre part, le résultat de ce processus est bien plutôt une version raffinée de l’application, qui consiste à recouvrir sa description de haut niveau par des informations additionnelles extraites d’éléments architecturaux (figure 2.3 (b)).

Le résultat pourrait être qualifié de “*architecture-aware application*”. On peut noter ici que cette approche est en fait tout autant applicable au cas de la compilation de code, où si l’algorithme originel est bien aussi indépendant de l’architecture, le compilateur lui-même (et ses phases d’optimisation) ont bien été construit en connaissance de la

3. <https://team.inria.fr/aoste/>

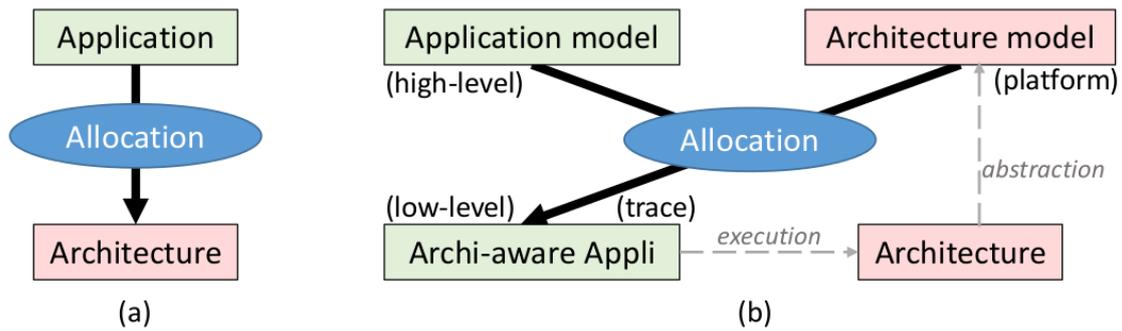


FIGURE 2.3: Allocation

plate-forme d'exécution visée (même si cela reste implicite dans l'utilisation faite du compilateur) ; et, enfin, le code de haut niveau représentant une algorithmique indépendante de la machine est bien transformé en code-machine de bas niveau, lui très dépendant de l'architecture matérielle cible.

Il existe en fait deux types d'usage très distincts de l'approche AAA/PBD, suivant qu'on cherche à optimiser le placement de l'application sur une architecture "immuable", ou au contraire qu'on cherche à optimiser le design d'une nouvelle architecture embarquée en fonction des cas d'utilisation ("use cases") prévisibles. Dans le second cas, les applications pourront représenter plutôt les multiples activités et trafics effectués sur la plate-forme qu'une vision strictement algorithmique d'une application unique.

Enfin, même dans une approche orientée par les modèles, il existe des niveaux d'abstraction forts divers où peut s'appliquer l'approche AAA/PBD. A un niveau très abstrait, il s'agit essentiellement de dimensionnement et de choix de design au long de phases amont de la conception. A un niveau plus concret, on peut vouloir fournir un modèle très précis mais avant fabrication de la plate-forme architecturale elle-même (un "prototype virtuel") afin de mettre au point le logiciel en avance de phase, pour être prêt à l'installation dès la livraison des premiers "chips" physiques.

En conclusion de cette section, il nous faut rappeler que le résultat du processus d'allocation sera, tout au long de ce document, une version raffinée de l'application sur laquelle des informations supplémentaires auront été apportées à partir des propriétés fonctionnelles et non-fonctionnelles de l'architecture. Pour bien poser ce point, et en anticipant quelque peu les définitions à venir, donnons quelques exemples :

- on peut parfois souhaiter que les occurrences distinctes d’une même tâche puissent être allouées sur des ressources différentes, suivant le contexte courant de leur exécution.

Au final, l’allocation visera à décrire une “trace”, c’est-à-dire une forme d’exécution symbolique de l’application, généralement ultimement périodique, et suffisamment décorée par les informations extraites de l’architecture au cours du calcul de l’allocation. Cette description peut prendre plusieurs formes, souvent plus complexes qu’une relation directe (application flèche architecture (figure 2.3 (a))), comme on le verra plus tard. Dans les sections suivantes nous allons revenir sur tous les ingrédients de modélisation nécessaires à cette description et à notre approche...

- de même, on peut établir lors du calcul de l’allocation elle-même les différents niveaux de performance requis (vitesse du processeur,...), et donc devoir les représenter comme résultats de l’allocation.

Au final, l’allocation visera à décrire une “trace”, c’est-à-dire une forme d’exécution symbolique de l’application, généralement ultimement périodique, et suffisamment décorée par les informations extraites de l’architecture au cours du calcul de l’allocation. Cette description peut prendre plusieurs formes, souvent plus complexes qu’une relation directe (application flèche architecture), comme on le verra plus tard. Dans les sections suivantes nous allons revenir sur tous les ingrédients de modélisation nécessaires à cette description et à notre approche.

2.3 Formalismes utilisés pour la modélisation du domaine

2.3.1 Modèles abstraits de l’application

Le partitionnement opère sur un modèle de l’application (ou une spécification). Le choix du formalisme pour décrire l’application a donc un fort impact sur la qualité des résultats. Une spécification peut être définie par un modèle de calcul MoC (*Model of Computation*) [16]. En effet, *E. A. Lee* et *A. L. Sangiovanni-Vincentelli* introduisent les

modèles de calculs dans [16] comme un ensemble des lois qui régissent l'interaction des composants dans le modèle. Particulièrement, pour modéliser les systèmes embarqués, les modèles de calcul doivent être apte de gérer la concurrence, la réactivité et le temps. En effet, tels systèmes reposent typiquement en un ensemble de ressources s'exécutant simultanément et opérant dans un environnement réel où le temps de réponse à leurs stimuli est aussi important que l'exactitude de la réponse. Le choix de modèle de calcul dépend de spécification souhaité. Ainsi par exemple, pour modéliser un système mécanique, la sémantique doit être apte de gérer la concurrence et la continuité temporelle, dans ce cas un modèle de calcul à temps continu comme celui trouvé dans *Simulink*, *VHDL-AMS* est plus approprié. Pour une modélisation purement calcul, des langages de programmation comme *C*, *C++*, *Java* et *Matlab* seraient adéquates.

Dans [16], les propriétés de différents modèles de calculs sont exposées afin de pouvoir les comparer. Ces propriétés différencient le type de temps utilisé (discret ou continu), le mode de communication (rendez-vous, messages, événements ou etc.) ou aussi le type de synchronisation utilisé (synchrone ou asynchrone). Nous nous exposons dans la suite les modèles de calculs : les graphes de tâches, les machines à états fini FSM et les graphes de flot de données et *Process Network*. Notons que les deux premiers MoCs sont orientés contrôle tandis que le dernier est orienté traitement.

2.3.1.1 Graphes de tâches

Pour tenir compte des évolutions des systèmes embarqués qui intègrent généralement plusieurs processeurs et au moins un système d'exploitation, différentes approches telles que [17] [18] [19] [20], considèrent un modèle de type graphe de tâches avec un ordonnancement statique en ligne préemptif. Les nœuds dans le graphe de tâche représentent les processus qui exécutent les opérations. Ces processus transforment les flots de données en entrée en flot de données en sortie. Les arcs représentent les relations entre les processus. Généralement, les tâches sont de forte granularité et traitées comme des boîtes noires avec des conditions de précédence. C'est le système d'exploitation qui gère l'exécution des tâches en utilisant ses propres structures de contrôle.

2.3.1.2 Machines à état fini (FSM)

Le modèle des machines à états finis (*Finite State Machine FSM*) [21] est très répandu et bien utilisé pour les systèmes comportant une forte dominante de contrôle. Les FSM consistent en un ensemble d'états, un ensemble d'entrées et de sorties, une fonction qui définit les sorties en termes d'entrées et d'états, et un contrôle qui active les fonctions suivant la séquence voulue. Du fait de leur nature finie, les FSM conviennent bien à l'analyse formelle et par conséquent, aux activités permettant de vérifier l'absence de comportements inattendus du système. Cependant, l'une des lacunes de ce modèle réside dans le fait que le nombre d'états possibles augmente rapidement en fonction de la complexité des systèmes. D'autre part, les FSM simples ne permettent pas de modéliser les comportements concurrents (parallélisme). Pour remédier à ces faiblesses, des extensions et des variantes du modèle simple FSM ont été introduites, citons en particulier : les CFSM (*Co-Design Finite State Machines*), proposé par Chiodo et al. [22] et utilisés dans l'environnement Polis [23], les *StateCharts* introduits par Harel [24] et les *SpecCharts* introduits par Gajski et al. [25]. Ces extensions peuvent porter sur la concurrence, la hiérarchie et la communication entre les FSMs. Outre ces extensions apportées aux FSM, il est possible d'utiliser les machines à états finis en combinaison avec d'autres modèles de calcul afin de surmonter leurs faiblesses.

2.3.1.3 Graphes flot de données et *Process Network*

Le modèle graphe de flot de données DFG (*Data Flow Graph*) est utilisé pour représenter les dépendances de données d'une application. Le DFG est un graphe orienté où les sommets sont des tâches et les arcs sont des liaisons de données entre les calculs réalisés par les tâches. Une tâche est prête à être exécutée dès que toutes ses entrées seront disponibles, elle consomme les données reçues sur chacun de ses arcs d'entrée, fait son calcul et produit des données sur chacun de ses arcs en sorties.

Parmi les modèles flots de données les plus fréquemment utilisés, nous citons les réseaux de Kahn (ou KPN pour *Kahn Process Network*) [26]. Ce modèle, illustré dans la figure 2.4 se compose d'un ensemble de processus qui communiquent exclusivement au

travers de canaux de communication FIFO⁴ non bornées, où seul le processus d'entrée peut écrire et seul celui en sortie peut lire. Les KPN sont les modèles flots de données les plus répandus pour décrire des applications, mais le fait que la quantité de mémoire nécessaire à l'exécution des processus est indéterminé ne les permet pas d'être adéquats pour être appliqués à certains domaines de l'informatique embarquée.

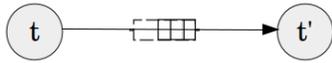


FIGURE 2.4: KPN à deux processus

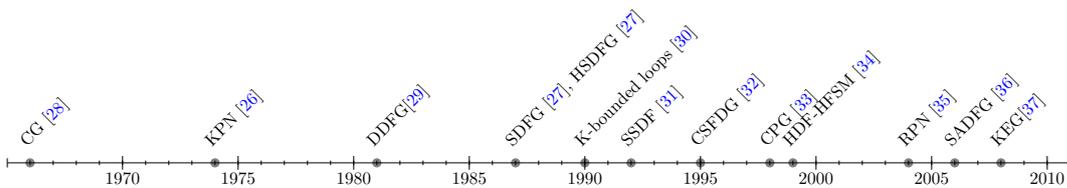


FIGURE 2.5: Modèle SDF à deux tâches

Un autre modèle graphe de flot de données répandu est le modèle flot de donnée synchrone SDF (*Synchronous Data Flow*) [27], connu dans la littérature sous le nom de graphes d'événements pondérés. Dans le modèle SDF, les données consommées et produites par une tâche sont connus statiquement, à la compilation. Le modèle SDF spécialise le modèle KPN et représente une application par un graphe orienté, illustré dans la figure 2.5, ayant des tâches liées par des arcs (de type mémoire FIFO). Cette mémoire nommée “*a*” dans la figure contient des données désignées par des jetons et a une quantité initiale de jetons, nommée marquage initiale $M_0(a)$. A chaque exécution de la tâche t , in_a désigne le nombre de jetons écrits dans la mémoire a et out_a désigne le nombre de jetons lus dans a pour démarrer une exécution de la tâche t' . Ce modèle SDF est souvent utilisé dans la conception de systèmes embarqués et systèmes temps-réel vu sa garantie d'un certain nombre de propriétés intéressantes. En particulier, SDF offre le déterminisme de l'exécution et permet ainsi d'éviter les *deadlock*, d'effectuer un ordonnancement statique (avant l'exécution) et de borner la consommation des ressources mémoire. SDF permet encore de détecter facilement le parallélisme propre à l'application et d'extraire les dépendances entre tâches

D'autres nombreuses variantes des modèles graphes flots de données ont été proposés pour la conception des systèmes mixtes matériel/logiciel dont nous énumérons des modèles significatifs dans la figure 2.6

4. First In First Out

FIGURE 2.6: Une chronologie partielle des principaux modèles de *Process Network*

Il existe d'ores et déjà plusieurs outils académiques permettant de compiler des applications *dataflows* à destination d'une architecture multiprocesseur. Parmi plusieurs outils, nous citons *ForSyDe*, *SDF3*, *StreamIt*, *Ptolemy (I et II)* (section 2.3.4).

2.3.2 Modélisation d'architecture

L'architecture générale d'un système sur puce est dite multi-composants car sa structure comprend en général des composants programmables (processeurs RISC, CISC, DSP, microcontrôleurs) et des composants non programmables (circuits intégrés spécifiques ASIC, circuits FPGA reconfigurables), d'un ensemble de périphériques et de composants de mémorisation, le tout interconnecté par un support de communication comme des bus simples ou complexes et parfois des réseaux sur puce NoC (*Network On-Chip*) [38].

Les concepteurs des SoCs ont cherché à améliorer les performances de tels systèmes en augmentant le nombre de processeurs intégrés dans une même puce. Néanmoins, la conception d'une architecture multi-coeurs doit prendre en compte nombreux paramètres liés par exemple aux types de calculs effectués et aux coûts de fabrication. Prenons l'exemple des téléphones mobiles intelligents, les performances des architectures sont liées fortement aux types d'applications supportées, aux fonctionnalités du téléphone ou encore à son autonomie. Une longue utilisation d'un tel système peut provoquer une forte consommation énergétique et par suite une importante génération de chaleur qui peut endommager les composants internes du téléphone et gêner l'utilisateur. Ainsi, la consommation énergétique et thermique présentent également un enjeu crucial des téléphones modernes. Le concepteur est ainsi face aux nombreux choix architecturaux concernant : les processeurs, les modèles d'exécution, les mémoires, le réseau d'interconnexion et le type d'architecture.

Face à cette complexification de design des circuits, il est devenu impossible de les concevoir dans un bas niveau RTL (*Register Transfert Level*) où il faut préciser chaque détail du comportement des composants. Pour y remédier, l'élévation du niveau d'abstraction est une solution optimale afin d'améliorer cette tâche. Ce niveau est dénommé conception de design au niveau système ESL (*Electronic System Level*). A ce niveau, il est permis de cacher les détails de bas-niveau et accélérer la phase de conception en utilisant un mécanisme automatisant le passage d'une modélisation à haut-niveau des systèmes au code permettant l'implémentation physique. Le concept central apporté à travers la conception ESL est celui des plate-formes matérielles virtuelles permettant l'assemblage des composants préexistants propriétaires (dits *IP Blocks* pour "*Intellectual Property*") pour la composition rapide d'un système complet. Dans ce contexte qu'ont été définis et raffinés de nombreux langages de description d'architecture tels que SystemC dans son niveau TLM et le standard IP-Xact, que nous présentons ci dessous. D'autre part, le profil UML-MARTE [15] de OMG propose également une modélisation haut niveau dirigée par les modèles (nous présentons le standard MARTE plus tard dans la section 2.4.2.2).

2.3.2.1 Langages de description des architectures

SystemC [39] est un langage basé sur un ensemble de classes C++, permettant de modéliser, analyser et simuler non seulement des systèmes matériels (numériques), mais aussi des systèmes mixtes matériels/logiciels, à différents niveaux d'abstraction. SystemC supporte également la modélisation hiérarchique du comportement du système. Ainsi, il isole respectivement la fonctionnalité et la structure dans des processus et des modules. Pour monter en abstraction, SystemC a proposé le standard TLM (*Transaction Level Modeling*). A ce niveau, les communications sont totalement abstraites. Le langage de description d'architectures SystemC est développé par l'*OSCI (Open SystemC Initiative)* et normalisé par l'IEEE en 2005 puis en 2011 (IEEE Std. 1666). Plusieurs plateformes de prototypage virtuels basés sur SystemC ont été proposées, comme SoCLib [40] et OVP [41].

IP-Xact [42] est une proposition de standardisation pour un ADL (*Architecture Description Language*) permettant la description de la conception des systèmes électroniques et les interconnexions des interfaces IPs (*Intellectual Property blocks*, souvent d'origine commerciale et fournis comme boîtes noires) dans une spécification standard et une représentation commune pour les fournisseurs d'IPs, aux intégrateurs de conception et vendeurs de EDA (*Electronic Design Automation*). Les Blocs IPs sont généralement connectés au moyen de supports de communication clairement définies. On peut citer les bus AMBA de ARM, les bus OCP-IP, en plus des nouveaux réseaux sur puce NoCs.

La modélisation des caractéristiques de l'architecture d'un système embarqué est une étape fondamentale dans le flot de co-design. En effet, un des problèmes majeurs depuis toujours consiste à trouver les meilleurs choix de configurations des ressources physiques. Cela doit être fait tout en respectant des exigences imposées sur le fonctionnement global du système. La consommation d'énergie, l'évolution de la température, la performance et l'allocation de fonctionnalités sur des ressources physiques sont des exemples d'analyse des systèmes embarqués hétérogènes. Les paragraphes suivants sont dédiés pour présenter certaines caractéristiques notamment la fréquence de fonctionnement, la puissance, la température ; ainsi que les techniques d'analyses et d'optimisations de ces caractéristiques.

2.3.2.2 Caractéristiques des architectures

Fréquence de fonctionnement appelée également cycle, correspond au nombre d'impulsions par seconde, s'exprimant en Hertz (Hz). La fréquence de fonctionnement et la tension d'alimentation jouent un rôle important dans la consommation du circuit (équation 2.2). La gestion de ce couple (tension v , fréquence f) s'effectue via des points de fonctionnements OPP (*Operating Performance Point*) a priori fixés par avance par les fabricants de circuits. Chaque OPP fixe un couple (v , f) qui peut être sélectionné dynamiquement (pendant le fonctionnement) en fonction du niveau de performance souhaité. En fonction de la charge du système et selon les stratégies d'énergie, l'OPP est ajusté pour que le système conserve les performances optimales du système, notamment en respectant éventuellement des contraintes de temps. Cette méthode d'ajustement des OPP, dénommée DVFS [43] (*Dynamic Voltage and Frequency Scaling*), est couramment utilisée

pour les processeurs. Nous présentons cette technique plus en détails dans le paragraphe suivant.

Puissance La consommation totale de puissance d'un dispositif électronique au niveau transistor est la somme de deux termes : la "consommation statique" due principalement à des courants parasites, et la "consommation dynamique" conséquence de l'activité de commutation (ou *Switching Activity*) des circuits. Elle peut être ainsi modélisée par l'équation 2.1 :

$$P_{totale} = P_{statique} + P_{dynamique} \quad (2.1)$$

La puissance dynamique ou encore appelé puissance de commutation, est dissipée lorsque les entrées de données ou d'horloge changent de niveau logique. Elle provient du fait qu'on charge et qu'on décharge la capacitance C_L de sortie d'un circuit au niveau logique, comme présenté dans la figure 2.7 (a) (technologie CMOS⁵). Dans les circuits CMOS, avec des finesses de gravures supérieures à $0.13 \mu\text{m}$, la puissance dynamique représente 80-85% de la puissance totale dissipée [44].



FIGURE 2.7: Puissance Consommée

La puissance dynamique est proportionnelle à la fréquence de fonctionnement, à la capacité de la charge et au carré de la tension d'alimentation. Elle peut être modélisée par la relation 2.2 où V_{dd} est la tension d'alimentation, f la fréquence d'horloge, C la capacité physique du circuit :

5. CMOS : Complementary Metal-Oxide-Semiconductor : une technologie de fabrication de composants électroniques

$$P_{dynamique} = C * V_{dd}^2 * f \quad (2.2)$$

La puissance statique est la puissance consommée lorsque le circuit est en régime permanent. Bien qu'il soit *OFF* (figure 2.7 (b)), il existe quand même des courants parasites dont le plus important est le courant de fuite résultant du courant "sous-seuil".

Dans un circuit idéal, cette consommation statique est nulle ; néanmoins, en réalité, un courant de fuite existe (de l'ordre du ρA habituellement) et cause une consommation énergétique non négligeable. Cette puissance peut être modélisée par l'équation 2.3 :

$$P_{statique} = V_{dd} * I_{leak} \quad (2.3)$$

Au niveau de l'architecture matérielle les techniques classiquement employées dans les systèmes embarqués pour l'optimisation de la consommation de la puissance dissipée visent soit la consommation de puissance statique, soit celle relative à la puissance dynamique. Nous citons trois techniques essentielles : le *DPM*, le *DVFS* et le *Clock Gating*.

Le *DPM* (*Dynamic Power Management*) (ou aussi *Power Gating* ou *Dynamic Power Switching*) : L'idée générale de cette technique est de faire rentrer le processeur ou une partie du circuit qui n'est pas utilisée dans le mode en veille , précisément, l'éteindre complètement en coupant la tension. Ainsi aucun courant de fuite ne circule ; ce qui permettrait d'éviter la perte causée par la puissance statique.

Le Pilotage Dynamique de la fréquence et de la tension *DVFS* (*Dynamic Voltage and Frequency Scaling*)[43] : vise à adapter la puissance dynamique et par conséquent les paramètres de l'équation (f , C et V_{dd}), afin de changer dynamiquement la quantité de travail d'un processeur durant son activité. Cela se fait en général en réglant les valeurs de fréquences/tensions des processeurs selon la grandeur de la quantité de travail à accomplir.

Le *Clock Gating* : vise à ne pas envoyer le signal d'horloge en entrée des différents blocs inutilisés (en état dormant). Ainsi, en appliquant le *Clock Gating*, la puissance dynamique consommée de ces blocs est supprimée. Les pénalités de passage de l'état

horloge inactive à horloge active (et inversement) sont très faibles. La difficulté réside plutôt dans la conception de l'arbre d'horloge qui doit assurer une synchronisation correcte des horloges sur les blocs du système. Le Clock Gating est fortement utilisé en pratique.

Température La gestion de la température a été largement étudiée dans un contexte de calcul hautes performances opérant à des fréquences élevées afin de réduire les points chauds (*hot spots*) qui peuvent apparaître si un processeur ou un cluster de processeurs est momentanément très chargé avec une fréquence de fonctionnement élevée. En effet, l'exécution des modules voisins [45], l'ordre d'exécution des tâches [46] et l'ordonnancement des tâches et des ressources impactent l'énergie consommée et par suite la température.

La plupart des techniques d'optimisation des propriétés thermiques dans les systèmes embarqués ont aussi un impact sur les autres propriétés de la conception tels que la surface de la puce et la consommation d'énergie. Les unités fonctionnelles FU (*Functional Units*) invoquant une haute connectivité peuvent être placées à côté pour minimiser la consommation d'énergie. Cependant, ceci peut augmenter le pic de la température. Ainsi, il y a une forte interdépendance entre l'énergie et la température et par suite les techniques d'optimisation de l'énergie doivent être employées afin de ne pas augmenter les pics de température.

Des techniques de gestion thermique ont été explorées à la fois au moment de la conception à travers des boîtiers (*packaging*) appropriés et des mécanismes de dissipation de la chaleur, et au moment de l'exécution des tâches à travers diverses formes de gestion dynamique de température (*DTM : Dynamic Thermal Management*). En effet, des études publiées cherchent principalement à équilibrer la charge (*load balancing*) entre processeurs dans les architectures multiprocesseurs embarquées. Par exemple, les travaux décrits dans [47] visent à alterner entre les deux processeurs de l'architecture dans le but de réduire le coût de refroidissement du système, et ce en prenant comme hypothèse que les tâches s'exécutent en un temps généralement plus petit que leur pire cas. Cependant ce travail ne considère aucune contrainte liée à l'ordonnancement effectif des tâches et à leurs échéances d'exécution. D'autres travaux exposés dans [48] proposent une technique d'équilibrage de charge entre quatre processeurs en exploitant les différents états de repos

des processeurs. Une autre technique similaire détaillée dans [49] pointe à ordonnancer les tâches en favorisant le critère thermique et ceci en utilisant les possibilités de DPM, DVFS et migration par rapport aux contraintes de temps réel.

2.3.3 Méthodes d'optimisation de l'allocation

L'application est définie par un ensemble de tâches, où chaque tâche est décrite par des paramètres (dépendance, période d'exécution, communication, mémoire, etc.). Cependant, certaines tâches sont spécifiées pour des GPP et/ou DSP, d'autres sont spécifiées pour les ressources d'exécution matérielles tel que blocs IP ou zones reconfigurables. Ainsi pour gérer l'ensemble, un ordonnancement et placement des tâches sur les ressources, une allocation mémoire et une gestion des ressources partagées, etc, sont requis.

Le placement et ordonnancement (*Mapping and Scheduling*) appelé aussi AAS (*Assignment Affection and Scheduling*) sont des étapes qui suivent l'étape de spécialisation du système embarqué. Leur rôle est d'implémenter l'application sur l'architecture sélectionnée. Ceci consiste principalement à assigner et ordonnancer les tâches et communications de l'application sur les ressources d'architecture cible de telle façon que les objectifs et contraintes spécifiés soient atteints. Le placement et ordonnancement sont parmi les phases les plus difficiles et critiques lors de la conception, de bons algorithmes de placement et ordonnancement sont cruciaux pour avoir un maximum de performance pour une application sur une architecture donnée.

D'autre part, la modélisation du temps dans certains systèmes embarqués sur puce, tels que ceux qui fonctionnent en temps réel, est très importante. La notion de temps peut être modélisée de différentes façons selon le type de comportement souhaité : Le comportement est dit *synchrone* si les entrées et les sorties sont produites au même instant ; il est dit *causal* si le temps est pris en compte selon les relations de cause à effet ; il est dit *discret* si un déclenchement ou une action spécifique décrit l'unité de temps et enfin il est dit *continu* si le temps est représenté par des valeurs réelles.

Dans la problématique d'analyse temporelle de haut niveau des systèmes embarqués, il est important de prendre en considération certaines contraintes temporelles telles

que l'ordonnancement des différentes tâches ainsi que des optimisations faites sur les caractéristiques des architectures cibles telles que le nombre et fréquence de processeurs et la taille des mémoires, etc. Cela conduit à des estimations de configurations idéales de systèmes. Pour raffiner ces estimations, des simulations de prototypages sont mises en place.

Les algorithmes d'ordonnancement de tâches sur des ressources peuvent être soit : *monoprocesseur/multiprocesseur* (selon le nombre de processeurs dans l'architecture), *préemptif/non-préemptif* (si l'ordonnancement peut interrompre l'exécution d'une tâche au profit d'une autre tâche plus prioritaire ou pas) et *hors-ligne/en-ligne* (si les décisions d'ordonnancement des tâches sont prises avant l'exécution ou en cours d'exécution). Dans la littérature, il existe plusieurs algorithmes d'ordonnancement tels que *Earliest Deadline First (EDF)* [50], *Rate Monotonic (RM)* [50] et *Deadline Monotonic (DM)*.

2.3.4 Systèmes de conception conjointe

De nombreux environnements d'exploration globale d'architecture tels que *MetroPolis* [51], *Milan* [52] facilitent l'analyse des performances au niveau système en permettant d'associer des spécifications comportementales d'application à des spécifications d'architecture. D'autres outils, tel que *StreamIt* et *ForSyDe*, au niveau système présentent des caractéristiques différentes qui font que chacun d'eux est plus spécifiquement adapté à une classe d'application (orientés flots de données, orientés flots de contrôle). De nombreux éditeurs proposent aussi des outils de conception conjointe tel que *Platform Architect* de *Synopsys*, *SynDEx*, *SDF3* et *Ptolemy*.

ForSyDe⁶ (*Formal Design System*) [53] est une méthodologie de conception des systèmes embarqués hétérogènes. Son modèle de spécification est basé sur le modèle de calcul synchrone et modélise le système avec un réseau hiérarchique de processus connectés par des signaux. ForSyDe offre plusieurs modèles de calcul définissant les modalités et contraintes de communication entre processus. On dénombre ainsi les flots de données

6. <https://forsyde.ict.kth.se/trac>

synchronisées SDF [27] et les machines à état fini FSM [21]. Il est possible de simuler ou de générer du code VHDL à partir des designs.

StreamIt⁷ [54] est une chaîne de compilation, basé sur son propre langage, nommé StreamIt. Cette chaîne est développée par une équipe du groupe CSAIL au Massachusetts Institute of Technology. StreamIt est conçu pour faciliter la programmation des applications de streaming, ainsi leurs compilations sur une grande variété d'architectures cibles, y compris les processeurs à architectures parallèles comme le RAW [55] ou le Tile64 [56]. Compte tenu de la sémantique de son langage, la principale limitation de StreamIt est son expressivité; par exemple une application StreamIt ne peut disposer que d'une unique d'entrée et d'une unique sortie.

SDF3⁸ [57] est un outil de génération aléatoire d'application flots de données, proposé au sein de l'université technologique d'Eindhoven aux Pays-Bas par le groupe *Electronic Systems*. SDF3 possède une vaste librairie des analyses de graphes SDF et des transformations d'algorithmes, ainsi des fonctionnalités pour les visualiser. SDF3 est fortement lié à [58], le réseau sur puce proposé par *Philips*.

Ptolemy (I et II)⁹ [7] [59] est l'un des premiers et plus matures outils "open source" pour la modélisation et la simulation des applications modélisées avec des modèles de calculs. Cet outil de recherche a été développé à l'université de Californie, à Berkeley, sous la direction du Professeur *Edward A. Lee*. Ptolemy vise la conception des systèmes embarqués, spécifiquement les systèmes qui combinent plusieurs technologies différentes. La spécificité de Ptolemy est la l'imbrication hiérarchique des modèles hétérogènes. Ptolemy propose un certain nombre de modèles dont les plus importants sont le modèle à événements discrets DE, le modèle à base de machines d'états finis FSM, le modèle à base de processus communicants CSP, le modèle à base de réseaux de processus synchrone SDF, etc.

7. <http://groups.csail.mit.edu/cag/streamit/>

8. <http://www.es.ele.tue.nl/sdf3/>

9. <http://ptolemy.eecs.berkeley.edu/>

SynDEx *SynDEx*¹⁰ (*Synchronized Distributed Executive*) [60] [61] est un environnement graphique, développé à l'INRIA de Rocquencourt, et qui concrétise la méthodologie AAA (Adéquation Algorithme Architecture) [62] pour le prototypage rapide et l'optimisation de l'implantation d'applications distribuées temps réel embarquées. *SynDEx* propose une interface graphique qui permet de spécifier le graphe d'algorithme, le graphe d'architecture ainsi que les caractéristiques de ces graphes. Cet outil est capable, ensuite, de construire et d'afficher automatiquement le diagramme temporel d'implantation optimisé de l'application grâce à l'heuristique qu'il renferme. Une fois les caractéristiques de ce graphe sont conformes avec les exigences temps réel de l'application, *SynDEx* génère automatiquement l'exécutif de chacun des processeurs de l'application ainsi que le "makefile" qui va automatiser les différentes phases de substitution, compilation, chargement et lancement de l'exécutif de l'application.

Platform Architect MCO Platform Architect MCO¹¹ (*Multi-Core Optimization MCO*) est un outil de prototypage virtuel développé par Synopsys. C'est un environnement graphique qui fournit aux concepteurs de systèmes, des outils de simulation d'analyse de performance et des méthodes SystemC TLM afin de garantir une conception efficace et une optimisation des systèmes multi-coeurs et des architectures de systèmes sur puce. Platform Architect MCO considère une architecture modélisée au niveau transactionnel (modèle abstrait en SystemC-TLM) dans le but d'obtenir des simulations rapides et de permettre le développement et la validation du logiciel applicatif qui devra être exécuté par le système réel. Ce modèle d'architecture au niveau transactionnel est appelé plateforme virtuelle, il permet de valider les aspects fonctionnels du système. En outre, Platform Architect MCO permet la configuration facile des graphes de tâches pour créer un modèle de performance SystemC de l'application et les allouer sur des unités de traitement virtuelles (*Virtual Processing Unit VPU*). Un VPU est un modèle abstrait d'un composant de l'architecture matérielle, il peut s'agir d'un processeur, d'un périphérique ou d'un composant matériel spécifique

10. <http://syndex.org/>

11. <http://www.synopsys.com/Prototyping/ArchitectureDesign/Pages/platform-architect.aspx>

Un graphe de tâche sous Platform Architect MCO, comme illustré dans la figure 2.8, est spécifié par :

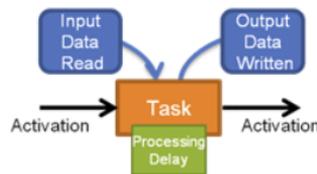


FIGURE 2.8: Graphe de tâches sous Platform Architect MCO

- Les tâches : également connus par les noeuds de graphe de tâches. Ils exécutent les opérations et transforment les flots de données en entrée en flot de données en sortie.
- Les dépendances : également connus par les arcs de graphe de tâches. Ils spécifient l'ordre dans lequel les tâches doivent être exécutées pour traiter un nombre spécifique de données.
- Taux d'activation : ce sont des paramètres des arcs et qui indiquent la fréquence des activations de la tâche.
- Délai d'exécution : c'est un paramètre de la tâche qui indique le temps qu'occupe une tâche pour être exécuté sur une ressource
- Accès à la mémoire par activation : c'est un paramètre de la tâche qui indique la quantité d'octets écrits et lus lorsque la tâche est activé.

Dans Platform Architect MCO, le graphe de tâches est présenté sous forme d'un fichier CSV (*Comma-separated values*), comme illustré dans la figure 2.9 constitué d'un ensemble de tableaux qui permettent de définir les propriétés pour chaque tâche en utilisant des paramètres bien spécifiques.

*Synopsys*¹², l'un des partenaires industriels du projet HOPE, nous a fourni *Platform Architect* pour le prototypage virtuel des modèles, particulièrement de notre cas d'étude que nous présentons ultérieurement. Dans des travaux précédentes, nous avons proposé une génération automatique de quelques éléments de notre modèle MUVARCH (application et architecture sans informations sur l'ordonnancement) vers Platform Architect

12. <http://www.synopsys.com/home.aspx>

| ## Task Table | | | | |
|---------------|--|--|--|--|
| task_name | | | | |
| T1 | | | | |
| T2 | | | | |
| T3 | | | | |
| T4 | | | | |
| T5 | | | | |

| ## Connection Table | | | | |
|---------------------|-------------|------------------------|--------------------|--------------------|
| task_name | destination | p_out_init_put_samples | p_put_rate_samples | p_get_rate_samples |
| T1 | T2 | | | 2 |
| T2 | T3 | | | 3 |
| T3 | T4 | | | |
| T2 | T5 | 2 | | 2 |
| T5 | T2 | | 4 | 2 |

| ## Function Table | | | |
|-------------------|---------------|----------------------|-------------------|
| task_name | function_name | function_type | processing_cycles |
| T1 | consume | scmi_fm_cpu_function | 1000 |
| T2 | consume | scmi_fm_cpu_function | 20000 |
| T3 | consume | scmi_fm_cpu_function | 10000 |
| T4 | consume | scmi_fm_cpu_function | 1000 |
| T5 | consume | scmi_fm_cpu_function | 5000 |

FIGURE 2.9: Graphe de tâches sous Platform Architect MCO

MCO. Nous aimerons dans le futur pouvoir connecter notre approche (enrichi d'un ordonnancement des tâches vers les ressources) à un tel outil pour permettre une analyse plus déterminée des propriétés non-fonctionnelles.

Notre approche dans ce mémoire sera basée premièrement sur l'ingénierie dirigée par les modèles (IDM) pour la modélisation de notre approche ; qui permettra après d'assurer la génération automatique du code à partir de notre modèle de haut niveau et améliorer ainsi la productivité des conceptions. Nous présentons dans la section suivante 2.4 cette pratique d'ingénierie.

2.4 Utilisation de l'IDM dans notre cadre

L'ingénierie dirigée par les modèles IDM (ou aussi *Model Driven Engineering MDE*) [63] est une pratique d'ingénierie des systèmes utilisant les capacités des technologies informatiques pour décrire au travers de modèles, concepts, et langages, à la fois le problème posé (besoin) et sa solution. Cette pratique est construite autour de deux notions fondamentales que sont les modèles et les transformations. L'objectif principal est que tout processus de production peut être vu comme un ensemble de modèles reliés par des transformations. En effet, un **modèle** est une représentation abstraite de la réalité. Il sert à représenter de façon schématique et simplifiée certains concepts d'un logiciel de façon à mieux en comprendre le fonctionnement et l'architecture. Pour comprendre ce que contient un modèle et les informations qu'il représente, il faut auparavant s'être entendu sur la définition de son contenu. C'est ce travail que l'on effectue lorsque l'on parle de

méta-modélisation. Un **méta-modèle** est donc le modèle d'un modèle, plus précisément un méta-modèle décrit les relations et les concepts pour construire un modèle.

L'architecture dirigée par les modèles (Model Driven Architecture MDA) [64] est une variante de l'ingénierie dirigée par les modèles IDM pour le génie logiciel. En effet, MDA est proposée et soutenue par l'*Object Management Group* (OMG), en novembre 2000, afin de faciliter le développement et la maintenance des systèmes essentiellement logiciels. MDA s'appuie sur un ensemble de standards de l'OMG et en particulier le langage de modélisation UML (*Unified Modeling Language*) [11] et le standard MOF (*MetaObject Facility*) [65].

2.4.1 MetaObject Facility MOF

L'OMG à travers du MOF [65] propose un langage standardisé afin de permettre la définition de nouveaux méta-modèles. MOF permet, à l'aide d'un vocabulaire restreint, de décrire tout type de méta-modèle. C'est pour cela que l'OMG le classe dans la catégorie des "méta-méta-modèles". De plus, ce langage définit les concepts de base tels que entité/classe, relation/association, type de données, référence, package. La spécification de MOF adopte une architecture de méta-modélisation à trois niveaux, schématisés dans la figure 2.10. Le *niveau M3* comporte le méta-méta-modèle permettant de décrire la structure des méta-modèles, d'étendre ou de modifier les méta-modèles existants. Dans, le *niveau M2*, on trouve le méta-modèle décrivant le langage de modélisation. Le *niveau M1* est composé du modèle représentant le système réel à modéliser.

2.4.2 Unified Modeling Language UML

UML [11] est un langage orientée objet de modélisation. Ce langage a pour but de modéliser un large éventail de systèmes relevant de différents domaines d'application. Il comprend un ensemble de diagrammes où chaque diagramme s'intéresse à un point de vue du système, et permet ainsi de diviser le modèle complexe d'un système en plusieurs sous-modèles complémentaires moins complexes et plus facilement abordables et

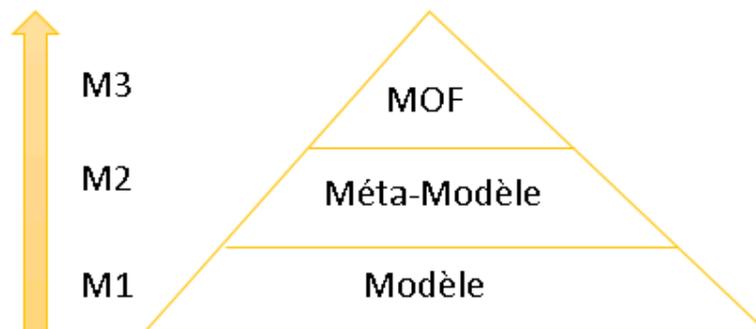


FIGURE 2.10: Structure des différents niveaux de modélisation de l'approche MDA

compréhensibles. Par exemple, le diagramme de classe vise l'aspect structurel du système, tandis que le diagramme d'activité vise l'aspect comportemental de ce système. Les méta-modèles d'UML ne sont pas modifiables. En contrepartie, UML offre des mécanismes d'extension à travers le mécanisme de *profil*. Un profil UML réalise le rapprochement et la spécialisation d'UML à un domaine spécifique. SYSML et MARTE sont des exemples de profils UML, que nous détaillons dans 2.4.2.1 et 2.4.2.2 respectivement. Face au succès du mécanisme d'extension, l'OMG a adopté une approche permettant de définir et utiliser des langages spécifiques aux métiers, les *Domain Specific Languages* (DSLs). Nous exposons ces langages ensuite dans 2.4.3.

2.4.2.1 SysML

SYSML (*Systems Modeling Language*) est un profil UML, et par suite une spécialisation d'UML, dédiée aux systèmes complexes, en particulier les systèmes dédiés à l'automobile, l'avionique, l'automatique et la communication. Comparé à UML, SYSML ajoute la possibilité de représenter des domaines spécifiques tels que les exigences du système (*Requirement diagrams*), les éléments non-logiciels (tel que la mécanique et l'hydraulique, etc.), les équations physiques (*Parametrics Diagrams*), les flux continus (tels que la matière, l'énergie, etc.) et les allocations. L'allocation ici est un mécanisme général pour interconnecter des éléments de différents types. Par exemple, pour relier un flot d'objet dans un diagramme d'activité à un connecteur dans un diagramme de bloc interne

(ibid dans SYSML), etc. Dans ce mémoire, nous nous sommes fortement inspirés de diagrammes paramétriques proposés par SYSML pour modéliser les aspects continus de notre approche.

2.4.2.2 MARTE

Le profil UML pour la modélisation et l'analyse des systèmes temps réel et embarqués MARTE (*UML profile for Modeling and Analysis of Real-Time and Embedded systems* en anglais) a pour but de spécialiser UML afin de l'utiliser dans une approche de développement dirigé par les modèles de systèmes temps réel et embarqués. MARTE fournit des supports pour les étapes de spécification, de conception et de vérification/validation d'un système. Ce profil offre aussi une modélisation unifiée pour les parties matérielles et logicielles du système. En plus, il facilite la construction de modèles sur lesquels on peut faire des prévisions quantitatives en prenant en compte des caractéristiques du matériel et de logiciel.

2.4.3 Langage dédié DSL

Les langages domaine-spécifiques DSLs (*Domain Specific Language*), appelés aussi langages dédiés, sont des langages conçus spécifiquement pour un domaine particulier offrant aux utilisateurs des concepts propres à leur domaine et dont ils ont la maîtrise. Parmi les DSLs, nous distinguons les langages dédiés de modélisation DSMLs (*Domain Specific Modeling Language*). Les DSMLs ont été développés et utilisés dans plusieurs domaines tels que l'automobile, l'avionique et les systèmes cyber-physiques. Des environnements tels que Microsoft DSL Tools¹³, Eclipse Modeling Framework EMF¹⁴ et MetaEdit+¹⁵ supportent la spécification de la syntaxe abstraite, de la syntaxe concrète, et des sémantiques statique et dynamique d'un DSML. Dans la suite de ce mémoire, nous nous intéressons au framework EMF pour la modélisation de l'approche MUVARCH, ainsi qu'à l'outil Sirius pour l'édition graphique des modèles.

13. <http://www.microsoft.com/en-us/download/details.aspx?id=2379>

14. <http://www.eclipse.org/modeling/em>

15. <http://www.metacase.com/me>

EMF (*Eclipse Modeling Framework*) [66] est un outil développé par la fondation Eclipse et permettant la modélisation, la méta-modélisation ainsi que la génération de code. Pour décrire ses modèles, EMF s'appuie sur Ecore [67]; un méta-méta-modèle basé sur le niveau 3 de MOF (voir figure 2.10). Ecore permet ainsi de définir n'importe quel type de méta-modèle et de nouveaux vocabulaires DSL (*Domain Specific Language*) (section 2.4.3); autorisant de travailler avec des modèles dans des contextes différents.

L'objectif principal d'EMF est la génération automatique du code permettant d'obtenir facilement une implémentation en Java. Également, EMF permet une migration facile entre ce code Java, UML et Ecore grâce à la possibilité de sérialisation de ces fichiers sous format XMI. Cette fonctionnalité est très intéressante dans notre contexte puisque le contenu du méta-modèle du contrôleur général doit être décrit lors du calcul de l'ordonnancement auprès d'un solveur SMT. Ce calcul sera injecté après à notre méta-modèle Ecore via une implémentation Java.

Un autre point fort du framework EMF est qu'il est facile à utiliser pour la création et l'édition des modèles. En plus, il permet la validation des modèles créés, ce qui est indispensable pour garder la conformité avec leurs méta-modèles. Il fournit également la possibilité d'enrichir les méta-modèles créés en ajoutant des contraintes OCL (*Object Constraint Language*) [68]. Ce que nous avons besoin après pour vérifier les règles de cohérence des relation d'Allocation.

Sirius [69] est un outil développé par la fondation Eclipse dans le cadre du projet *Modeling*. Sirius permet de créer très facilement un atelier de modélisation personnalisé et entièrement dédié au domaine d'expertise du concepteur et ceci en se basant sur la représentation d'un DSL au sein d'un modèle EMF. La grande force de Sirius est d'offrir la possibilité de créer des ateliers de manière totalement graphique, sans avoir à écrire de code, ce qui lui rend de plus en plus répandu par rapport au GMF (*Graphical Modeling Framework*) qui est extrêmement complexe à prendre en main et peu intuitif.

C'est, par exemple, grâce à la technologie mise en œuvre dans *Sirius* que *Thales* a défini une méthodologie adaptée à ses enjeux d'ingénierie (ARCADIA [13]), et un environnement de modélisation associé offrant le support à l'implémentation de vues d'ingénieries

de spécialité. Les experts de *Thales* utilisent cet environnement pour le design et l'analyse de systèmes complexes nécessitant des correspondances entre diverses préoccupations issus de domaines d'expertises différents. Chaque expert se base sur son propre langage et représente le système à partir de son propre point de vue. L'activité de modélisation où les préoccupations sont divisés sous plusieurs domaine est appelé **modélisation multi-vues**. Dans la section suivante 2.4.4, nous détaillons la modélisation multi-vues et nous discutons quelques travaux existants basées sur ce type de modélisation.

2.4.4 Cas spécial de la modélisation multi-vues

La modélisation multi-vues est proposée comme une solution pour gérer la complexité des systèmes. Cette modélisation définit le système sous différents vues où chaque vue s'intéresse des préoccupations des experts d'un domaine spécifique. Le standard ISO-42010 [6] est une référence de cette approche de modélisation. Les **vues** sont définies par les experts domaines ayant leurs propres concepts et langages pour représenter les éléments de leurs domaines particuliers. Pour maintenir la consistance entre les vues du système, des **règles de correspondance** sont mises en place en associant les différents éléments du système étudié. Selon [6], la modélisation multi-vues peut être approchée de deux façons : l'approche *projective* et l'approche *synthétique*.

L'approche projective considère les vues comme des **projections** d'un **modèle de référence**. Dans cette approche, le modèle de référence regroupe toutes les informations sur le système étudié. Les vues des *réflexions* ou encore des *requêtes* du modèle référence ; ils effectuent ainsi des projections cachant les informations non pertinentes lors de l'étude d'un aspect particulier du système.

D'autre part, l'approche synthétique utilise les vues étant des **définitions partielles** ou des observations attendues du système. Ainsi, il n'y a pas de modèle de référence dans le système. Toutefois, les informations sur le système peuvent être obtenues dynamiquement en combinant les informations observés dans les différentes vues.

Dans la partie qui suit, nous exposons quelques approches basées sur la modélisation multi-vues. Nous citons deux approches projectives : VUML et SYSML et une approche synthétique : la modélisation multi-vues avec *ModHel'X*.

VUML L'approche VUML (*View based Unified Modeling Language*) [70] est une spécialisation du standard UML qui a été développée afin que UML supporte la notion de vue et ceci en permettant d'analyser et concevoir un système logiciel par une approche combinant objets et points de vue. Le principe de cette d'approche consiste à définir un diagramme de classe comportant les classes classiques de UML ainsi qu'un nouveau type de classes multi-vues définies par de par deux stéréotypes : *base* et *view*. Le stéréotype *base* représente la référence commune accessible par tous les points de vue du système. Tandis que le stéréotype *view* représente la partie spécifique du base associée à un point de vue particulier du système. Pour relier ces deux stéréotypes, deux relations de dépendance stéréotypées *ViewDependency* et *ViewExtension* sont définies dans VUML. Ils s'agit donc de deux relations de correspondances entre les points de vues.

SysML Comme présenté précédemment dans la section 2.4.2.1, SYSML est une spécification d'OMG qui spécifie le profil UML pour le domaine d'ingénierie système. Quelques éléments de ce standard représente les concepts basique du standard IEEE-42010 [6] pour définir une approche multi-vues. Une vue (*view*) [71] dans SYSML est une sorte de *package* utilisée pour montrer une perspective particulière sur un modèle, comme la sécurité, ou les performances. Un point de vue (*viewpoint*) représente une aspect particulier qui spécifie le contenu d'une vue. Une vue est ainsi *conforme* à un point de vue. Cette relation de conformité est représentée par une relation UML de dépendance.

Modélisation multi-vues avec *ModHel'X* *Boulangier et al.* présentent dans [72] une approche synthétique pour la modélisation multi-vue. En effet, cette approche considère les vues comme des couches superposées qui ajoutent les informations les unes aux autres pour construire un modèle globale du système. Cette approche définit des vues indépendants d'un modèle du système avec des blocs *ModHel'X*[73], où chaque bloc présente un comportement visible du système. Dans le contexte de la modélisation multi-vue, un bloc

spécifie le comportement du système à partir d'un point de vue particulier. Par exemple, un système peut avoir un comportement fonctionnel, un comportement thermique ou encore un comportement énergétique. Dans ce travail, les correspondances sont représentées à travers des relations comportementales à travers les vues. Cette approche basée sur *ModHel'X* propose le recours d'un unique langage, défini en *ModHel'X*, pour exprimer la représentation multi-vues du système à travers les points de vues et les correspondances.

Dans les travaux de ce mémoire, nous proposons une approche multi-vues pour la conception du système qui réunit les avantages des approches projective et synthétique. En effet, nous définissons des correspondances explicites dans un modèle de référence, ainsi que des vues indépendantes spécifiées pour chaque domaine d'expert. Nous détaillons notre approche et ses éléments dans le chapitre suivant.

2.5 Conclusion

Dans ce chapitre nous nous sommes attachés à mettre en évidence le rôle central des modèles dans le processus de conception des systèmes tel qu'il est vu dans le contexte de l'ingénierie des modèles IDM. Cette approche que nous adoptons pour nos travaux s'inscrivant dans le cadre de cette thèse étant donné sa capacité à accélérer les développements et de faciliter leur réutilisation. Nous avons présenté également différentes méthodologies de conception, notamment pour la conception de systèmes logiciels, visant à apparier un modèle abstrait d'application avec un modèle abstrait d'architecture sous des contraintes fonctionnelles et non fonctionnelle. Nous avons évoqué certains formalismes et outils participant à ces méthodologies de conception. Dans le cadre de l'approche MDA de l'OMG, nous avons cité quelques standards dans le domaine notamment l'IDM, UML et ses profils. Plus particulièrement, nous avons présenté le cas spécial de la modélisation multi-vues et ses différentes approches.

Dans ce qui suit, nous présenterons nos contributions quant à la modélisation multi-vues à haut niveau. La définition de notre approche nommée *MUVARCH* fera donc objet du chapitre suivant.

MU_VARCH :

UNE APPROCHE MULTI-VUES POUR
LA MODÉLISATION DES SYSTÈMES
EMBARQUÉS HÉTÉROGÈNES

Sommaire

| | | |
|------------|---|-----------|
| 3.1 | Introduction | 43 |
| 3.2 | Principes de notre modélisation | 44 |
| 3.3 | Porte vues : Vue architecturale de base | 49 |
| 3.4 | Autres vues spécialisées | 51 |
| 3.4.1 | Aspects communs des autres vues | 51 |
| 3.4.2 | Spécialisation des vues non-fonctionnelles | 54 |
| 3.4.2.1 | Vue performance | 54 |
| 3.4.2.2 | Vue puissance | 55 |
| 3.4.2.3 | Vue thermique | 56 |
| 3.4.3 | Spécialisation de la vue applicative | 57 |
| 3.5 | Relations entre vues : Abstraction et Association | 58 |
| 3.5.1 | Association | 58 |
| 3.5.2 | Abstraction | 60 |
| 3.6 | Contrôleur général de MuVArch et Relation d'Allocation | 61 |

| | | |
|------------|--|-----------|
| 3.6.1 | Contraintes issues des relations de MU _V ARCH | 62 |
| 3.6.2 | Relation d'Allocation effective | 63 |
| 3.6.3 | Représentation alternative d'allocation comme FSM | 66 |
| 3.7 | Conclusion | 68 |

3.1 Introduction

Le présent chapitre présente la définition de notre langage nommé MU_VARCH (pour Modélisation mUlti Vues de l'ARCHitecture). MU_VARCH est un DSML (*Domain Specific Modeling Language*) dédié pour la modélisation haut niveau des propriétés fonctionnelle et extra-fonctionnelle des systèmes-sur-puce embarqués, prenant en compte les aspects de : (a) la performance, (b) la consommation énergétique et (c) la température, ainsi que (d) des scénarios applicatifs pour exercer la plate-forme d'exécution. Ces aspects sont spécialisés dans des “vues”, définies par les experts de chaque domaine. Plus prosaïquement, l'organisation des éléments dans la vue sera censée refléter une intention pour le design physique ultérieur. Les vues sont spécifiées indépendamment, cependant, leurs relations et associations sont très importantes pour comprendre la consistance et la cohérence globale du système. Dans la modélisation multi-vues, ces relations seront donc nommées “*correspondances*”. La figure 3.1 nous servira tout au long de ce chapitre pour décrire l'organisation générale de MU_VARCH.

Notons que parmi les vues, il existe une vue particulière qui présente le support de toutes les autres vues. Dans ce qui suit, nous appelons indifféremment cette vue de base “référence”, “backbone”, “porte vues” ou encore plus génériquement “architecture”. Signalons d'ores et déjà ici notre volonté de faire apparaître, pour des raisons de symétries et similitudes, la vue fonctionnelle “application” (notée (d) ci-dessus) comme une vue particulière portée sur l'architecture au même titre que les autres non-fonctionnelles.

Nous commençons le chapitre (section 3.2) par présenter les principes de notre modélisation en les commentant sur la figure 3.1. Nous présentons après en détails les aspects de méta-modélisation des différents éléments de MU_VARCH. Dans la section 3.3, nous présentons la vue de base (“backbone”) supportant et permettant la définition des autres

vues du système (a), (b), (c) et (d). Les aspects communs et spécialisés de ces derniers sont détaillés dans la section 3.4. Ensuite, nous représentons les relations inter-vues assurant la cohérence du système dans la section 3.5, en s’attardant sur la relation spéciale d’allocation dans la section 3.6.

Notons que ce chapitre contiendra essentiellement les méta-modèles de nos vues, sans exemples notables pour les illustrer. L’objet du chapitre 4 suivant sera justement de démontrer l’utilisation de notre formalisme sur un cas d’étude d’architecture *big.LITTLE* (de type processeur multi-cœurs ARM).

3.2 Principes de notre modélisation

Avant de décrire les détails de nos modèles et de leur caractérisation dans un environnement de méta-modélisation, nous illustrons les grandes lignes de notre approche à l’aide du support de la figure 3.1.

Au centre de la modélisation se trouve le modèle d’architecture (la vue “*support*”, ou “*backbone*”). Il se compose d’un bloc-diagramme traditionnel, avec des composants de type calcul (*processors*), communication (*interconnect*), stockage (*memory*) ou périphériques (*I/O devices*). Le niveau de description peut être très abstrait ou plus détaillé sans influencer sur le mode de représentation. Les autres vues ; performance, power, thermique et application référencés par (a), (b), (c) et (d) respectivement dans la figure 3.1 ; comprendront de nombreux aspects communs ou similaires, que nous chercherons à factoriser pour en faire mieux apparaître les grandes lignes.

Aspect Structurel des vues : Tout d’abord, les trois vues extra-fonctionnelles (Performance ou Clock, Power, Température) reposeront structurellement sur la notion de *Domaine*. Un domaine est un regroupement de composants au niveau de la vue architecturale de base, et les domaines forment une partition de cette vue support. On pourrait aussi considérer des domaines hiérarchiques, avec une inclusion de sous-domaine. Nous aurons donc des *Clock Domains CD*, des *Voltage/Power Domains PD* et des *Thermal Domains TD* respectivement pour ces vues. A chaque domaine est associée (au moins)

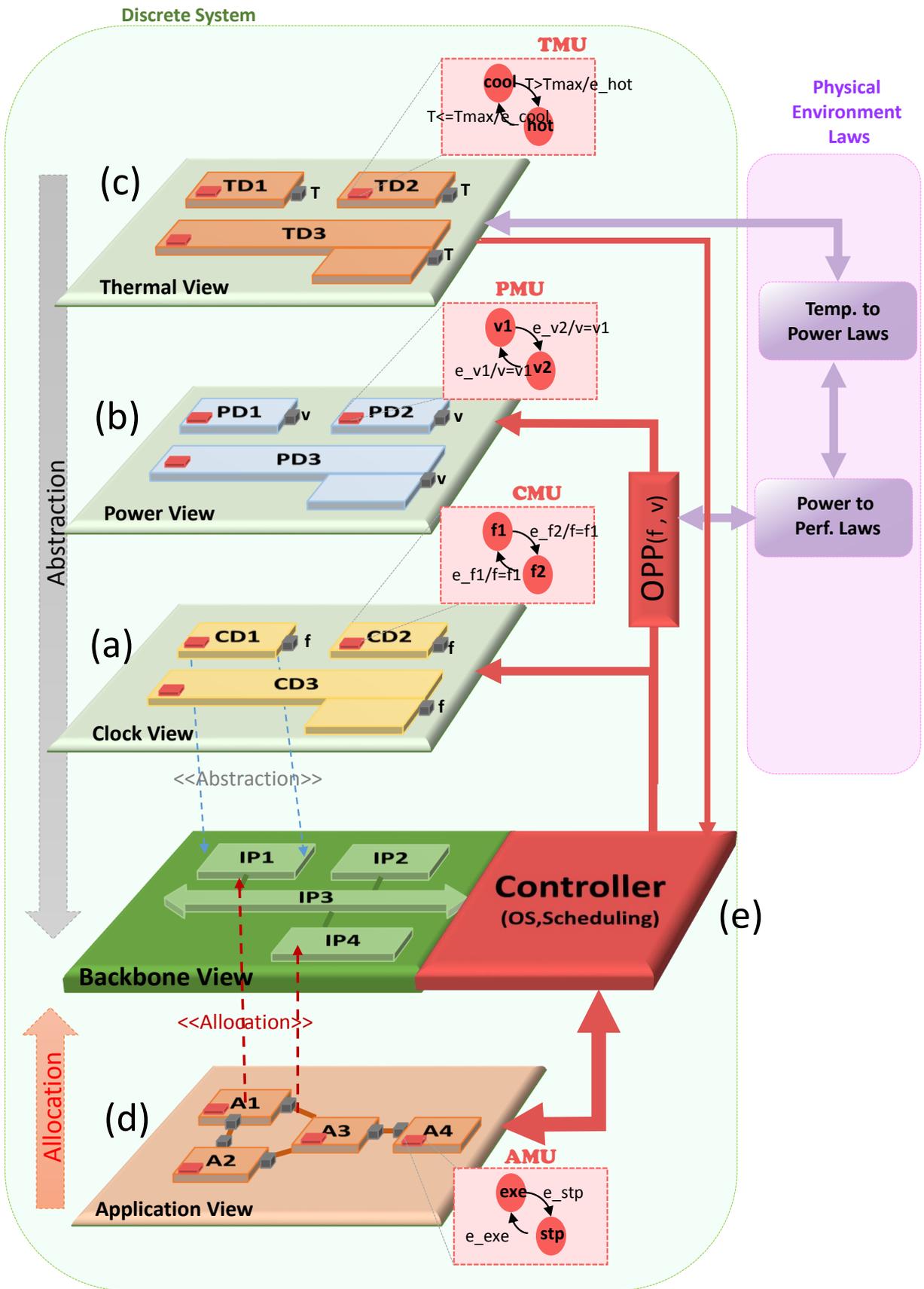


FIGURE 3.1: Vue d'ensemble de notre approche MU_VARCH

une variable principale (un nom et une valeur : fréquence f , voltage v , température T) qui contient le niveau courant pour ce domaine dans la vue concernée. Ces valeurs peuvent être discrètes ou continues, avec bien souvent des plages de valeurs nominales discrètes qui indiquent des niveaux spécifiques autorisés (ou limites) du fait de la conception matérielle physique du SoC. Le cas de la vue applicative est à la fois différent et similaire. Dans ce cas c'est, à l'inverse, un ensemble de nœuds du graphe de tâches qui se trouvent regroupés sur un même bloc de calcul, et les communications regroupés sur les blocs d'interconnexion et/ou de mémoire. Nous soulignons le fait que la relation est donc "de sens contraire" en positionnant la vue applicative au-dessous de la vue architecturale. Nous utiliserons les termes *Abstraction* pour désigner la relation qui lie un Domaine d'une vue non-fonctionnelle à la vue architecturale-support, et *Allocation* (ou *Mapping*) pour la relation qui lie un élément de la vue applicative à son domaine (sa localisation) dans la vue de base. Nous verrons ultérieurement que cette relation peut être plus complexe que généralement reconnu dans les outils de MBSE. Chaque vue peut également contenir une structure propre auxiliaire (les dépendances entre tâches applicatives, les contiguïtés du floorplan de la vue thermique, les hiérarchies des domaines d'horloges et de puissance).

Aspect Comportemental des vues : Au niveau comportemental, la vue de architecturale prévoira l'existence d'un contrôleur (général), pour tenir le rôle joué au niveau concret par le système d'exploitation (*OS Operating System*) et le firmware qui devront opérationnellement allouer et modifier les valeurs et paramètres des autres vues au cours de l'exécution réelle. Ce contrôleur figurera donc le comportement de la vue architecturale, sa fonctionnalité principale étant de décider l'allocation (temporelle et spatiale) des tâches applicatives sur l'architecture en prenant en compte les paramètres non-fonctionnels et leur état dans les autres vues. En ce qui concerne ces autres vues, chaque vue extra-fonctionnelle doit définir comment les valeurs des différents domaines évoluent et sous quel effet. A un domaine donné d'une vue donnée correspondra donc un Clock/Power/-Thermal Manager Unit, selon la terminologie en vigueur (dénotés CMU, PMU, TMU respectivement). Dans le cas (très fréquent) ou une plage de valeurs discrètes a été identifiée par le fabricant du circuit, cela mène à la définition d'une machine à états finis (FSM) par domaine (i.e., par xMU), dont les états sont les valeurs typiques admissibles dans cette

vue. Les transitions entre ces états peuvent être commandées par le contrôleur général qui forme le comportement de la vue de base, ou plus indirectement par (re)positionnement des valeurs des variables en suivant les influences entre vues. En ce qui concerne la vue applicative on peut par analogie considérer l'existence d'états de fonctionnement (idle, active, stand-by) qui reflètent également l'effet du contrôleur de la vue de base, vu comme abstraction d'un *Operating System*, vu sous l'angle unique du déclenchement des tâches. On parlera alors d'AMU (Application Manager Unit) pour notre modélisation.

Relations inter-vues : Le dernier point à considérer est celui des influences et relations de dépendance “directe” entre vues extra-fonctionnelles. Il existe de manière évidente de telles relations entre les vues performances et power (diminuer le voltage et la consommation énergétique réclame généralement de baisser la fréquence du processeur), et entre les vues power et température (le circuit chauffe à proportion de l'énergie consommée). Nous nommerons *Association* ce type de relations inter-vues. Il s'agit en effet de l'association des propriétés des vues. Ce sont en général des abstractions de lois physiques, mais comme il a été dit plus haut on cherchera au maximum à les représenter par un nombre fini de combinaisons discrètes pré-tabulés. En particulier les valeurs cohérentes admises entre valeurs de performance (fréquence) et valeurs de power (voltage) sont dénommées OPP (*Operating Performance Points*) dans la documentation technique des microprocesseurs.

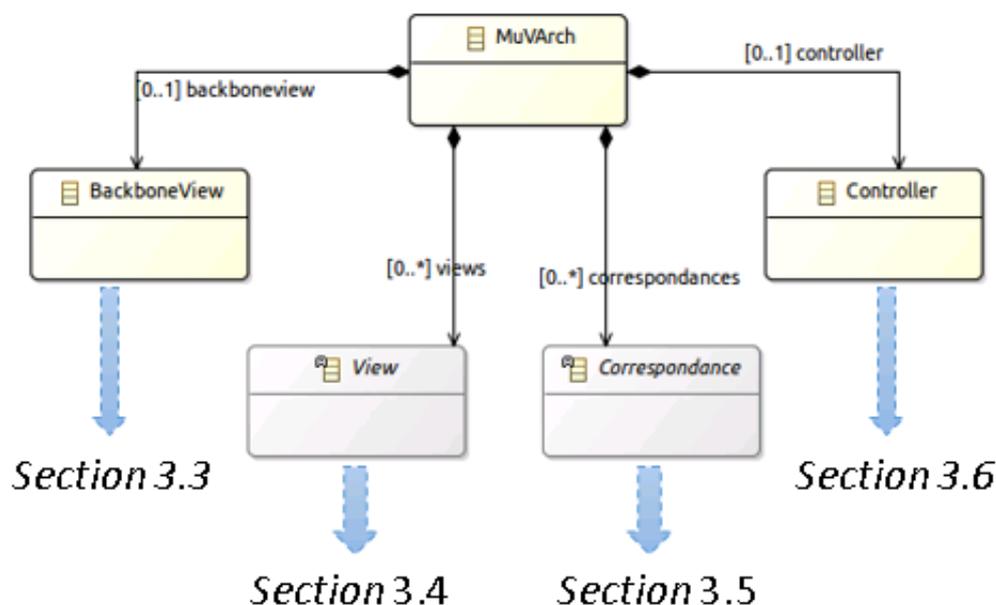
Cas particulier de l'Allocation : En ce qui concerne la vue applicative, on devrait considérer que les valeurs effectives de la relation d'Allocation/Mapping résultent de fait d'un calcul (à la compilation ou à l'exécution) reflétant une politique globale cherchant le meilleur compromis entre efficacité en performance vs efficacité en consommation énergétique, sous contraintes de température. Comme cette équation ne dépend pas uniquement de lois physiques mais également de choix humains et algorithmiques, il est hors de notre propos de tenter de représenter totalement ces aspects. Comme il l'a été mentionné en introduction, nous visons plutôt à procurer un environnement de méta-modélisation dans lequel ce “trou noir” (le composant qui établit la relation de mapping et les niveaux de performance en vitesse et consommation des différents domaines) puisse être représenté, soit par des outils et méthodes d'analyse qui viennent “pêcher” les informations pertinentes de

base dans nos modèles (comme SynDEx), soit (ensuite) être fixé comme solution d'allocation donnée et pré-établie, pour permettre une (co)-simulation du systèmes dans toutes ses vues complexes par traduction vers un environnement-tiers (comme Synopsys MCO Platform Architect).

En conclusion, on voit que nous retenons pour notre approche de modélisation des aspects constants et communs aux différentes vues (domaines et variables d'états associées, plages de valeurs discrètes et Manager Units sous la forme FSM dès que possible, liens entre vues par Association, liens avec la vue de base architecture par abstraction ou Allocation/Mapping). D'autres aspects restent plus fluctuants mais peuvent être simplement ajoutés (si besoin) : Allocations plus sophistiquées d'une communication au niveau de l'application vers plusieurs composants de l'architecture (interconnexions et mémoires). Le cas particulier de la relation d'Allocation, qui peut et doit apparaître comme le résultat algorithmique reflétant les fonctionnalités d'un OS abstrait au niveau de la vue de base, peut se traiter aux deux niveaux et avec les deux types d'activités et d'outils que notre approche propose de considérer uniformément : analyse et optimisation ou validation par simulation).

La figure 3.2 présente un extrait basique du méta-modèle de notre approche. L'élément racine *MuVArch* est le concept de base pour spécifier l'architecture du système à travers les vues. *MuVArch* contient un *BackboneView* qui est la porte vues ; un ensemble de vues (spécialisées après en *PerformanceView*, *PowerView*, *ThermalView* et *ApplicationView*) ; un ensemble de correspondances reliant ces vues (spécialisées après en *Abstraction*, *Allocation* et *Association*) et enfin un contrôleur général. Tous ces éléments de MU_VARCH seront développés respectivement par la suite dans les sections 3.3, 3.4, 3.5 et 3.6. Notons que les classes ayant une couleur grise sont des classes abstraites¹.

1. Une classe est dite abstraite si son implémentation n'est pas complète et qui n'est pas instanciable. En effet, elle sert de base à d'autres classes spécialisées.

FIGURE 3.2: Extrait du méta-modèle de MU_VARCH

3.3 Porte vues : Vue architecturale de base

MU_VARCH est basée sur un socle de modélisation supportant et permettant la définition des autres vues spécifiques du système. Pour des raisons historiques, nous parlerons également de vue pour ce modèle, bien que le terme ne soit pas réellement approprié (c'est un "porte-vues"). Il s'agit donc de la vue architecturale, dénommée *BackboneView* dans la figure 3.3. Cette vue de base fournit la représentation logique du système en cours de développement. Dans le domaine de conception des systèmes sur puces SoC, cette vue définit la structure principale des différents éléments architecturaux (*IPBlock* dans la figure 3.3) comme les processeurs, les mémoires, les blocs de propriété intellectuelle (IP) (IP pour *Intellectual Property*)², les blocs d'entrée/sortie ou les médias de communications. Tous ces blocs peuvent être connectés par des connecteurs à travers leurs ports (définis respectivement *Connector* et *Port* dans la figure 3.3)

Cette vue de base présente la référence des éléments définis dans les autres vues. En effet, les domaines de (puissance, horloge, thermique) représenteront après des abstraction

2. Un bloc de propriété intellectuelle, ou IP est une description textuelle qui permet la génération d'un élément programmable ou matériel que l'on peut inclure dans un circuit intégré.

des éléments architecturaux et les tâches de la vue d'application seront mappés vers ces éléments.

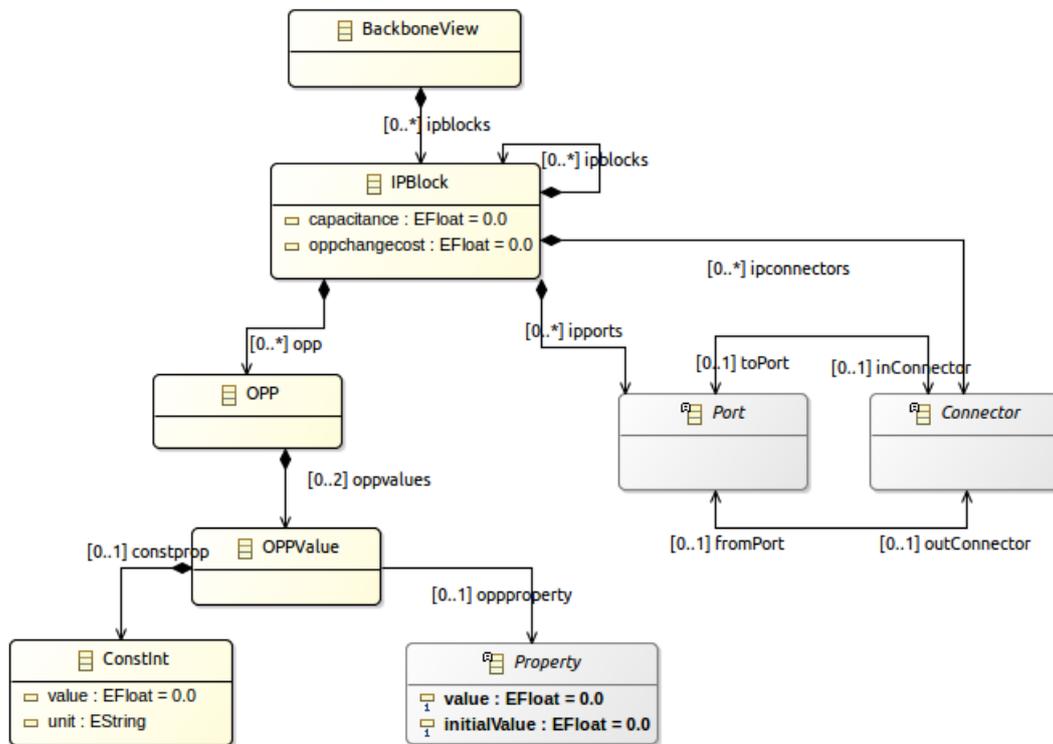


FIGURE 3.3: Méta-modèle de *BackboneView*

Un bloc dans la *BackboneView* peut avoir des différents points de fonctionnements OPP (Operating Performance Point) ; où chaque point de fonctionnement fixe un couple (tension, fréquence) qui peut être choisi en fonction du niveau de performance souhaité. Les points OPP sont fixés par avance par les fabricants du circuit. Selon la figure 3.3, un OPP contient deux valeurs : une valeur pour la fréquence et une autre pour le voltage. Chaque valeur fait référence à une propriété prédéfinie dans la vue non-fonctionnelle (performance et power) et chaque valeur définit une constante. Pour éclaircir plus cette modélisation d'OPP, la figure 3.4 montre un exemple d'OPP.

Au niveau comportemental, la vue architecturale prévoira l'existence d'un contrôleur général. Ce contrôleur qui va réaliser une stratégie d'ordonnancement pouvant allouer et modifier les valeurs et paramètres des autres vues au cours de l'exécution réelle. Nous reviendrons sur cet aspect de contrôle ultérieurement dans la section 3.6 ; après avoir parlé des autres vues du système et leurs interconnexions.

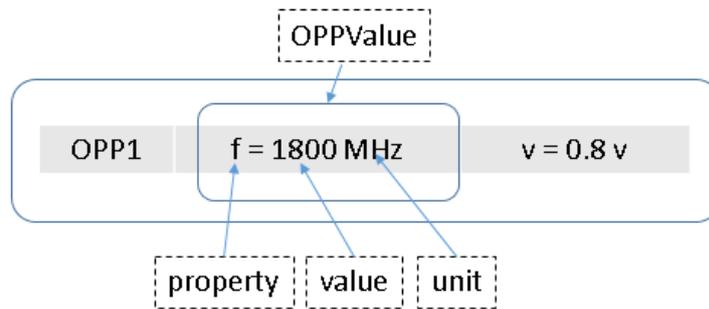


FIGURE 3.4: Exemple d'un OPP

3.4 Autres vues spécialisées

Les autres vues de MU_VARCH viennent étoffer la base architecturale en ajoutant des propriétés non-fonctionnelles dans les vues performance, puissance et thermique et des propriétés fonctionnelles dans la vue applicative. Ces vues comprennent des aspects structurels et comportementales communes (présentés dans la figure 3.5) et d'autres aspects propres à chaque vue. Nous exposons ces aspects dans les sections suivantes.

3.4.1 Aspects communs des autres vues

Les aspects communs des vues sont présentés dans le méta-modèle de la figure 3.5. Au niveau structurel (la partie (a) du méta-modèle 3.5), ces vues possèdent des éléments structurels (*StructureElement*) spécialisé en domaines et tâches selon la vue étudiée. Chaque élément structurel représente un groupe distinct de composants de la plate-forme architecturale de base. La vue performance se base sur un certain nombre de domaines d'horloge (*CD Clock Domain*) où chaque CD englobe un nombre de blocs architecturaux s'exécutant avec la même fréquence d'horloge. Similairement, la vue puissance (*power*) se base également sur une partition de l'architecture ayant le même voltage, cette fois en domaines de puissance (*PD Power Domain*). La vue thermique, de son côté, se base sur un *floorplan* ou des domaines thermiques (*TD Thermal Domain*), qui consiste également en une partition de la structure architecturale, avec cette fois-ci également une relation de voisinage pour la diffusion de la chaleur entre zones contiguës. Enfin, la vue applicative introduira un modèle de l'application à base de graphe de tâches qui partitionnera les

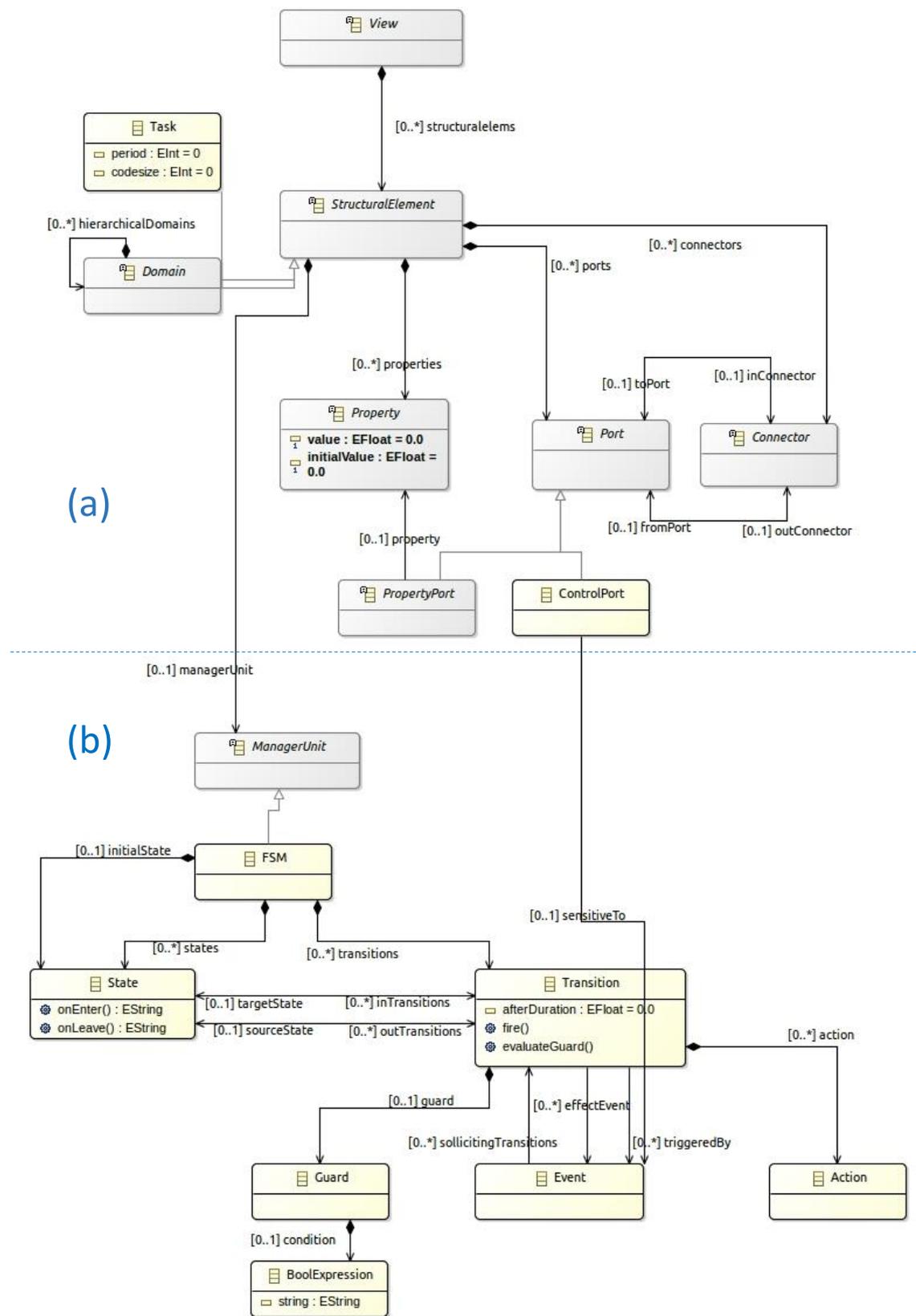


FIGURE 3.5: Méta-modèle d'une vue dans MuVARCH

tâches applicatives et leur connexions en fonction des ressources (de calcul, de stockage, de communication, d'entrées-sorties) de la structure architecturale.

Les éléments structurels peuvent être liés par des connecteurs via des ports. Particulièrement, les tâches de la vue de l'application peuvent échanger des données. Ces ports sont soit un port spécifiant une propriété (*PropertyPort* dans la figure 3.5) soit un port de contrôle (*ControlPort*). A chaque domaine est associée une (ou plusieurs) propriété (*Property*) comme la fréquence, la température et le voltage. Ces propriétés sont associées après entre eux pour l'harmonisation de l'exécution inter-vues (section 3.5.1).

Au niveau comportemental, chaque élément structurel (domaine ou tâche) est contrôlé par une unité de contrôle, dénommée *Manager Unit* dans la figure 3.5. Cette unité est décrite par une machine à états finis FSM (*Finite State Machine*).

Une FSM contient des états du système liés par des transitions. Un état représente un mode de comportement (idle/actif, on/off, cool/hot, etc) selon la vue spécifique. Une transition est :

- (a) soit activée par un événement déclencheur et peut par suite exécuter une action définie (par exemple, dans une vue de performance, suite à un événement reçu du contrôleur général, un domaine d'horloge passe d'un état "speed" à un état "medium" et une action $f = 800\text{Mhz}$ est exécutée) ;
- (b) soit activée suite à une réponse vraie d'une condition de garde et lance ainsi un événement déclencheur pour synchroniser les autres vues. Cette condition de garde est une expression logique sur les propriétés des domaines (par exemple, la température d'un domaine thermique chauffe trop et dépasse un seuil prédéfini, ce domaine doit changer ainsi de mode d'exécution "hot" vers "cold" tout en envoyant un événement au contrôleur pour changer et synchroniser les paramètres des autres vues, notamment la fréquence et le voltage).

Les domaines et les tâches reçoivent/envoient les notifications et événements de/vers le contrôleur général à travers leurs ports de contrôles.

3.4.2 Spécialisation des vues non-fonctionnelles

3.4.2.1 Vue performance

La figure 3.6 présente le méta-modèle spécialisé de la vue performance (*Performance View* nommée aussi *Clock View*). Cette vue est basée principalement sur des domaines d'horloges. Chaque domaine d'horloge englobe un ensemble de blocs de l'architecture dont ils forment une partition, et peut inclure éventuellement des sous domaines hiérarchiques. En plus, à chaque domaine de cette vue est associé une propriété de type fréquence.

Du côté comportemental, un *Clock Manager Unit CMU* correspond à chaque domaine d'horloge. Le CMU est une spécialisation de *ManagerUnit* de la figure 3.5, ainsi un CMU est décrit par une FSM. Le rôle de ce CMU est de sélectionner et changer les valeurs de fréquences dépendants des entrées et événements provenant des autres vues. Ces valeurs de fréquences appartiennent aux couples (f,v) des points de fonctionnements ou OPP (*Operating Performance Points*) prédéfinis par les fabricants de circuits. Le changement de point de fonctionnement suit la technique de DVFS [43], afin de conserver les performances optimales du système.

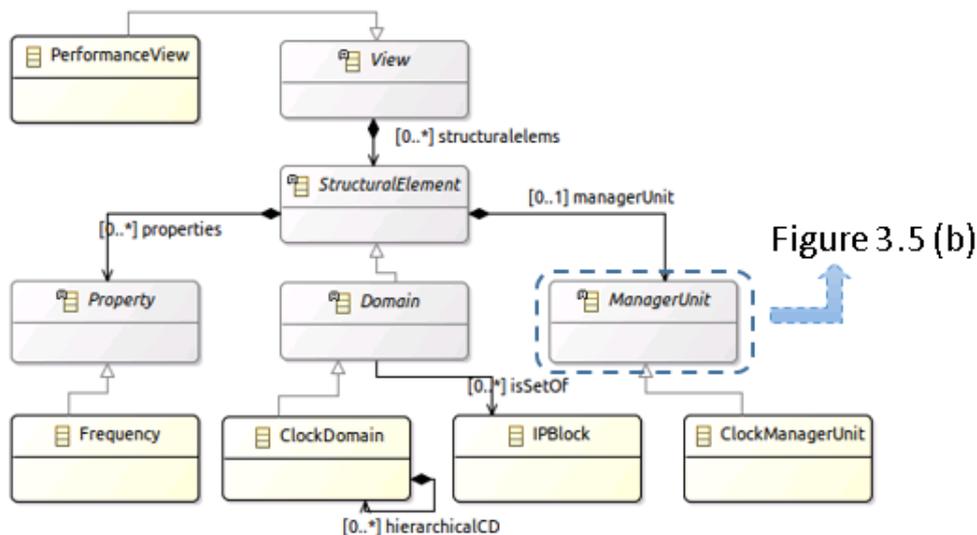


FIGURE 3.6: Spécialisation de la vue de performance

3.4.2.2 Vue puissance

La figure 3.7 présente le méta-modèle de la vue de puissance (power). Similairement à la vue performance, la vue power est basée sur des domaines (pouvant être hiérarchiques aussi) dont chacun englobe un ensemble de blocs définis dans la vue de base architecturale ; mais pour cette vue, il s'agit des domaines de puissance (ou de voltage).

Chaque domaine de voltage est contrôlé par un *Power Manager Unit PMU* (une spécialisation d'un *ManagerUnit* présenté dans la figure 3.5) qui permet de changer entre un nombre d'états admissibles (off, idle, standby, active,..) et ainsi changer le voltage correspondant à chaque état du système. Comme la fréquence dans la vue performance, les valeurs de voltage sont sélectionnées à partir des OPP prédéfinis par le constructeur du circuit. En effet, la technique de DVFS [43] permet de changer le couple de (voltage/fréquence) du bloc hardware lors de l'exécution et ainsi d'adapter sa puissance (et donc sa consommation) à la charge réelle nécessaire pour l'exécution d'une application. Le changement des états de la FSM associé à chaque PMU est effectué par des événements reçus du contrôleur général de la vue architecturale. Ces états dépendent aussi de la température du circuit car ce dernier chauffe à proportion de l'énergie consommée.

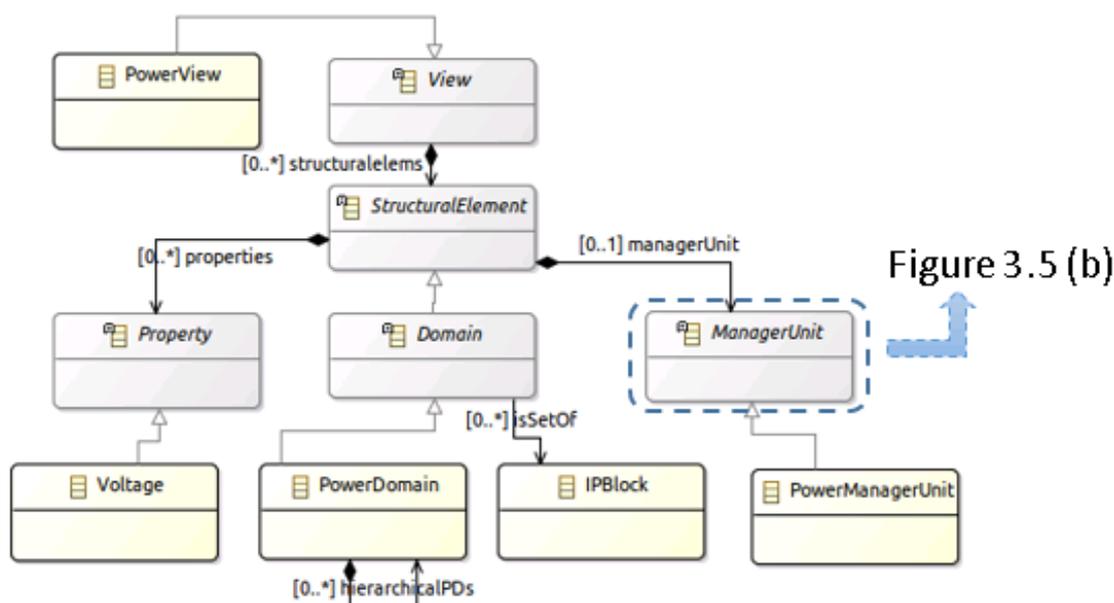


FIGURE 3.7: Spécialisation de la vue de puissance

3.4.2.3 Vue thermique

La figure 3.8 présente la spécialisation du méta-modèle conçu pour la vue thermique. Cette vue décrit des domaines thermiques spécifiés par les experts de la thermique afin de représenter les caractéristiques thermiques de la vue *backbone*. Précisément, la vue thermique se base sur un *floorplan* ou des domaines thermiques, qui consiste en une partition de la structure architecturale, avec cette fois-ci une relation de voisinage pour la diffusion de la chaleur entre zone contiguës.

Également, chaque domaine thermique dispose d'une unité de management thermique TMU décrite par une FSM (figure 3.5). Les transitions dans cette FSM diffèrent des autres FSMs dans les CMUs et les PMUs. En effet, les transitions ici dépendent d'une condition de garde booléenne sur la propriété de cette vue : la température. Une fois la température dépasse un seuil prédéfini, un événement est déclenché par la TMU pour changer l'état du système en diminuant ou augmentant le niveau de fréquence et voltage dans la table des OPPs.

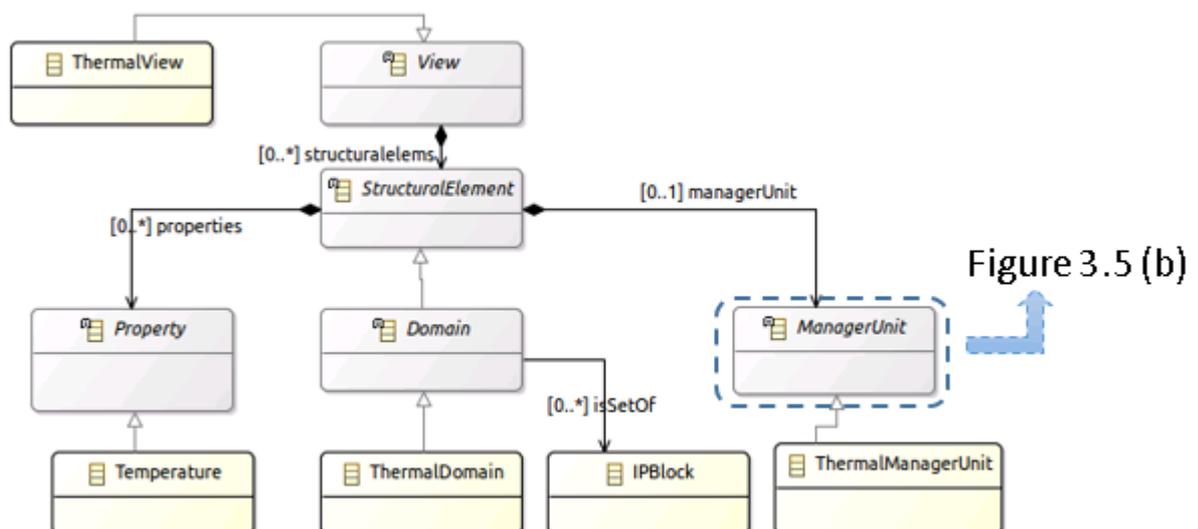


FIGURE 3.8: Spécialisation de la vue thermique

3.4.3 Spécialisation de la vue applicative

La vue d'application, dont le méta-modèle est représenté dans la figure 3.9, introduit un modèle de scénario d'exécution à base de graphe de tâches. Celui ci sera ultérieurement mappé et exécuté sur l'architecture. Éventuellement, la tâche peut imposer une vitesse d'horloge pour son exécution sur un processeur dans le hardware, ou encore une configuration d'énergie. Également, l'application pourra être de type *data flow* (section 2.3.1) et aura donc d'échange de données sur ces canaux de communications. La tâche a une latence sous forme de nombre de cycles de calcul et optionnellement une période. Il faut ici être bien conscient que ce nombre donné en temps logique ne se traduira en une durée physique que une fois établie une fréquence (vitesse) du processeur hôte. Pour les tâches périodiques, on supposera que la période est supérieur à la durée d'exécution.

Chaque tâche de l'application dispose d'une unité de management AMU (*Application Manager Unit*, ainsi nommée par similitude aux autres unités CMU, TMU et PMU). Cette AMU est représenté par une FSM décrivant les états d'exécution de chaque tâche (execute, stop, standby...).

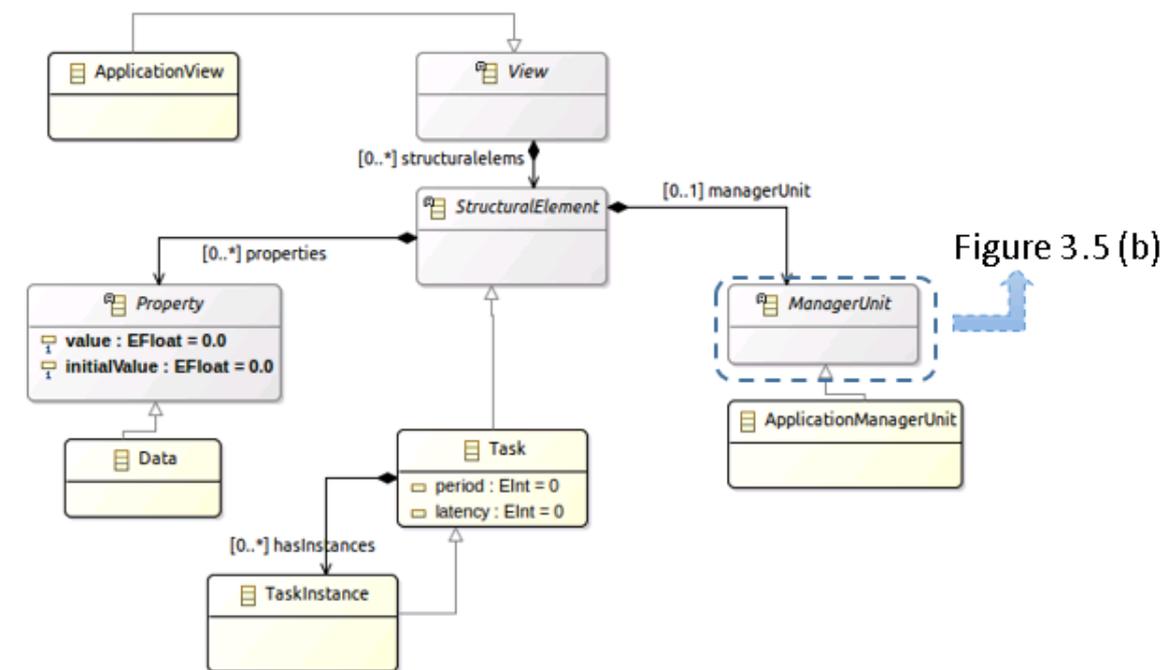


FIGURE 3.9: Spécialisation de la vue d'application

3.5 Relations entre vues : Abstraction et Association

Nous avons décrit les différentes vues distinctes qui sont nécessaires à l'approche multi-vue *low-power* retenues dans le projet HOPE et préconisées dans MUVARCH. ces vues ne sont naturellement pas autonomes, et il nous faut maintenant définir et décrire leurs inter-relations. Celles-ci seront de deux ordres :

- (a) les relations; dénommées Associations en MUVARCH; qui viendront décrire les dépendances au niveau physique entre vues distinctes (ici les liens entre performance et consommation énergétique, consommation et température,...).
- (b) les relations qui indiquent comment les vues extra-fonctionnelles sont portées sur la vue de base de la plate-forme d'exécution. Il s'agira ici principalement des relations d'Abstraction, qui indiquent comment ces vues regroupent les éléments architecturaux en domaines, avec les implications concernés.

Il existe un autre type de relation dans cette rubrique, la relation d'Allocation entre la Vue Applicative (fonctionnelle) et la plate-forme. Vu sa particularité et sa complexité celle-ci sera décrite dans une section spécifique plus loin (section 3.6), comme résultant des calcul du Contrôleur Général de MUVARCH, lui-même basé sur les informations procurées par les vues décrites dans cette section courante ;

3.5.1 Association

La première association que nous identifions est celle entre la vue performance et la vue power. Cette relation est exactement l'objectif de la technique de DVFS [43], qui vise à réduire la tension (et la vitesse d'horloge) pour économiser la consommation d'énergie. Ce couple tensions-horloges constituent une table de point de fonctionnement (les OPP *Operating Performance Points*). Ainsi, un changement de niveau dans l'OPP conduit à un changement des deux propriétés des deux vues performances et power (f et v).

La deuxième association inter-vues est celle entre la vue thermique et la vue power. En effet, la consommation d'énergie augmente avec l'augmentation de la chaleur. Par exemple, le comportement de la température dans la vue thermique contient un état

caractérisé par l'équation thermique $\frac{dT}{dt} = \frac{P}{C_{th}} + \frac{1}{R_{th}C_{th}} (T - T_{env})$, où T est la température du processeur (le processeur est ici un élément abstrait de la vue architecturale de base), C_{th} et R_{th} sont respectivement la capacitance et résistance thermique du processeur, T_{env} est la température de l'environnement du processeur et P est la consommation de power définie dans la vue power et décrite par l'équation $P_{dynamique} = C * V_{dd}^2 * f$.

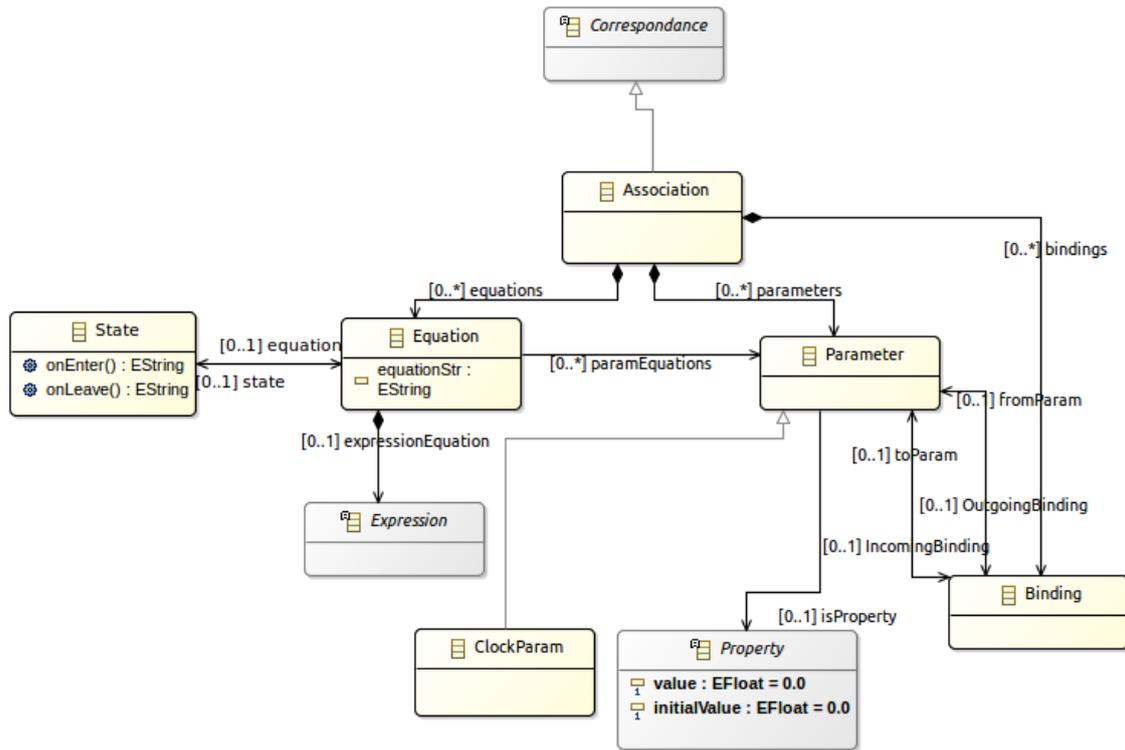


FIGURE 3.10: Relation d'association dans MUARCH

L'association permet la liaison des propriétés des différentes vues pour définir les contraintes physiques de l'environnement. Cette correspondance définit l'aspect continu du système à travers des représentations de contraintes ou d'équations à des fins d'analyse. En effet, *Abstraction* permet de représenter des contraintes sur les valeurs de paramètres système tels que performance, fiabilité, masse, etc. Elle fournit ainsi un support servant après pour les études d'analyse système. Plus précisément, cette correspondance définit l'évolution des propriétés non-fonctionnelles spécifiées dans les autres vues du système. Cette évolution est représentée par des équations *Equation* décrivant les relations acausales entre des paramètres du système *Parameter*. Ces paramètres peuvent représenter les

propriétés des vues (fréquence, température, puissance), ou les paramètres de l'architecture (capacité, intensité) ou également le pas d'exécution à travers *ClockParam* qui sert à recevoir les événements exécutant l'évaluation des équations. Toutes ces paramètres sont connectés à travers *Binding*.

L'association est ainsi une liaison des propriétés des différents vues (f, v, T) pour définir les contraintes physiques de l'environnement. Elle fournit un support servant après pour les études d'analyse système. Nous modélisons cette relation dans MU_VARCH par le méta-modèle figuré dans 3.10. L'association est représentée par des équations décrivant les relations acausales entre des paramètres du système. Ces paramètres peuvent représenter les propriétés des vues (fréquence, température, puissance), ou les paramètres des IP Blocs (tels que la capacité et la résistance) ou également le pas de temps d'exécution qui sert à recevoir les événements exécutant l'évaluation des équations.

3.5.2 Abstraction

Dans les vues des propriétés non-fonctionnelles (performance, power et thermique), chaque domaine structurel encapsule un ensemble de IP block de l'architecture définie dans la *Backbone View*; c'est ce que nous appelons *Abstraction*. Précisément, cette correspondance relie des éléments de type *PowerDomain*, *ClockDomain* et *ThermalDomain* avec des éléments de type *IPBlock*, comme présenté dans la figure 3.11.

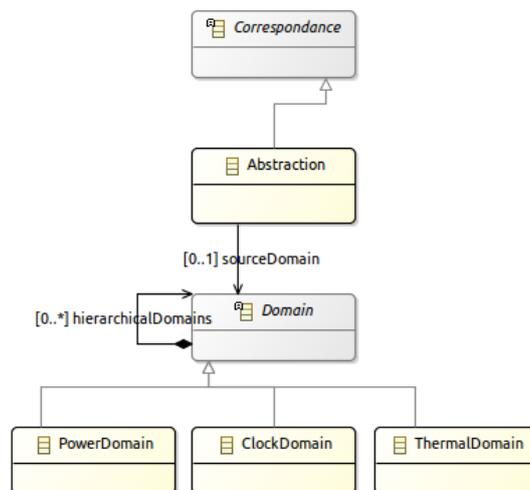


FIGURE 3.11: Relation d'abstraction dans MU_VARCH

3.6 Contrôleur général de MuVArch et Relation d'Allocation

On voit dans la figure 3.1 le rôle central joué par le *Controller* (en rouge) qui vient figurer la partie comportementale de la vue de base “*Backbone*”, et qui interagit de manière centralisé pour calculer le fait l'Allocation (= distribution spatiale + ordonnancement temporel) des tâches applicatives sur les ressources architecturales (de calcul et de communications), en optimisant certains critères à base d'informations (de performance, de power,...) et changement procurées par les vues du haut non-fonctionnelles. Des exemples concrets de ces changements sont : les sondes de température font qu'on ralentit ou on migre des tâches applicatives pour éviter les points chauds (*hot spots*), le niveau de batterie peut commander le ralentissement ou la mise en veille de composants (protocoles de communications, illumination écran, etc...). En ce sens le Contrôleur reprend les fonctionnalités traditionnelles d'un *Operating System*, (allocation et ordonnancement), à la différence que nous voulons traiter de ces questions au niveau des modèles et en autorisant des approches analytiques d'une part (à *compile time*), et que nous voulons également reporter de tels résultats pour les exploiter en simulation, encore une fois au niveau des modèles, dans des environnements de simulation du type de *Synopsys MCO Platform Architect* par exemple.

Dans le premier rôle, celui où notre environnement MU_VARCH servira à supporter une approche formelle où des méthodes algorithmiques/mathématiques seront utilisées pour calculer une allocation à partir de la combinaison des paramètres fournis entre toutes les vues, il est hors de notre propos de modéliser ces stratégies de calcul, que nous supposons implantées dans un outil tiers de résolution (externe). Du côté de la modélisation MU_VARCH, il sera seulement demandé d'exprimer les relations causales de tout ordre intra-vues et inter-vues dans un langage textuel de contraintes. Nous abordons partiellement ce point en sous section 3.6.1. Nous présenterons ensuite, dans le prochain chapitre 4, les détails techniques de la connexion de MU_VARCH avec un tel outil prototype.

Nous voulons avoir la possibilité dans MU_VARCH de représenter le résultat d'un tel calcul d'allocation, afin de savoir le “rejouer” en simulation avec une relation d'allocation

maintenant fixé. Ce second niveau peut également être considéré dans le cas, en fait bien plus courant, où une allocation est donnée de manière impérative par l'utilisateur, et où la simulation n'est utilisée "que" pour explorer différentes possibilités d'allocation et les variations qu'elles induisent dans l'espace des solutions du design.

3.6.1 Contraintes issues des relations de MuVArch

L'outil de résolution d'ordonnancement que nous utilisons est un solveur Satisfaisabilité Modulo Théories (SMT) [74] dont l'approche est illustrée dans la figure 3.12. Dans ce solveur, le problème du placement et de l'ordonnancement est codé sous forme de contraintes logiques sur un macro-cycle calculé. Ces contraintes sont exprimées par des informations extraites de modélisation de MU_VARCH. Au premier temps, ces informations sont décrites par une syntaxe concrète textuelle (éventuellement inspiré des scripts Python). Celles ci portent sur les spécifications de l'architecture et de l'application tout en considérant les propriétés non-fonctionnelle telles que la performance et le voltage. Ainsi, les tâches de l'application, leurs périodes, leurs temps d'exécution sur une ressource donnée sont envoyés vers le solveur. Les ressources d'exécution et leurs OPPs (couple de voltage et fréquence) sont également spécifiés pour le solveur.

Afin de soumettre au solveur les contraintes requises, nous avons besoin de factoriser les informations collectées sous un certain nombre d'hypothèses de validité (de type OCL (*Object Constraint Language*)) sur une allocation. Notons que ce travail est encore en cours ; nous décrivons néanmoins quelques contraintes cibles.

Supposons que T_i est une instance i de la tâche T , $T_{i,r}$ est l'exécution de l'instance T_i sur la ressource r et $P[T_i]$ est la période de la tâche T_i :

$$(C1) : stop_time[T_i] = start_time[T_i] + P[T_i]$$

$$(C2) : start_time[T_{i+1}] \geq stop_time[T_i]$$

$$(C3) : deadline_time[T_i] \geq stop_time[T_i]$$

$$(C4) : \mu[T_{i,r1}] + \mu[T_{i,r2}] \leq 1$$

La contrainte (C1) affirme que le temps d'arrêt d'exécution de l'instance T_i est égale à sa période plus la date de début d'exécution. La contrainte (C2) affirme que l'instance T_{i+1} ne doit pas être exécuter avant l'instance T_i . La contrainte (C3) assure que le *deadline* ne doit pas dépasser la date d'arrêt. Enfin, la contrainte (C4) énonce que la même instance de tâche ne peut pas être exécuter sur deux coeurs différents en même temps. Notons qu'à ce moment, ces contraintes sont calculés et décrites par le solveur SMT.

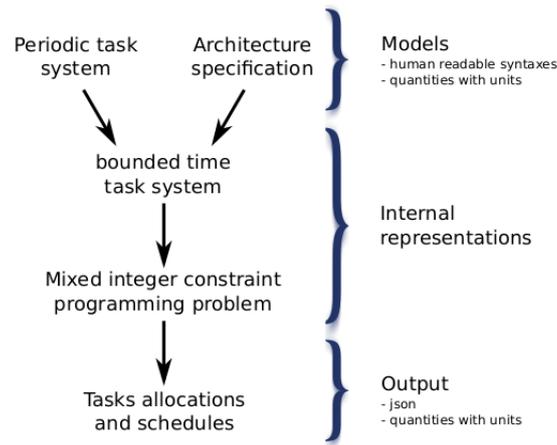


FIGURE 3.12: Approche du solveur SMT existant [74]

L'ordonnancement calculé est injecté ensuite dans MUVARCH pour pouvoir enrichir le contrôleur général et représenter son comportement. Ce que nous introduisons dans la section suivante.

3.6.2 Relation d'Allocation effective

Le modèle d'allocation est destiné à représenter le comportement du contrôleur général (figure 3.1 (e)) *après* calcul d'un ordonnancement. Il nous faut donc nous interroger sur la forme que doit avoir l'objet représentant une allocation entre tâches applicatives et ressources architecturales, prenant en compte des contraintes extra fonctionnelles exprimées dans les vues du même nom, dans le cas de notre approche MUVARCH. A la base, une allocation prendra la forme d'une trace parallèle d'exécution.

Une allocation élémentaire (*Elementary Allocation* dans la figure 3.13) alloue une instance de tâche (= un "job") sur une ressource donnée. Il est possible que différentes

instances soient allouées sur des processeurs différents, pour obtenir une représentation finie on demandera que cette allocation soit au moins ultimement périodique. L'allocation générale consiste à allouer chacune des tâches sur des ressources (soit l'ensemble des instances de toutes les tâches).

Enfin, les traces (finies ou périodiques) d'exécution devront indiquer les niveaux des valeurs de paramètres dans les diverses vues (quel sera la fréquence d'horloge à l'exécution, le voltage, etc...). Il est à noter que des règles de cohérence (plusieurs tâches actives sur le même coeur en même temps devront utiliser le même OPP,...) devront être vérifiées, ce que l'approche MU_VARCH rendra possible (mais que nous n'avons pas encore implanté).

La figure 3.13 décrit le méta-modèle de la relation d'allocation. On voit qu'au coeur de la modélisation, il y a une relation (*Elementary Allocation*) associant une Instance de tâche à une ressource (un composant) de calcul de la vue architecturale, ainsi qu'une OPP disponible sur cette ressource, et une date de début (et de fin) d'exécution de cette instance de tâche (job) sur cette ressource.

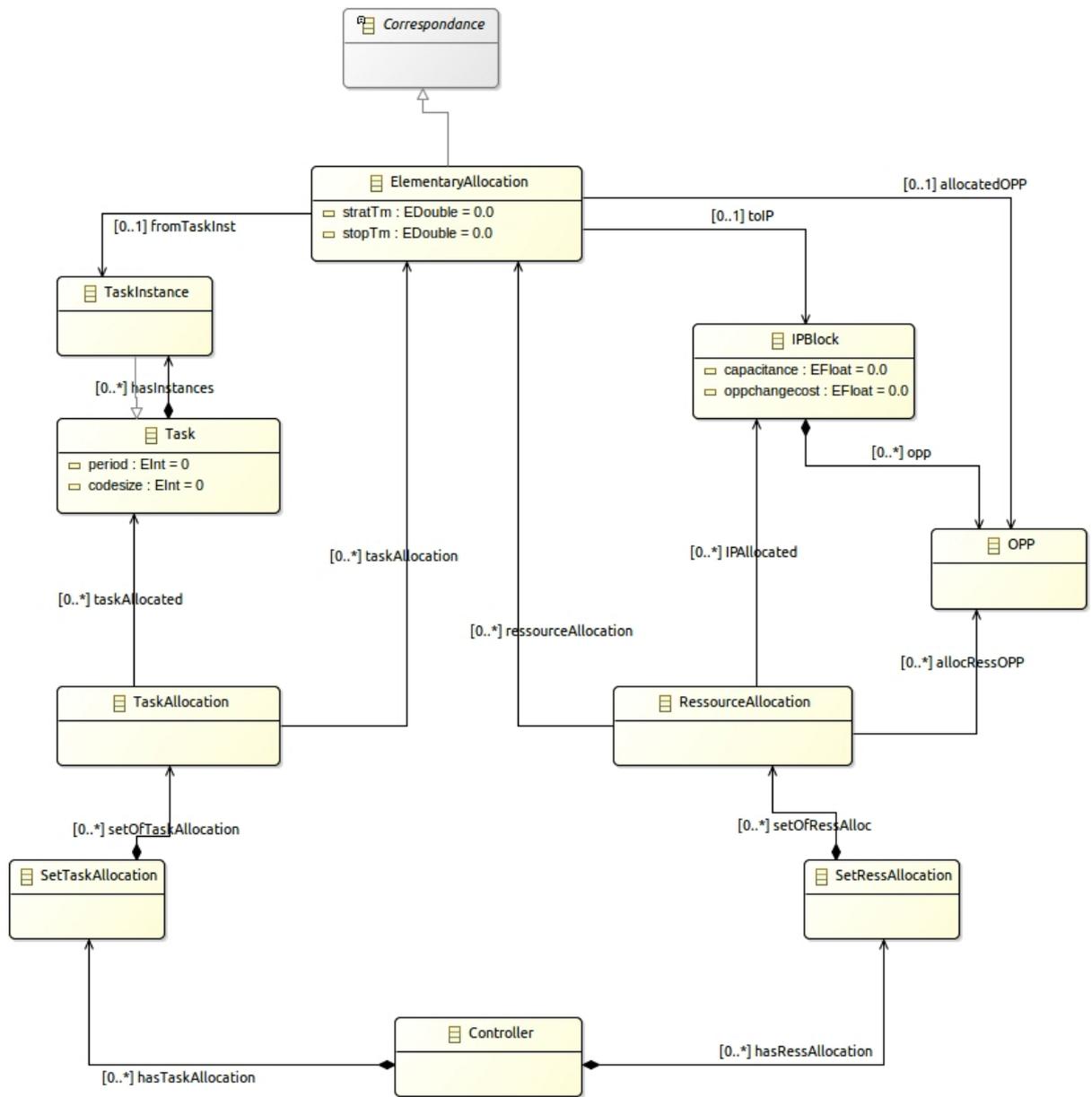


FIGURE 3.13: Relation d'Allocation dans MUVARCH

On voit que pour la représentation, on peut factoriser une allocation globale de deux manières :

- (a) en procurant pour chaque tâche (en ligne) la suite des allocations de ses instances, graduées sur une échelle temporelle, et contenant le nom de la ressource comme valeur ; comme l'exemple présenté dans la figure 3.14

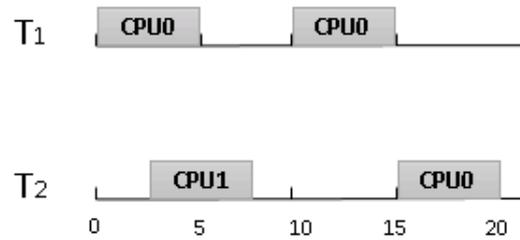


FIGURE 3.14: Allocation globale pour les tâches

- (b) alternativement, en procurant pour chaque ressource (en colonne), la suite des instances de tâches qui s'y exécutent (exemple figure 3.15).

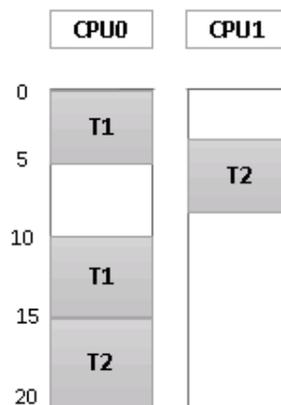


FIGURE 3.15: Allocation globale pour les ressources

3.6.3 Représentation alternative d'allocation comme FSM

La trace parallèle (et annotée) résultant de l'allocation peut également être représentée de manière sérialisée, puisque les débuts et fins de tâches sont formulés avec une

datation synchrone. Le défaut de cette représentation est de séparer l'événement de début d'une instance de tâche d'avec l'événement de la fin ; et de mélanger tous ces événements (début/fin) en une même séquence comportementale banalisée. L'avantage est que la représentation à base de machine d'états FSM (*Finite State Machine*) est plus souvent disponible dans les outils de méta-modélisation qu'une représentation très large d'objets de type "sequence diagrams". Nous procurons ci dessous la méta-modélisation alternative de l'allocation de contrôleur en FSM.

La figure 3.16 présente le méta-modèle de contrôleur en FSM. Le contrôleur définit tous les événements en relation avec les autres vues de l'architecture. Ces événements seront déclenchés qu'après la réception du résultat de calcul d'allocation. Ce dernier sera enregistré dans le contrôleur sous forme d'une machine à états finis FSM. Les transitions de cette FSM présenteront le début ou la fin de chaque allocation d'une tâche vers une ressource architecturale. Nous exploitons un cas d'étude en projetant en détails les aspects de cette FSM dans le chapitre suivant 4.

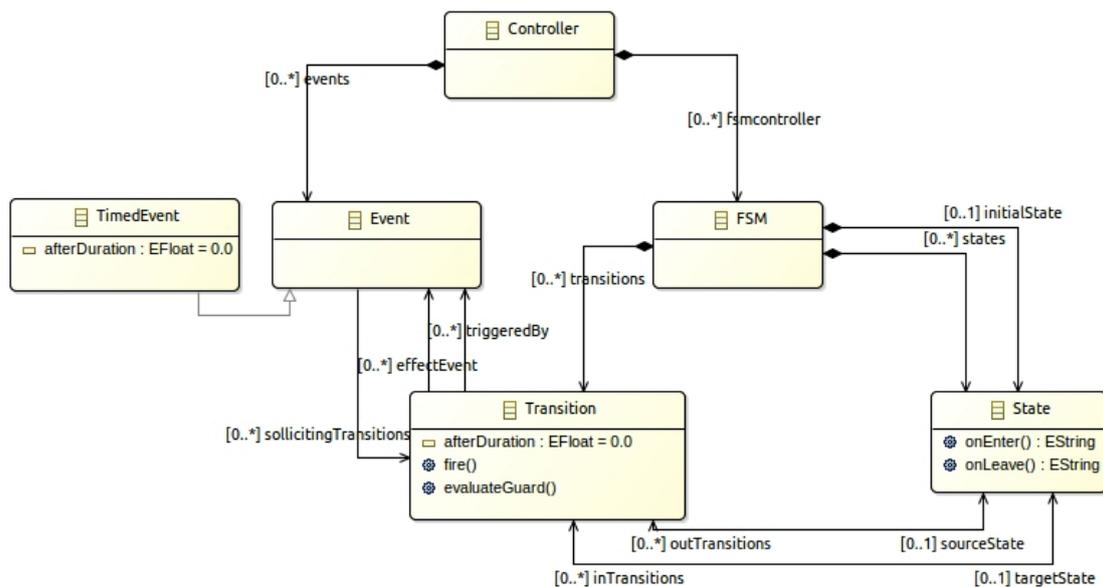


FIGURE 3.16: Méta-modèle du contrôleur général dans MUVARCH

3.7 Conclusion

Dans ce chapitre, nous avons présenté une approche de la modélisation en haut niveau des propriétés fonctionnelle et non-fonctionnelle des systèmes sur puce embarqués. Cette approche, appelée MUVARCH, repose sur la modélisation multi-vues. Tout d'abord, nous avons souligné l'allure globale de notre approche. Celle-ci est fondée sur une base architecturale décrivant la plateforme d'exécution. A cette base, nous ajoutons les autres vues appropriées pour prendre en compte les aspects non-fonctionnels : performance, power et thermique, ainsi qu'une vue applicative pour exercer la plateforme d'exécution.

Nous avons détaillé ensuite les méta-modèles des différentes vues de notre approche et les inter-connexions entre eux. Particulièrement, nous avons montré que la relation d'allocation, étant plus complexe, est un résultat du calcul fait dans le contrôleur général et qui en devient donc le comportement dans une seconde phase (par exemple pour simuler cette allocation dans le système et en retirer des informations, sur la variation de la température notamment). Dans le chapitre suivant, nous décrivons l'instrumentation de cette approche dans un atelier logiciel basé sur un ensemble d'outils du domaine de l'IDM. Nous illustrons l'utilisation de cet atelier et l'application de notre approche sur un cas d'étude.

APPLICATION ET VALIDATION :

MUVARCH FRAMEWORK

Sommaire

| | | |
|------------|---|-----------|
| 4.1 | Introduction | 71 |
| 4.2 | Atelier logiciel pour l'instrumentation de MuVArch | 71 |
| 4.3 | Présentation de l'architecture <i>big.LITTLE</i> | 74 |
| 4.4 | Modélisation de notre cas d'étude avec MuVArch | 77 |
| 4.4.1 | Modélisation de la Porte Vues (<i>Backbone View</i>) | 80 |
| 4.4.2 | Modélisation de la vue Performance | 81 |
| 4.4.3 | Modélisation de la vue Puissance | 82 |
| 4.4.4 | Modélisation de la vue Thermique | 83 |
| 4.4.5 | Modélisation de la vue Application | 84 |
| 4.4.6 | Prise en compte du contrôleur général en Sirius | 85 |
| 4.5 | Connexion avec un solveur SMT existant | 87 |
| 4.5.1 | Génération du code pour le Solver SMT | 87 |
| 4.5.2 | Calcul des contraintes et de l'ordonnancement | 90 |
| 4.5.3 | Prise en compte de l'ordonnancement par le contrôleur de MU- VARCH | 91 |
| 4.6 | Conclusion | 96 |

4.1 Introduction

L’approche MUVARCH est une approche de méta-modélisation qui vise la définition de modèles de systèmes à partir de différents points de vues spécifiques venant étoffer une base de modélisation architecturale commune. Ces vues sont inter-connectés et synchronisés à travers des relations de correspondances. Après avoir présenté MUVARCH dans le chapitre précédent, nous proposons dans le chapitre présent (section 4.2) une instrumentation de notre approche. Cette instrumentation est réalisée pour la mise en place d’un atelier logiciel en se basant sur un ensemble d’outils.

Ensuite, nous introduisons dans la section 4.3 un cas d’étude pour valider notre approche MUVARCH. Notre cas d’étude portera sur une architecture *big.LITTLE* constituée principalement de deux processeurs (quadri-coeurs) un “*big*” et l’autre “*LITTLE*”. Le premier processeur dispose d’une haute performance et par conséquent d’une consommation d’énergie élevé ; tandis que le deuxième processeur dispose d’une performance plus faible et ainsi une énergie moins que le premier processeur.

Dans la section 4.4, nous proposons sa modélisation avec notre langage MUVARCH en modélisant tout d’abord la porte-vues (*Backbone View*), ensuite les différentes vues non-fonctionnelles(performance, puissance et thermique), la vue applicative et le contrôleur.

D’après ce modèle, nous visons un solveur SMT existant pour résoudre le problème du placement et de l’ordonnancement des différentes tâche de notre scénario d’exécution vers l’architecture *big.LITTLE* (section 4.5). Enfin, nous injectons les résultats retournés dans notre modèle pour avoir un modèle enrichi avec les différents allocations (section 4.5.3).

4.2 Atelier logiciel pour l’instrumentation de MuVArch

Après avoir présenté l’allure globale de notre approche MUVARCH ainsi que les différents liens entre ses vues, nous essayons désormais d’instrumentaliser notre proposition. Comme nous l’avons vu dans les chapitres précédents, notre approche s’inscrit dans le

contexte de l'IDM. Nous avons souhaité donc que notre framework repose sur des standards de ce domaine et une chaîne d'outils permettant de décrire des méta-modèles et capable de générer automatiquement du code à partir de ceux-ci. Cette chaîne regroupe essentiellement trois outils s'exécutant dans un IDE (Integrated Development Environment) Eclipse¹ : *EMF* (section 2.4.3), *Sirius*(section 2.4.3) et *Acceleo*. Les deux derniers ont été développés principalement en tant que *Eclipse Modeling Projects* par la société Obeo².

La figure 4.1 présente l'architecture technique de notre atelier de conception de MU_VARCH, la manière avec lesquels les différents outils sont inter-connectés ainsi que la chaîne de transformations que nous utilisons pour la génération d'un modèle enrichi avec un ordonnancement calculé par un solveur SMT existant.

La chaîne dans la figure 4.1 s'appuie sur 5 étapes :

1. La première étape concerne la création de notre langage de modélisation MU_VARCH à l'aide de EMF. Ce langage était le sujet du chapitre 3 précédent.
2. La deuxième étape est la création d'un modèle conforme au méta-modèle de MU_VARCH à l'aide d'une interface graphique. Celle ci est réalisée avec l'outil Sirius et s'appuie sur le code généré par EMF. Dans ce qui suit, nous partons sur une architecture *big.LITTLE* comme modèle. Ce dernier constitue le point d'entrée pour notre moteur de génération de codes Acceleo.
3. La troisième étape exécute les règles de transformations et de génération de codes, définis dans des templates Acceleo. Cette étape produit trois fichiers servant pour le calcul de l'ordonnancement.
4. La quatrième étape applique d'un solveur SMT tiers pour le calcul de l'ordonnancement. Ce solveur prend en entrée les fichiers décrivant l'application et l'architecture et générés de l'étape précédente et génère en sortie un fichier d'extension ".json" contenant un ordonnancement convenable des tâches de l'application vers les ressources de l'architecture spécifié. Ce travail a été réalisé conjointement avec *Emilien Kofman*, autre doctorant dans notre équipe de recherche.

1. <https://www.eclipse.org/>

2. <https://www.obeo.fr/fr/produits/projets-eclipse>

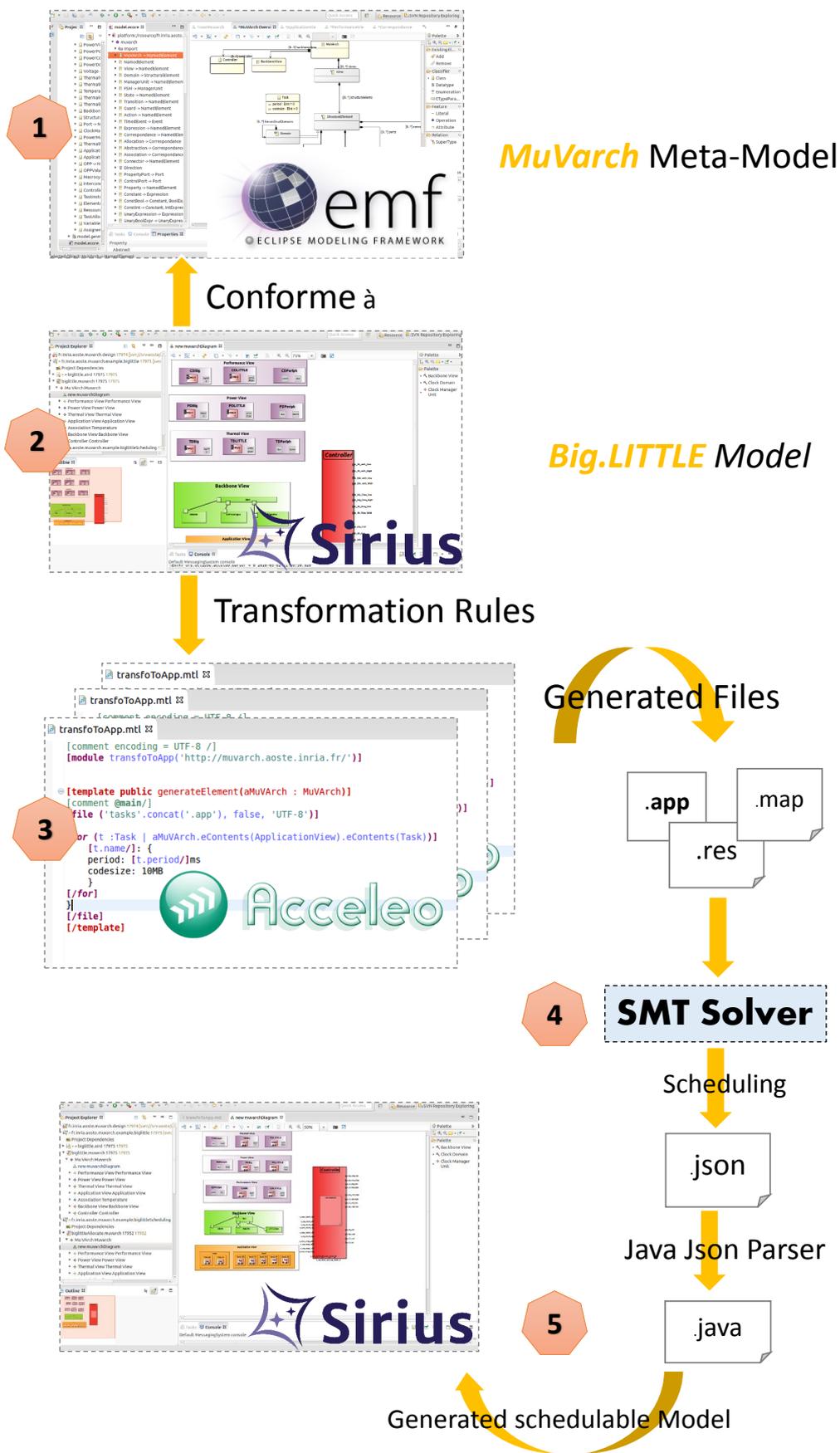


FIGURE 4.1: Architecture technique de notre atelier d'instrumentation de MUVARCH

5. La dernière étape de ce processus cherche à obtenir le code MU_VARCH complet après l'ordonnancement calculé. Pour ce faire, le concepteur doit intervenir pour enrichir le méta-modèle MU_VARCH, à l'aide de code java généré depuis EMF, pour y intégrer les parties manquantes spécifiques au contrôleur. Le fichier obtenu est toujours visualisé avec Sirius.

Notons que toutes les figures représentant les modèles graphiques exposées dans la section suivante ont été directement réalisées sous Sirius (même si leur aspect ressemble à du *PowerPoint* ou *LibreOffice*). Il est intéressant de considérer qu'un outil permet donc d'allier les aspects documentations et présentation graphique avec une "vraie" structuration des objets "modèles".

4.3 Présentation de l'architecture *big.LITTLE*

big.LITTLE est une architecture octo-cœur (figure 4.2) annoncée par la société ARM³ au début de l'année 2013. Elle est couramment employée pour de nombreuses plate-formes de processeurs pour smartphones, comme la plate-forme Exynos de Samsung. C'est surtout un exemple d'architecture hétérogène conçu pour diminuer la consommation d'énergie des appareils mobiles. *big.LITTLE* est constitué de deux *clusters* (groupe de processeurs) hétérogènes : le premier, avec des *big* cœurs, est optimisé pour les performances ; le second, avec des *LITTLE* cœurs, vise à limiter la consommation énergétique. Ainsi, un *big* cœur expose une consommation d'énergie élevée avec une haute performance, tandis que le *LITTLE* cœur fait le comportement inverse. Le *LITTLE* coeur est utilisé la majorité du temps, pour toutes les tâches peu consommatrices en puissance de calcul, en particulier celles qui fonctionnent en arrière plan, telles que la messagerie, la musique et les appels téléphoniques. Si un besoin en performances important apparaît, le *big* coeur prend en charge les calculs, par exemple pour un jeu vidéo, ou pour le multimédia.

Les deux clusters partagent non seulement le même jeu d'instructions (ISA Instruction-Set Architecture), mais encore un bus d'interconnexion spécialement conçu pour le transfert de données entre ces deux clusters [75]. Ainsi, il est pratique pour une tâche dans un

3. ARM : Advanced RISC Machines <https://www.arm.com/>

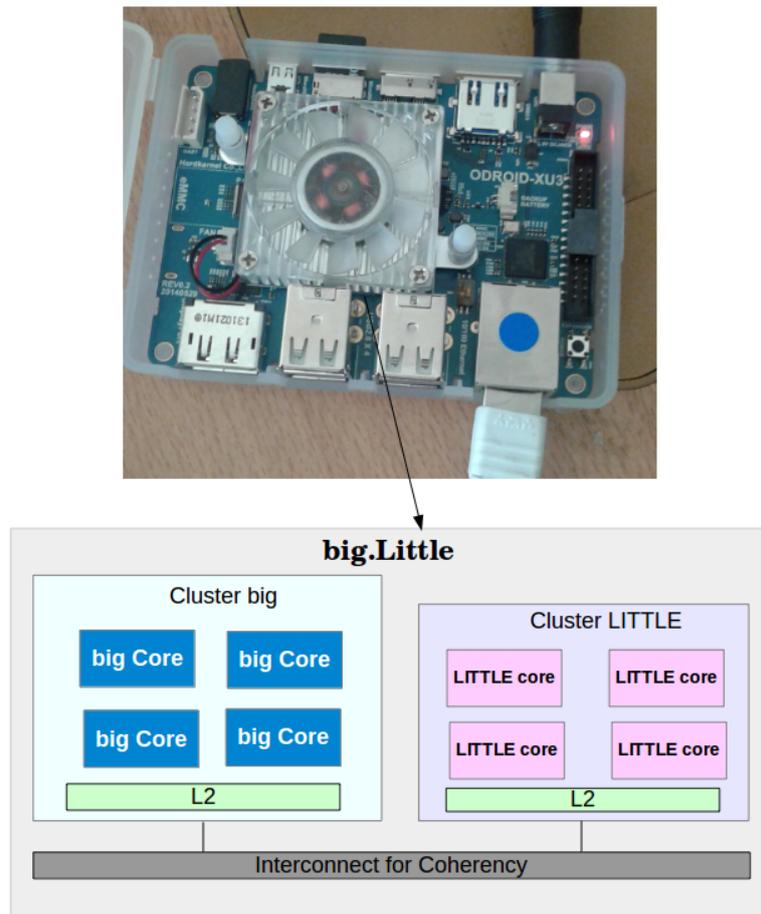


FIGURE 4.2: ODROID-XU3 équipé d'un processeur *big.LITTLE*

cluster de migrer vers un autre cluster au milieu de l'exécution, ce qui ne peut pas être réalisé dans la plupart des multi-cœurs architectures existantes.

L'architecture *big.LITTLE* prévoit le pilotage dynamique de la fréquence et de la tension (DVFS) par cluster. Pour mesurer les couples fréquence-voltage des clusters, nous avons utilisé un ODROID-XU3 board [76] équipé d'un big.LITTLE processeur de type Samsung Exynos-5422 : 4 cœurs Cortex-A15 (big) et 4 cœurs Cortex-A7 (LITTLE). (figure 4.2) Le ODROID est équipé de capteurs pour mesurer la consommation d'énergie des clusters big et LITTLE individuellement. Tous les couples fréquence/voltage (ou OPP Operating Performance Point) des deux clusters mesurées sont présentés dans la table 4.1. Notons que l'architecture *big.LITTLE* supporte uniquement un point de fonctionnement ou couple voltage/fréquence (niveau DVFS) par cluster ; c'est à dire qu'il est permis d'appliquer différents couples de voltage/fréquence pour les clusters big et LITTLE ; par

contre les coeurs dans le même cluster fonctionnent avec les mêmes voltage/fréquence.

TABLE 4.1: Niveaux Voltage/Fréquence pour les coeurs big et LITTLE

| big | | | | | | |
|-----------------|------|------|------|------|------|------|
| Voltage (V) | 0.9 | 1 | 1 | 1.1 | 1.1 | 1.2 |
| Fréquence (MHz) | 1100 | 1200 | 1500 | 1600 | 1800 | 1900 |
| LITTLE | | | | | | |
| Voltage (V) | 0.9 | 1 | 1 | 1.1 | 1.1 | 1.2 |
| Fréquence (MHz) | 700 | 800 | 900 | 1000 | 1100 | 1300 |

On peut aisément comprendre que l’architecture *big.LITTLE* pose des défis (voire des casses-têtes) de grande ampleur aux programmeurs pour sa bonne utilisation au niveau du placement et de l’ordonnancement des tâches. On pourra à ce sujet suivre sur [77] un historique des solutions définies pour Linaro/Android dans le cadre de la fondation Linux. On voit que dans une première génération seul le processeur big ou le processeur LITTLE pouvait être allumé, avec une migration de toutes les tâches entre coeurs correspondants de l’un à l’autre si nécessaire. Dans un second temps, cette approche était raffinée et déclinée par coeur (le coeur i , pour $i = 0, 1, 2$ ou 3 , est actif seulement dans la version big ou la version LITTLE, mais pas les deux à la fois, avec encore une fois migration si besoin). Actuellement la solution la plus souple (tous les coeurs disponibles pour le meilleur mapping (dynamique) désiré), dénommée *Energy-Aware Scheduling*, est toujours à l’état de projet en développement [78]. La complexité du problème vient de ce qu’on cherche à trouver un mapping des tâches au mieux possible sur les coeurs individuels, mais que ces coeurs sont liés entre eux par des relations particulières qui font que tous les coeurs d’un même bloc (big ou LITTLE) doivent fonctionner sur la même OPP, et que si un des blocs (big ou LITTLE) est entièrement libre alors on peut l’éteindre, ce qui procure un gain en “*static leakage*”.

Notre objectif dans ce chapitre consistera donc à prendre ce type d’architecture *big.LITTLE* comme exemple pour notre approche de modélisation MU_VARCH, de montrer comment nous pouvons la représenter, et de considérer comment les contraintes décrites au paragraphe précédent se modélisent de manière appropriée dans ce cadre.

Dans ce qui suit, nous partons de la modélisation du modèle d’architecture de base (ce que nous avons appelé précédemment le *Backbone View*). Nous modélisons ensuite les trois vues non-fonctionnelle : performance, power et thermique en divisant l’architecture respectivement en domaines d’horloges, domaines de voltage et domaines thermiques. Ensuite, nous modélisons l’application (un graphe de deux tâches indépendantes dans notre cas). Cette modélisation nous autorise la traduction vers des méthodes d’optimisation pour le calcul effectif des fonctions d’allocations (ordonnancement et mapping). Ces fonctions d’allocation seront ajoutées après à nos modèles.

4.4 Modélisation de notre cas d’étude avec MuVArch

Dans le chapitre précédent, nous avons exposé les aspects de méta-modélisation de notre langage MUARCH. Cette modélisation est basé sur le framework EMF et réalisé au sein de la plateforme d’Eclipse (étape 1 dans la figure 4.1). Maintenant pour la modélisation de cas d’étude (étape 2 dans la figure 4.1), nous avons employé l’outil Sirius. Pour la réalisation de cette étape, nous avons travaillé dans une instance d’Eclipse qui charge les plugins que nous avons générés précédemment du méta-modèle, dans le but de pouvoir créer un exemple de notre modèle.

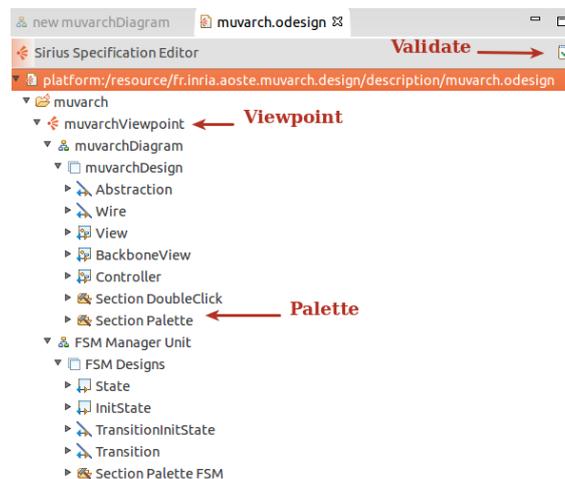


FIGURE 4.3: Fichier “.odesign” de spécification de notre éditeur

La création des éditeurs avec Sirius se réalise au travers des projets nommés “*Viewpoint Specification Project*”. Un fichier d’extension “.*odesign*” est créé et contient la spécification de notre éditeur et c’est l’unique fichier que nous avons modifié. La figure 4.3 expose notre exemple de ce fichier. Le “Viewpoint” sert à définir une manière de représenter un modèle. La “Palette” sert à créer et ajouter des éléments à notre modèle.

La figure 4.4 montre un extrait de notre éditeur créé avec Sirius. A gauche, nous visualisons un extrait de modèle conçu et édité avec Sirius. Dans cette vue, les détails des unités des management (les FSMs) ne sont pas visibles pour ne pas encombrer la figure. Pour afficher les détails de ces FSMs, il suffit de double cliquer sur chaque unité ; une nouvelle fenêtre s’ouvre avec ces détails. Dans la figure 4.4, nous visualisons également à droite un extrait de notre palette d’outils personnalisée facilitant la création des éléments de notre diagramme. Dans ce qui suit, nous détaillons les éléments du modèle affiché dans cette figure.

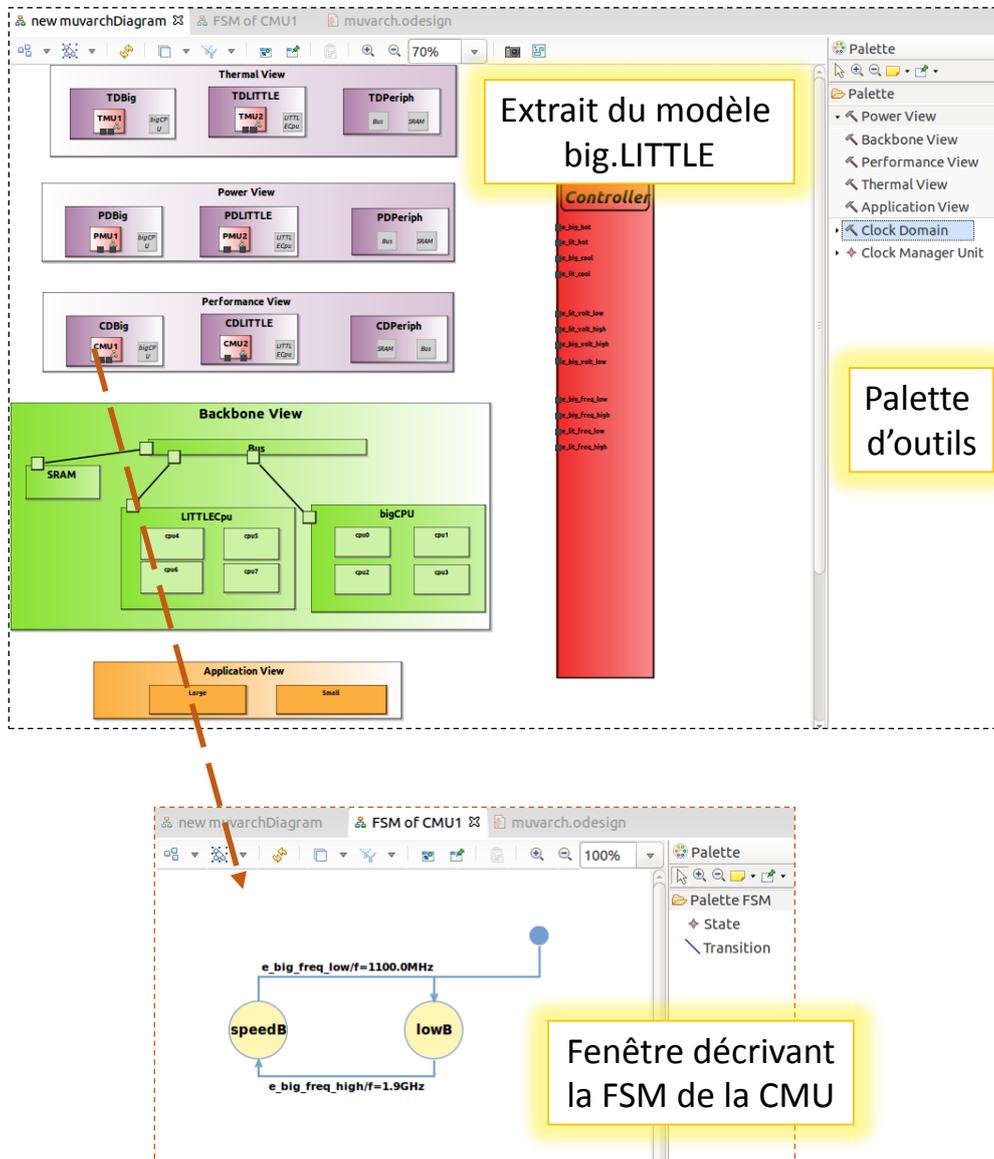


FIGURE 4.4: Extrait de l'éditeur réalisé avec Sirius

4.4.1 Modélisation de la Porte Vues (*Backbone View*)

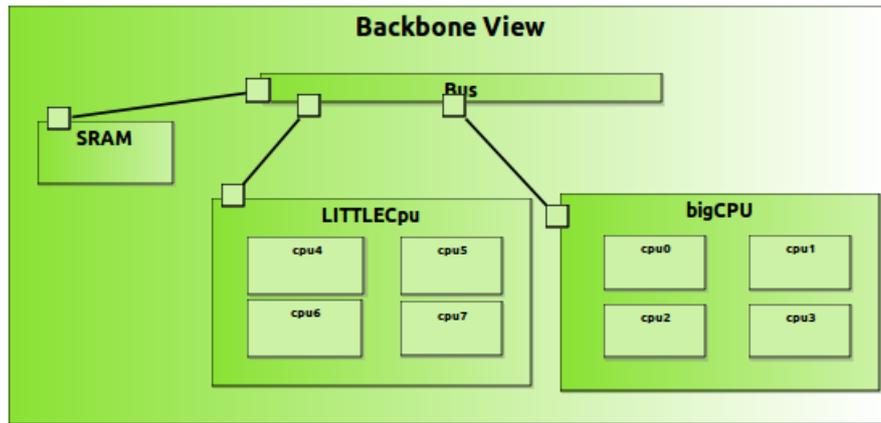
Dans MUVARCH, l'architecture logique permettant la construction de toutes les vues du système doit être définie en premier lieu. Pour notre cas d'étude, notre architecture de base (dénommée *BackboneView* dans MUVARCH) est constituée d'un cœur big, un cœur LITTLE, une mémoire SRAM; tous interconnectés par un Bus. Comme présenté précédemment, le niveau de performance du cœur big est bien plus élevée que le coeur LITTLE mais nécessite également plus d'énergie. Nous avons choisi deux niveaux d'OPPs pour chaque cœur (niveaux extraits de la table 4.1) : un niveau élevé (*high*) et un niveau bas (*low*) présentés dans la table 4.2.

TABLE 4.2: OPPs des cœurs big et LITTLE employés dans notre exemple

| big | | | | LITTLE | | | |
|------------|-----|------|-----|---------------|-----|-----|-----|
| high | | low | | high | | low | |
| f | V | f | V | f | V | f | V |
| 1900 | 1.2 | 1100 | 0.9 | 1100 | 0.9 | 700 | 0.9 |

La figure 4.5 présente le modèle de *Backbone View* de notre architecture exemple ; réalisé par notre éditeur Sirius. Ce modèle est conforme au méta-modèle de la figure 3.3. Dans ce modèle, nous définissons que la représentation logique du système. Nous avons ainsi 3 blocs (bigCPU, LITTLECPU, et SRAM) connectés par un *interconnect* (bus). Chaque CPU dispose de deux niveaux de fonctionnement (*low* et *high*). Ces composants représentent par la suite la référence des domaines de performance, de voltage et de thermique qui seront définis dans les vues non fonctionnelles. En plus, ils seront les cibles de mapping de nos tâches définis dans la vue d'application.

Au niveau comportementale, le modèle *big.LITTLE* envisagera l'existence d'un contrôleur pour allouer et modifier les paramètres des autres vues, notamment les niveaux de fonctionnement (OPPs). Ce contrôleur est enrichi suite aux résultats collectés du solveur SMT qui fera le calcul de ces allocations et modifications des paramètres au cours du temps. Nous exposons le modèle du contrôleur dans la section 4.4.6, ensuite nous traitons le contrôleur enrichi avec un ordonnancement calculé en détails dans la section 4.5.3.

FIGURE 4.5: Modèle de la vue architecturale de *big.LITTLE*

4.4.2 Modélisation de la vue Performance

Comme présenté précédemment dans le méta-modèle de la figure 3.6, la vue performance est une vue basée principalement sur des domaines de fréquences, où chaque domaine englobe un nombre de blocs du système logiciel. Dans notre exemple, la vue de performance (figure 4.6) contient trois domaines d’horloges (*CD Clock Domains*) : *CDBig*, *CDLITTLE* et *CDPeriph* incluant respectivement les ressources *BigCPU*, *LITTLECPU* et le couple (*SRAM*, *Bus*). Ces ressources sont masquées en gris dans la figure 4.6 pour montrer qu’il s’agit ici uniquement de leur abstraction (ou encore importation) depuis la vue de base et non pas de nouvelles instances.

Chaque domaine d’horloge dans la vue performance est équipé d’un CMU (*Clock Management Unit*). Dans l’exemple, nous nous intéressons uniquement aux unités de management associés à *CDBig* et *CDLITTLE* : *CMU1* et *CMU2* respectivement. Ces deux CMUs sont représentés par des FSMs (dont nous avons détaillé la définition dans la section 3.4.1). Les deux cœurs d’exécution big et LITTLE disposent de deux niveaux de fonctionnement OPP, c’est à dire de deux couples (fréquence,voltage). Pour modéliser ce changement de niveau, nous spécifions deux états *speedB* et *lowB* pour le *CDBig* (respectivement *speedL* et *lowL* pour le *CDLITTLE*). La transition entre ces états est sensible aux événements reçus du contrôleur général. A chaque fois le CMU est notifié par un événement du contrôleur, son état actuel change de *speed* vers *low* ou l’inverse.

Ce changement d'état provoque une exécution d'une action. Cette action consiste en un changement de fréquence dans la table d'OPP.

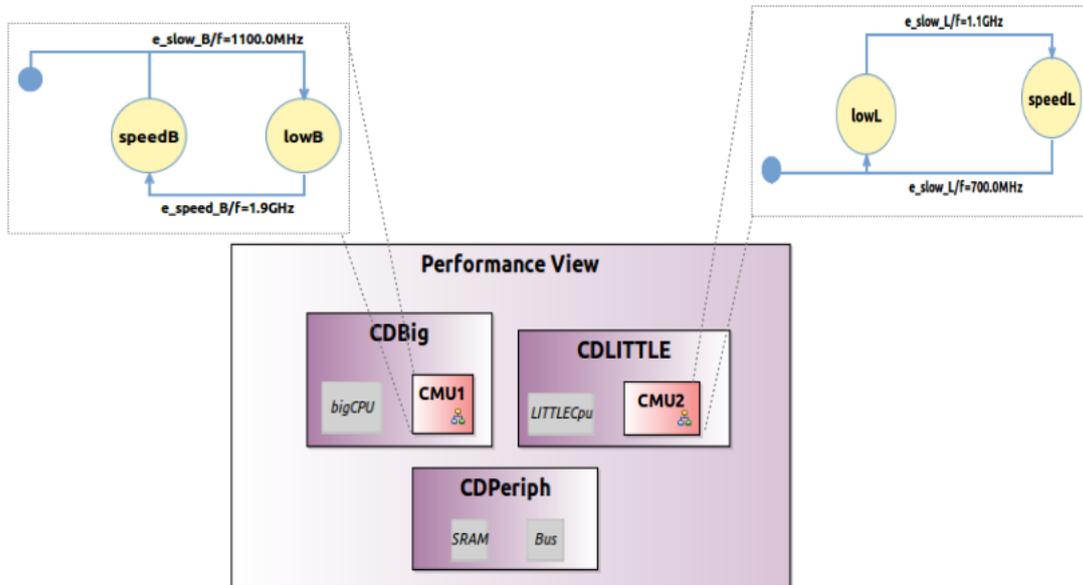


FIGURE 4.6: Modèle de la vue Performance

4.4.3 Modélisation de la vue Puissance

La figure 4.7 présente le modèle de la vue power de notre exemple. Ce modèle est une instanciation du méta-modèle de la figure 3.7. Nous l'avons divisé en trois domaines de puissance (ou domaines de voltage) : *PDBig*, *PDLITTLE*, et *PDPPeriph* encapsulant chacun les blocs big, LITTLE et le couple (SRAM, Bus) respectivement.

Comme les domaines de fréquence, chaque domaine de puissance contient une unité de management PMU (*Power Management Unit*), spécifiés ici par PMU1, PMU2 pour le PBig et le PDLITTLE respectivement. Chaque PMU est décrite par une FSM contenant deux états *low* pour un voltage bas et *high* pour un voltage élevé. Ces deux états représentent les deux niveaux de voltage dans les tables d'OPP. Ainsi, à chaque changement d'états résulte un changement de voltage. Comme les CMUs, le changement des états de

la PMU est sensible aux événements envoyés par le contrôleur. Notons qu'un changement de fréquence engendre un changement de voltage (suivant la technique de DVFS [43]).

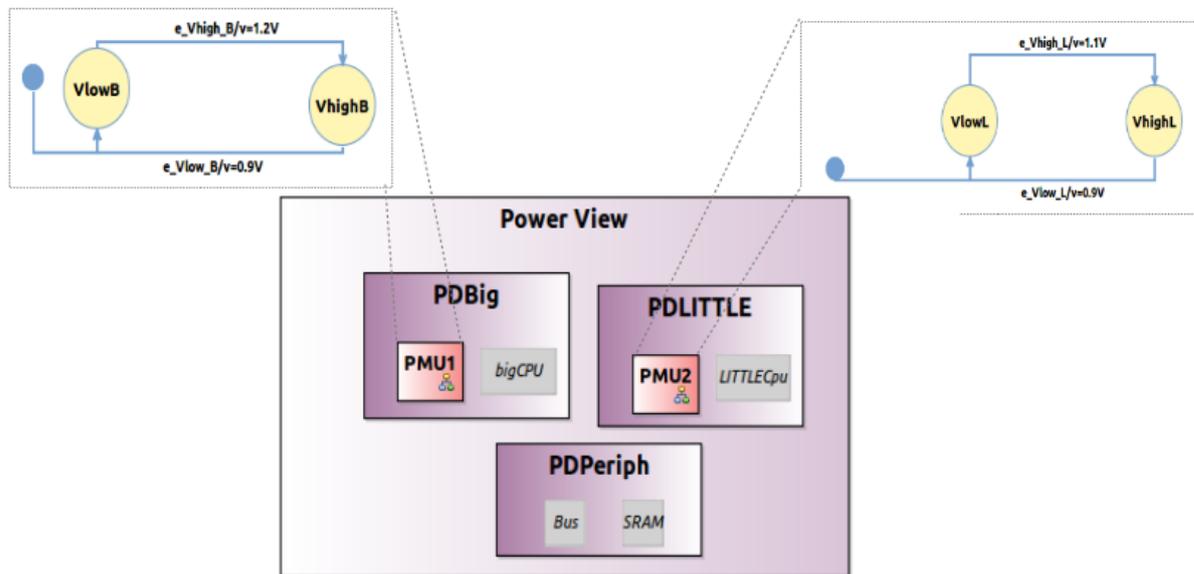


FIGURE 4.7: Modèle de la vue Puissance

4.4.4 Modélisation de la vue Thermique

La vue thermique représente les aspects thermiques de l'architecture *big.LITTLE*. Nous avons présenté les aspects de son méta-modèle dans la figure 3.8. Dans cette partie nous lui instancions un modèle conforme. Dans notre exemple, cette vue contient 3 domaines thermiques *TDBig*, *TDLITTLE* et *TDPeriph*. Ce sont des abstractions thermiques des composants big, LITTLE et le couple (bus, SRAM) définis dans la *BackboneView*. Le comportement thermique TMU (*Thermal Management Unit*) de chaque domaine thermique est décrit par une FSM. Contrairement aux CMU et PMU définis respectivement dans la vue de performance et la vue power, TMU contrôle la valeur de la température et envoie des notifications vers le contrôleur général pour régler le couple (fréquence, voltage).

Le TMU de *TDBig* (respectivement pour le *TDLITTLE*) contient une machine à deux états : *hot* pour représenter que la température de cpu augmente et *cool* pour représenter que la température de cpu diminue. Contrairement aux transitions des CMU et PMU (définis respectivement dans la vue performance et la vue power), les transitions entre les états contiennent des gardes, où la température du cpu (big ou LITTLE) est évaluée

afin de déclencher une transition et changer l'état. A chaque fois une garde est vraie, un événement est envoyé au contrôleur général pour régler le couple (fréquence, voltage). Pour évaluer la valeur de la température, les états de la TMU sont associés à des équations thermiques.

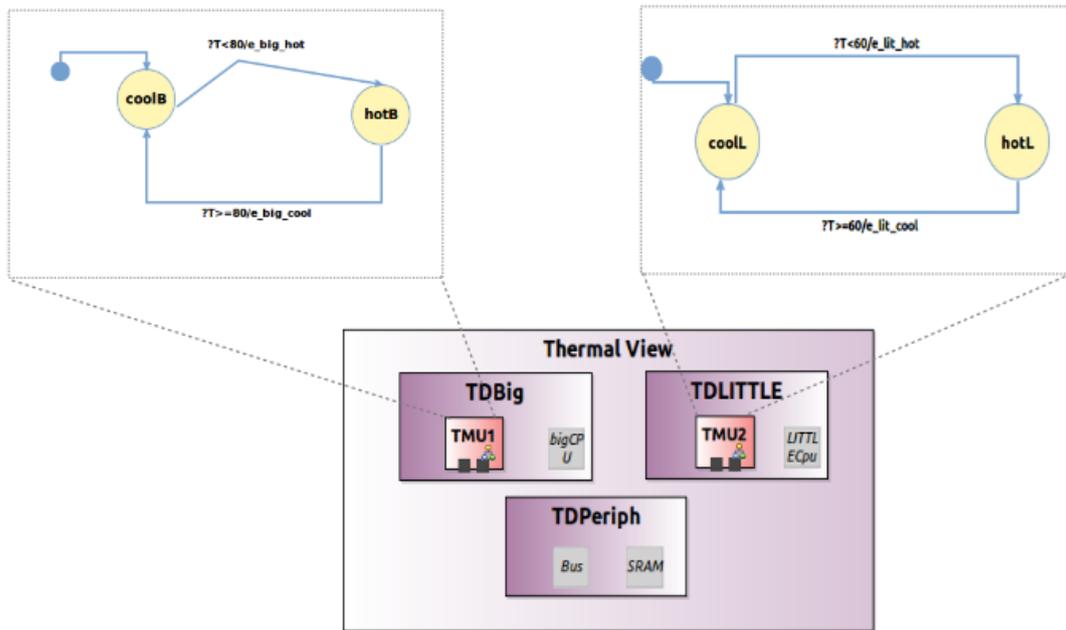


FIGURE 4.8: Modèle de la vue Thermique

4.4.5 Modélisation de la vue Application

Dans notre cas d'étude, nous avons choisi comme scénario d'exécution un graphe de deux tâches *Large* et *Small* périodiques. La figure 4.9 montre le modèle d'application. A ce niveau, le modèle contient que les deux tâches qui seront instanciés dans un second temps. La tâche *Large* est plus coûteuse et demande plus de performance que la tâche *Small*. Les tâches sont caractérisées par le couple $(P_i$ et $C_{i,r}$), où P_i est la période d'exécution de la tâche i et le $C_{i,r}$ est le coût d'exécution de la tâche i sur une ressource r (table 4.3). Les périodes de ces deux tâches sont équivalents à leurs échéances. Nous supposons que les tâches sont indépendantes, c'est à dire qu'elles n'ont pas de dépendances de données et ne partagent aucune ressources sauf les cœurs big et LITTLE de notre architecture.

TABLE 4.3: Coûts des tâches en Méga-Cycles

| | Large | Small |
|--------|-------|-------|
| big | 25 | 50 |
| LITTLE | 10 | 15 |

Une tâche générique i donne naissance à un ensemble d'instances. Chaque instance de la tâche T_i doit s'exécuter entièrement à l'intérieur de la période P_i . Une instance ne peut pas être exécutée en parallèle sur plusieurs cœurs. Comme soutenu par l'architecture *big.LITTLE*, une instance de tâche peut migrer d'un cœur big vers un cœur LITTLE (ou d'un cœur LITTLE vers un cœur big). Dans la figure 4.9, nous n'avons que les tâches génériques *Large* et *Small*. Leurs instances ainsi que leur AMUs (*Application Manager Units*) seront générés lors de l'ordonnancement ; ce que nous détaillons dans la section 4.5.3.

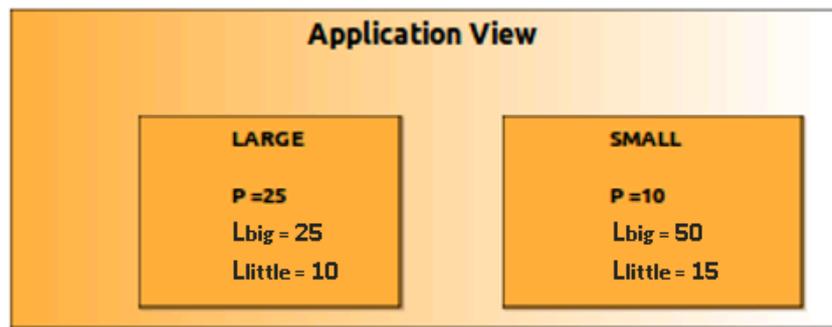


FIGURE 4.9: Modèle de la vue d'application

4.4.6 Prise en compte du contrôleur général en Sirius

A ce niveau de modélisation, nous avons uniquement besoin de décrire le contrôleur général, en Sirius, comme un composant (vide de comportement) avec son interface de ports pour le lier aux autres vues, comme visualisée dans la figure 4.10. Plus tard, ce modèle devra contenir une modélisation (par notre modèle d'allocation) de la trace instrumentée calculée par un ordonnancement/placement des tâches à l'aide d'un solveur de contraintes externes.

La figure 4.10 expose le modèle global de *big.LITTLE* crée et édité avec Sirius. Nous visualisons les parties structurelles de la porte vues (BackboneView) ainsi que les autres

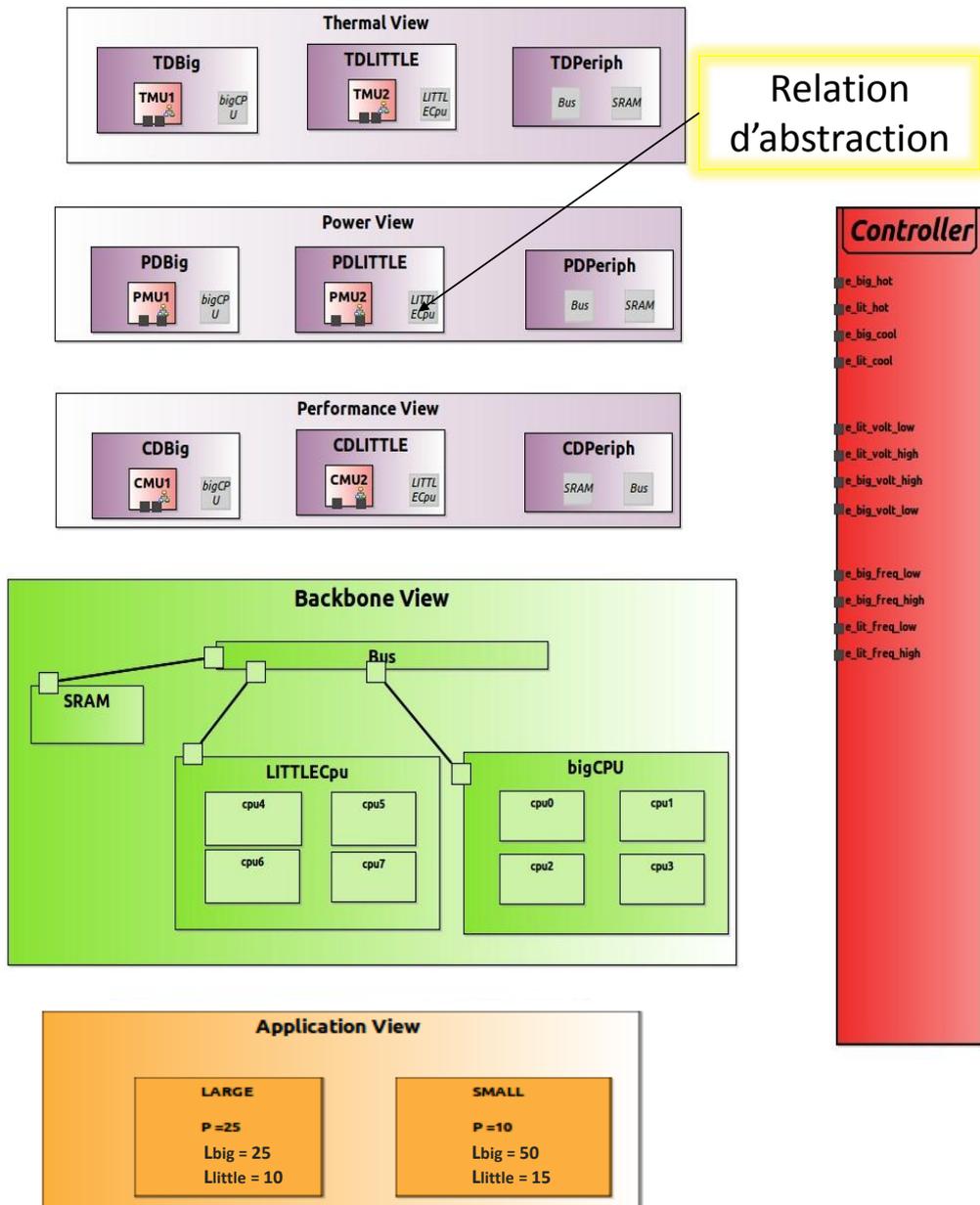


FIGURE 4.10: Modèle global de *big.LITTLE* créée et éditée avec Sirius

vues (performance, power, thermal et application) sont représentés. Également, le contrôleur définissant les événements à déclencher pour synchroniser chaque vues sont exposés. Notons qu'à ce niveau la vue d'application ne définit pas encore des AMUs. Ces derniers seront générés avec leurs instances de tâches suite à l'importation de l'ordonnancement calculé. Un AMU sera contenu pour chaque instance de tâche. Pareil pour le contrôleur qui ne dispose pas encore de FSM ; cette dernière sera aussi générée suite à la réception des allocations construites reçues du solveur SMT.

Comme nous l'avons noté dans les sections précédentes, la relation d'abstraction dans la figure 4.10 est représentée par des blocs de couleur grise.

4.5 Connexion avec un solveur SMT existant

Cette section consiste à développer les étape 3, 4 et 5 de la figure 4.1. En fait dans cette étape, nous employons tout d'abord Acceleo, l'outil de transformation de modèle à texte pour générer du code ; décrivant notre cas d'étude ; compréhensible par le solveur SMT [74]. Ce code est exécuté après par ce dernier qui nous calculera un ordonnancement adéquat de l'architecture spécifiée. Enfin, cet ordonnancement est transformé de sorte qu'il soit interprété par notre contrôleur général. Nous détaillons tous les étapes ci dessous.

4.5.1 Génération du code pour le Solver SMT

Dans cette étape, nous abordons la réalisation du génération des fichiers servant d'entrées pour le solveur tiers. Cette génération renferme des templates permettant de générer du texte à partir d'un modèle EMF. Pour ce faire, nous nous appuyons sur Acceleo [79] afin de réaliser une génération automatique à partir de modélisation haut-niveau. Fondé sur la démarche MDA, Acceleo offre un générateur de code qui permet de transformer des modèles vers du code. Cet outil est nativement intégré à Eclipse et est compatible avec les standards UML, EMF, XMI, MOF. Le principe général de fonctionnement d'Acceleo, comme indique la figure 4.1, consiste à adopter un template de génération à un modèle de départ pour obtenir en sortie un code. Un template Acceleo est un fichier

d’extension “.mtl” conçu pour configurer le code à générer sur un modèle cible. Il est constitué de deux parties : une en-tête et une liste de scripts contenant des “trous” qui seront remplies, lors de la génération, avec les données correspondantes du méta-modèle. Un exemple de template Acceleo est représenté ultérieurement par la figure 4.11.

Afin de mettre en place nos templates de génération, nous nous appuyons dans un premier temps sur notre définition du MUVARCH à savoir notre Méta-modèle MUVARCH défini dans le chapitre 3 et dans un deuxième temps sur le modèle de *big.LITTLE* décrit et modélisé plus haut. Un modèle en entrée d’Acceleo est examiné comme étant un arbre de données que l’on parcourt efficacement avec l’éditeur Acceleo, puis on spécifie le code à générer grâce à des éléments de contrôle comme les boucles. Dans notre processus et afin de connecter notre modèle au solveur SMT tiers, nous avons besoin de trois fichiers en entrée pour ce solveur afin de calculer un ordonnancement adéquat. Ces fichiers décrivent les contraintes issues de MUVARCH sous forme contextuelle :

1. un fichier d’extension *.app* contient les informations liées aux tâches de l’application, notamment la période.
2. un fichier ayant l’extension *.res* présente les caractéristiques des ressources de l’architecture : la capacitance en nF, le coût de changements de niveau de fonctionnement et les OPPs (leurs noms et valeurs fréquence-voltage).
3. Enfin, un troisième fichier d’extension *.map* contient le coût d’exécution de chaque tâche sur les ressources ; ce coût est exprimé en Macro-cycles.

Pour avoir ces trois fichiers, trois templates de générations sont requis. La figure 4.11 représente un exemple d’un de nos templates Acceleo pour la génération du fichier d’extension “.res”.

Dans ce template, nous avons ajouté un squelette de code pour un fichier décrivant les ressources. Dans ce squelette, nous parcourons les différents éléments qui composent le modèle *big.LITTLE* d’entrée et nous générons le code qui correspond à chacun de ces éléments.



```

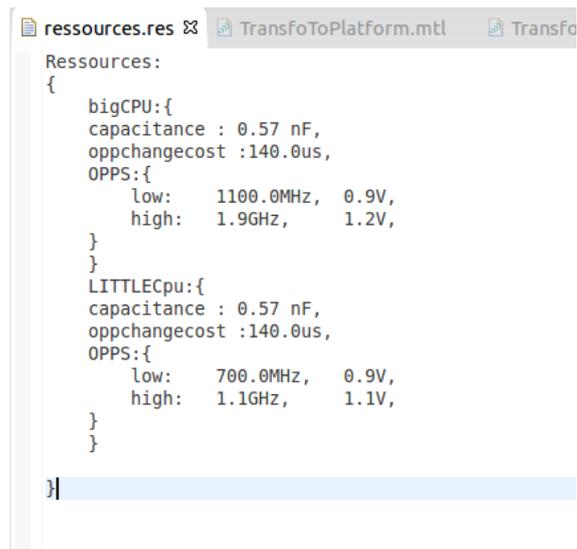
[comment encoding = UTF-8 /]
[module TransfoToPlatform('http://muvarch.aoste.inria.fr/')]

[template public generateElement(aMuVArch : MuVArch)]
[comment @main/]
[file ('ressources'.concat('.res'), false, 'UTF-8')]
Ressources:
{
  [for (res : IPBlock | aMuVArch.eContents(BackboneView).eContents(IPBlock))]
    [res.name/]:{
      capacitance : [res.capacitance/] nF,
      oppchange cost : [res.oppchange cost/]us,
      OPPS:{
        [for (opp : OPP | res.eContents(OPP))]
          [opp.name/]: [for (oppval: OPPValue | opp.eContents(OPPValue))] [oppval.constprop.value/][oppval.const
        [/for]
      }
    }
  [/for]
}
[/file]
[/template]

```

FIGURE 4.11: Template Acceleo pour la génération du fichier “res”

Une fois tous nos templates de génération sont mis au point, il s’agit de générer les fichiers correspondants. Il est toutefois possible de faire des génération massives de plusieurs codes sources correspondant à différentes modèles d’architectures, grâce aux chaînes de lancement successives qu’offre Acceleo. Dans notre cas, nous générons que les trois fichiers nécessaires pour viser le solveur SMT. La figure 4.12 montre l’exemple de fichier d’extension “res” généré. Ce fichier contient donc les caractéristiques de nos ressources *big* et *LITTLE* et les niveaux OPPs.



```

Ressources:
{
  bigCPU:{
    capacitance : 0.57 nF,
    oppchange cost :140.0us,
    OPPS:{
      low: 1100.0MHz, 0.9V,
      high: 1.9GHz, 1.2V,
    }
  }
  LITTLEcpu:{
    capacitance : 0.57 nF,
    oppchange cost :140.0us,
    OPPS:{
      low: 700.0MHz, 0.9V,
      high: 1.1GHz, 1.1V,
    }
  }
}
}

```

FIGURE 4.12: Fichier “res” généré depuis Acceleo

4.5.2 Calcul des contraintes et de l'ordonnancement

Avant le calcul de l'ordonnancement, des contraintes et des hypothèses valides sont nécessaires pour lancer ce calcul. Pour ce faire, un ensemble de transformations depuis les trois fichiers générés en dessus vers des contraintes sous forme concrète et textuelle est demandé. Pour le moment, cet ensemble de transformations est accompli dans l'approche du solveur SMT. Nous visons dans le futur proche accomplir ces transformations dans notre approche MUVARCH et ceci par la spécification des contraintes de type OCL (*Object Constraint Language*). Ainsi, à partir de MUVARCH, il sera possible de pondérer toutes ces contraintes directement vers le solveur SMT.

Supposons que pour une tâche t_{stop} , t_{start} , t_a , t_p sont respectivement son temps physique d'arrêt d'exécution, son temps physique de départ d'exécution d'une tâche, son temps d'apparence et sa *deadline*. Des exemples de contraintes soumis au solveur SMT sont les suivantes :

$$t_{stop} = t_{start} + t_p$$

$$t_{stop} \geq t_a$$

$$t_d \geq t_{stop}$$

...

Après le calcul de l'ordonnancement, le résultat en sortie du solveur SMT est retourné dans un fichier *JSON*⁴. Dans ce fichier, dont un extrait est visible dans la figure 4.13, nous avons toutes les informations de l'ordonnancement calculé de nos tâches *Small* et *Large* sur nos ressources *big* et *LITTLE* avec les niveaux des OPPs adéquats. L'étape suivante consiste donc à enrichir le modèle *big.LITTLE* par cet ordonnancement calculé.

4. <http://www.json.org/json-fr.html>, JSON est un format léger d'échange de données dérivé de la notation des objets du langage *JavaScript*.

```

{
  "SchedulingResults": [
    {
      "color": "0x9732cb",
      "label": "Large_{0}",
      "map": "bigCPU",
      "opp": "low_big",
      "precedences": [],
      "start": "2255.63909774436",
      "stop": "24982.9118250171"
    },
    {
      "color": "0x9732cb",
      "label": "Large_{1}",
      "map": "bigCPU",
      "opp": "low_big",
      "precedences": [],
      "start": "25017.0881749829",
      "stop": "47744.3609022556"
    },
    {
      "color": "0xcb3297",
      "label": "Small_{0}",
      "map": "LITTLEcpu",
      "opp": "high_little",
      "precedences": [],
      "start": "0",
      "stop": "9090.90909090909"
    },
    {
      "color": "0xcb3297",
      "label": "Small_{1}"
    }
  ]
}

```

FIGURE 4.13: Extrait du fichier JSON de l'ordonnancement calculé généré par le solveur SMT

4.5.3 Prise en compte de l'ordonnancement par le contrôleur de MuVArch

L'étape 5 représenté dans la figure 4.1 consiste à injecter le résultat sous forme de fichier JSON de l'ordonnancement calculé au contrôleur général de notre modèle *big.LITTLE* à base de diagramme Ecore. Comme nous l'avons signalé, EMF permet une migration facile entre le code Java et le modèle Ecore. L'idée donc pour connecter le résultat, obtenu sous forme de fichier JSON du solveur SMT, à notre modèle MuVARCH est de passer par le code Java généré suite à la création du méta-modèle avec EMF. Ce code Java, dont un extrait apparaît dans la figure 4.14, permet une navigation dans notre modèle MuVARCH ainsi qu'une navigation dans le fichier JSON grâce à l'API *org.json*. Dans ce code Java,

nous créons les représentations des traces parallèles d'exécutions et une instance de machine à états finis FSM pour une représentation séquentielle.

```
// JSON FILE
try {
    // read JSON file
    FileReader reader = new FileReader(fileJsonPath);

    //create parser to JSON
    JSONParser jsonParser = new JSONParser();
    JSONObject jsonObject = (JSONObject) jsonParser.parse(reader);

    // get the scheduling array from the JSON object
    JSONArray array= (JSONArray) jsonObject.get("SchedulingResults");

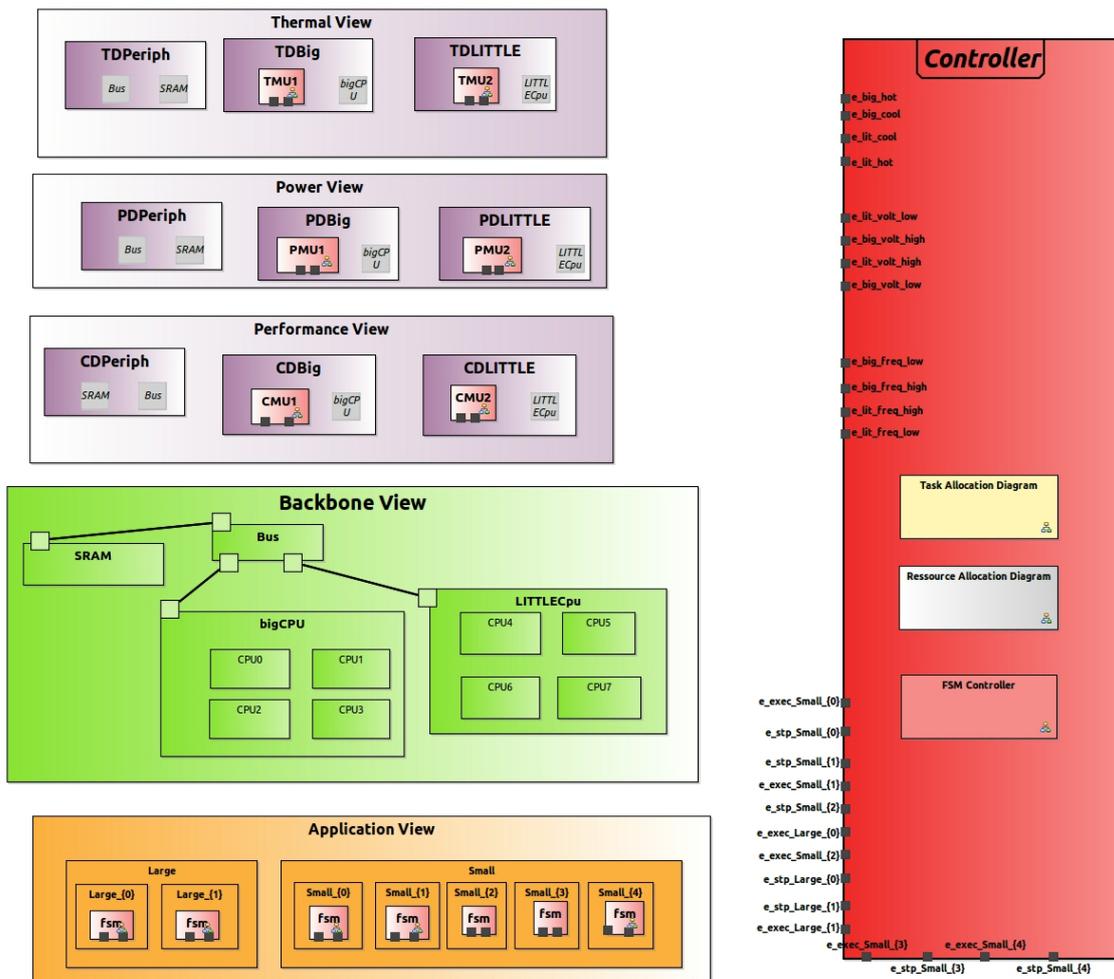
    SetTaskAllocation setTaskAll = MuVArchFactory.eINSTANCE.createSetTaskAllocation();
    cntr.getHasTaskAllocation().add(setTaskAll);
    setTaskAll.setName("Task Allocation Diagram");
    SetRessAllocation setRessAll = MuVArchFactory.eINSTANCE.createSetRessAllocation();
    cntr.getHasRessAllocation().add(setRessAll);
    setRessAll.setName("Ressource Allocation Diagram");

    for (View v : root.getViews()){
        for (StructuralElement t : v.getStructuralelems()){
            if (t instanceof Task){
                TaskAllocation taskAll = MuVArchFactory.eINSTANCE.createTaskAllocation();
```

FIGURE 4.14: Extrait de l'analyseur (*Parser*) Java JSON

Une fois ce code exécuté, un nouveau modèle MuVARCH est généré. Ce nouveau modèle est ainsi enrichi par la stratégie d'ordonnancement calculée par le solveur SMT. La figure 4.15 montre ce nouveau modèle MuVARCH généré. Désormais, le comportement du contrôleur est doté d'un ordonnancement représenté de deux manières; l'une pour l'ensemble des tâches et l'autre pour l'ensemble de ressources. Une autre représentation séquentielle est mise en place. Cette dernière est exprimée par une machine à états finis. L'ensemble de ces représentations est illustré ci dessous.

Notons aussi que les instances de tâches (ou job) produites lors de l'ordonnancement sont incluses dans ce nouveau modèle généré. Chaque instance de tâche est équipée d'une AMU (*Application Manager Unit*) servant après pour la synchronisation de l'exécution du modèle.

FIGURE 4.15: Modèle global de *big.LITTLE* créée et éditée avec Sirius

Nous avons exposé dans la section 3.6 que ce comportement peut être représenté de différentes manières que nous illustrons ici. Rappelons que toutes ces représentations sont créées et conçues dans notre atelier que nous avons implémentée et à travers l'outil Sirius précisément.

- La première est en procurant pour chacune des ressources l'ensemble des instances de tâches qui s'y exécutent. Cette représentation appliquée à notre cas d'étude est affichée dans la figure 4.16. Deux blocs d'allocation de ressources sont affichés sur la figure et qui correspondent à nos deux ressources : le *big* et le *LITTLE*. Sur chaque bloc, les instances de tâches exécutées sont affichées avec leurs dates de début et de fin.

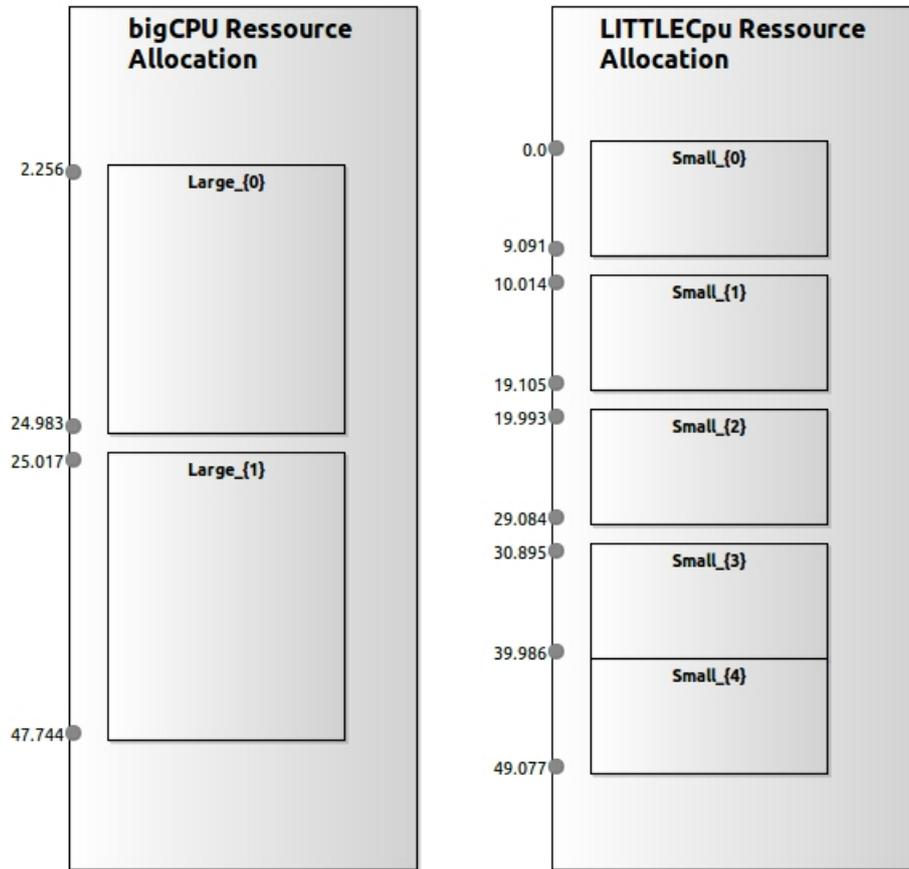


FIGURE 4.16: Représentation des allocations des ressources

- La deuxième est en procurant pour chaque tâche la suite des allocations de ces instances, les ressources allouées ainsi que le niveau d'OPP utilisé. La figure 4.17 nous montre ce type de représentation. Nous visualisons sur chaque bloc d'allocation des tâches (*Large* et *Small*), l'instance qui est entrain de s'exécuter (par exemple *Large_0*), la ressource allouée (*bigCPU*) et le niveau de fonctionnement actif (*low_big*), s'exécutant entre 2.256 ms et 24.983 ms.
- La troisième est une représentation séquentielle sous forme d'une machine à états finis FSM comme affiché dans la figure 4.18. Les transitions dans cette machine correspondent aux actions de début et fin d'exécution d'une instance de tâche. Les transitions déclenchent les événements associés avec ses actions (par exemple exécuter une tâche, arrêter une tâche, activer un niveau de voltage/fréquence,

etc). La lacune apparente de cette représentation se situe dans le fait que le nombre d'états augmente rapidement en fonction du système.

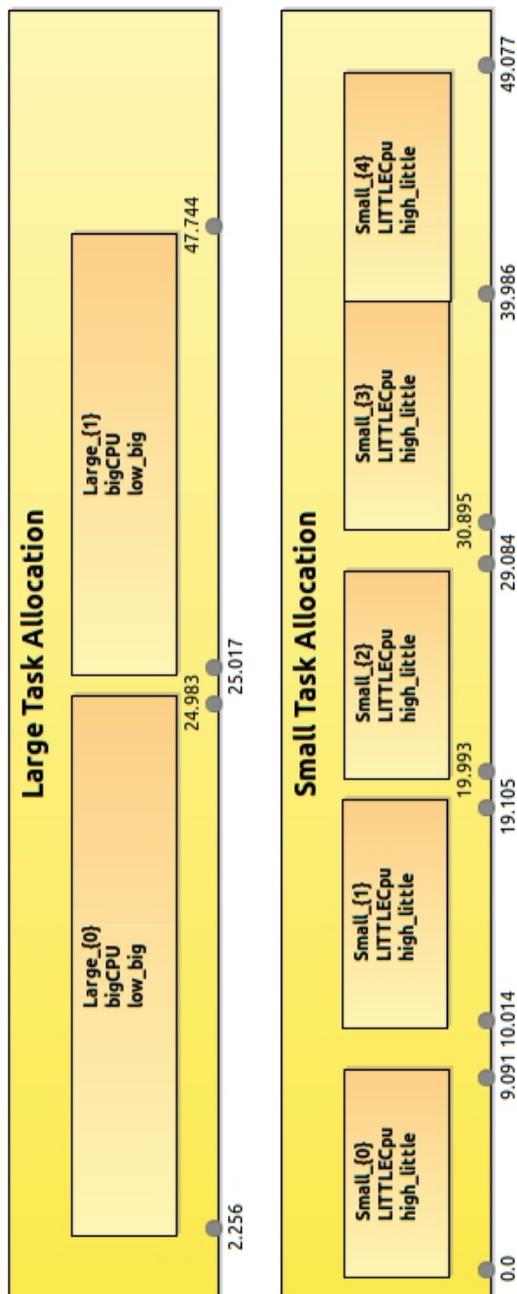


FIGURE 4.17: Représentation des allocations des tâches

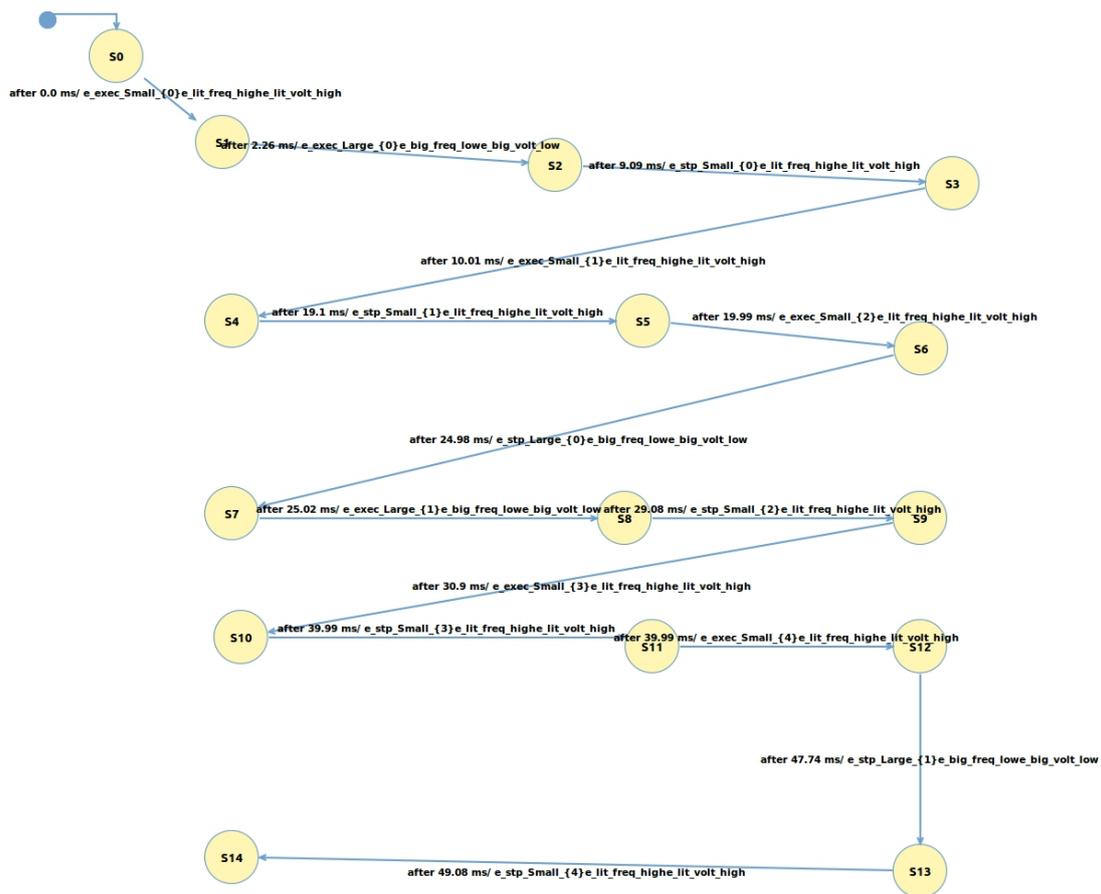


FIGURE 4.18: Représentation en FSM du contrôleur général

4.6 Conclusion

Dans ce chapitre, nous avons introduit l'implémentation de l'atelier logiciel que nous avons réalisée pour instrumenter notre approche MUVARCH. Notre but en réalisant cette implémentation était de pouvoir employer notre approche de manière pratique et concrète et expérimenter son utilisation sur différents différents modèles. Pour cela, nous nous sommes situés dans le contexte technique standard du domaine MDA et nous avons employés ainsi une chaîne d'outils de ce domaine : *EMF* pour la définition du méta-modèle, *Sirius* pour la création de modèles d'une manière graphique concrète et *Acceleo* pour la génération du code à partir de ces modèles vers un Solveur SMT tiers pouvant calculer après des ordonnancements adéquats.

Également, nous avons présenté dans ce chapitre un cas d'étude pour mettre en

oeuvre et valider notre approche MU \hat{V} ARCH. Nous nous sommes intéressés à une architecture *big.LITTLE* multi-processeurs. Nous avons donc modélisé, avec notre atelier logiciel proposé, les différentes vues du système et les relations nécessaires. En second temps, nous réalisons une chaîne de transformations pour pouvoir connecter ce modèle à un solveur SMT existant. Ce dernier nous calcule et nous renvoie l'ordonnancement adéquat, qui sera ensuite le comportement du contrôleur général du modèle. Ce comportement est représenté sous forme d'allocation globale de tâche ou d'allocation globale de ressources ou encore sous forme d'une machine à états finis.

BILAN ET PERSPECTIVES

Sommaire

| | | |
|---------------------|--|-----|
| 5.1 | Bilan | 98 |
| 5.2 | Perspectives | 100 |

5.1 Bilan

Les travaux présentés dans ce manuscrit s’inscrivent dans le cadre du projet ANR HOPE. Il s’agit ici d’apporter essentiellement une contribution concernant une méthodologie de modélisation multi-vues des systèmes hétérogènes embarqués en se basant sur les modèles. Notre méthodologie s’appuie fortement sur le standard OMG : l’Ingénierie Dirigée par les Modèles (IDM). La figure 5.1 illustre nos contributions (les cadres en jaune et vert).

Nous proposons une approche, appelée MUVARCH, dont l’objectif est de permettre de modéliser le système de différents points de vues et conserver la cohérence entre ces vues. Précisément, MUVARCH se base sur une vue (support) décrivant le modèle architecturale. A cette base, des vues d’autres domaines (performance, puissance, thermique et fonctionnel) sont “raccrochées”. MUVARCH assure la cohérence d’interconnexion entre toutes ces vues avec des relations d’abstraction (entre la vue de base et les vues de propriétés non fonctionnelles) et d’association (entre les propriétés des vues). Une autre relation plus spéciale est présente : la relation d’allocation (entre les tâches de l’application et

les ressources de l'architecture). Dans MUVARCH, cette relation s'avère plus complexe qu'une simple flèche connectant la tâche à la ressource. Elle incarne plutôt toute une algorithmique d'ordonnancement et de placement de tâches. Une fois ce calcul fait, MUVARCH permet la représentation de cette relation d'allocation où les divers instances de tâches peuvent se voir allouer sur plus qu'une ressource, ainsi que les valeurs des vues non-fonctionnelles (tels que le voltage et la fréquence) peuvent varier selon les instances de tâches.

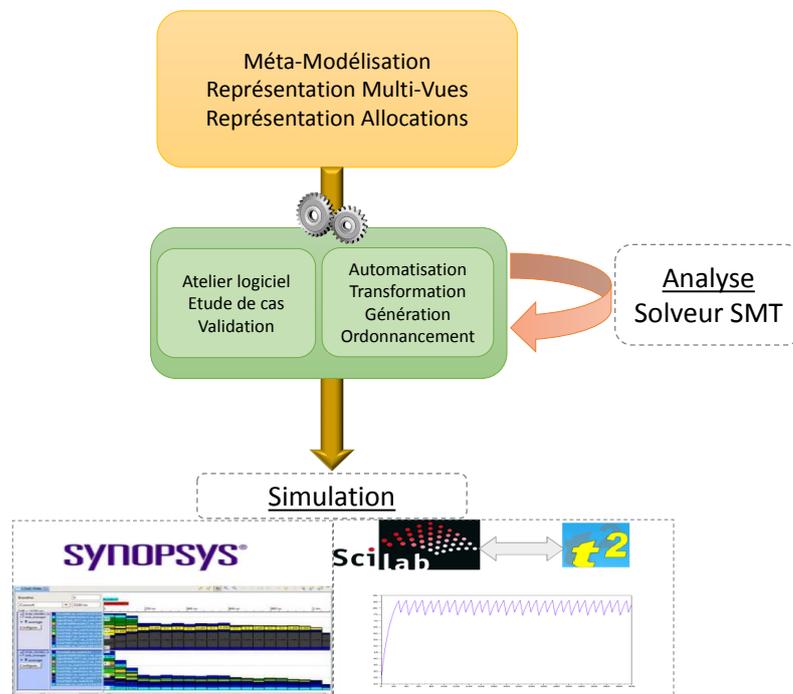


FIGURE 5.1: Illustration des différentes contributions de la thèse

Notre approche est implémentée dans un framework (ou atelier logiciel) appelé MUVARCH. Ce framework a été conçu pour s'appuyer sur le *Eclipse Modeling Framework* (EMF) afin de bénéficier de l'outillage d'ingénierie des modèles existant. Également, cet atelier emploie *Sirius*, un outil permettant une modélisation personnalisée et graphique.

Pour valider notre approche, nous avons étudié un cas d'étude portant sur une architecture multi-processeur nommée *big.LITTLE*. Cette architecture hétérogène conçue spécialement pour diminuer la consommation d'énergie des appareils mobiles. La modélisation de cette architecture avec MUVARCH ne contenait pas une stratégie d'ordonnancement en premier temps. Cela nécessitait une connexion avec un solveur SMT tiers existant. Cette connexion est réalisée via des générations de code de MUVARCH vers le solveur en employant le générateur de code *Acceleo*. Le solveur à son tour employait les informations reçues de MUVARCH et calcule un ordonnancement adéquat des tâches de l'application décrite vers les ressources de l'architecture. Après avoir reçu et lu l'ordonnancement calculé, le contrôleur général du modèle est capable de représenter les allocations calculées.

5.2 Perspectives

Les travaux que nous présentons dans ce mémoire ouvrent des perspectives pour le domaine de la modélisation multi-vues, pour la modélisation et représentation relations d'allocation que nous proposons et pour la co-simulation pour évaluer plus finement les différents aspects non-fonctionnelles sur nos modèles.

- Comme perspectives à court termes, nous viserons calculer les contraintes à soumettre dans le solveur SMT. Ces contraintes seront traduites de notre approche MUVARCH et ceci en s'appuyant sur des contraintes OCL.
- Dans MUVARCH, nous décrivons la représentation de la stratégie d'ordonnancement et de placement calculé par un solveur tiers. Cette relation montre particulièrement l'allocation des instances de tâches applicatives vers les ressources architecturales. Dans un objectif d'amélioration de l'utilisabilité de notre approche, il nous paraît intéressant d'envisager d'autres types d'allocation de communications et de stockage de données.
- Notre approche traitant de l'hétérogénéité des domaines, il serait intéressant d'envisager de l'exécuter. Dans cette optique, nous avons réalisé des travaux

anciens pour l'analyse thermique sur une version antérieure (présentés dans l'annexe 5). Comme travaux futurs, nous projeterons ces anciens travaux sur notre approche actuelle MUVARCH en l'enrichissant avec une sémantique d'exécution permettant l'analyse des propriétés non-fonctionnelles, éventuellement la température.

- L'un des axes de recherche que nous envisageons également concerne la simulation avec des outils tiers d'analyse industriels tels que *DoceaPower* et *Synopsys*. Cela permettra de reporter les résultats de l'ordonnancement et de placement pour les exploiter en simulation et évaluer plus finement les aspects non fonctionnelles.

EXÉCUTION DES MODÈLES HÉTÉROGÈNES POUR DES ANALYSES THERMIQUE

Sommaire

| | |
|--|------------|
| A.1 Contexte des analyses | 103 |
| A.2 Co-Simulation conjointe | 104 |
| A.3 Étude de cas : Contrôleur thermique d'un processeur | 107 |

Le chapitre ci-dessous représente une version plus ancienne de nos travaux appliqués sur l'approche PRISMSYS [5] (dont on hérite MUARCH), sur laquelle nous avons conduit des études associant la consommation énergétique et le bilan thermique, en utilisant l'outil *Scilab* pour la modélisation (continue) de la température. Faute de temps nous n'avons pas pu "porter" ces travaux sous l'environnement MUARCH, alors que la modélisation conjointe power/température était (et reste) un de nos objectifs. Nous décrivons en fin de chapitre la nature du travail restant à réaliser dans cette perspective.

A.1 Contexte des analyses

PRISMSYS est une approche multi-vue dirigée par les modèles, basée sur un profil UML qui s'appuie autant que possible sur les profils SYSML et MARTE. Le modèle sémantique qui maintient la cohérence est spécifié avec le langage CCL. La figure A.1 montre les éléments principaux de PRISMSYS. Une vue dans PRISMSYS se base essentiellement de trois sous vues : une sous vue structurelle (*StructuralSubView*), une sous vue comportementale (*BehavioralSubView*) et une sous vue équationnelle (*EquationalSubView*).

La sous vue structurelle définit les concepts et les relations des éléments dans un domaine spécifique. Ces éléments définissent les propriétés non fonctionnelles pertinentes dans ce domaine. Le comportement de ces éléments est décrit par une machine à états finis FSM. La FSM spécifie les modes opérationnels des éléments et les transitions sont sensibles aux événements générés par la sous vue comportementale. Des composants UML sont utilisés pour représenter les éléments et les propriétés non fonctionnelles NFP de MARTE pour typer les propriétés définis dans cette sous vue.

La sous vue équationnelle spécifie les équations qui caractérisent les propriétés identifiées dans la sous vue structurelle. SYSML *Parametric Diagram* est employé pour représenter cette sous vue.

La sous vue comportementale commande l'exécution des éléments spécifiés dans la sous vue structurelle en se basant également sur des FSMs. Contrairement à la FSM de la sous vue structurelle, les transitions du FSM du contrôleur sont sensibles à l'évaluation d'une condition (garde). Pour différencier les deux FSMs, le terme mode-FSM est employé pour spécifier la FSM de la sous vue structurelle.

La cohérence entre les vues est définie à travers des relations de correspondance. Le relation de caractérisation (figure A.1) par exemple, associe une équation à un mode dans la mode-FSM. Toutes les correspondances ainsi que les éléments de l'approche PRISMSYS sont détaillés de la thèse [5] de *Carlos Ernesto Gómez Cárdenas*, ancien doctorant dans l'équipe AOSTE.

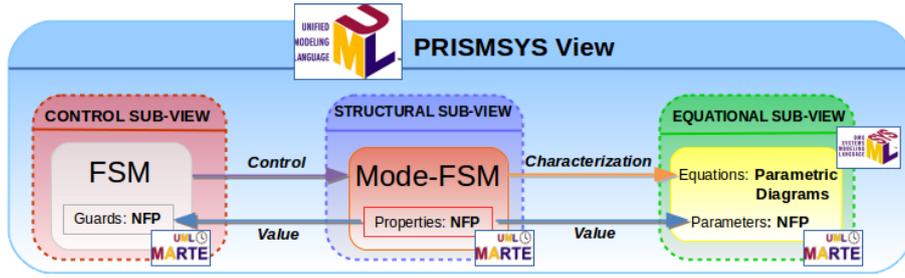


FIGURE A.1: Vue dans PRISMSYS

La spécification du modèle de PRISMSYS est complétée par la définition de la sémantique d'exécution. Cette sémantique permet l'analyse des propriétés non-fonctionnelles en fonction de temps définis dans le modèle. Cette analyse est possible une fois que le modèle est simulé et les propriétés sont évaluées en fonction de temps. La sémantique d'exécution de la sous vue de contrôle *BehavioralSubView* utilise CCSL (*Clock Constraint Specification Language*) [80]; un langage qui permet la modélisation de contraintes synchrones et/ou asynchrones sur des couples d'horloges. Ainsi, *BehavioralSubView* exprime le scénario d'exécution des différentes vues de PRISMSYS. Une fois la sémantique est définie, le modèle est simulé et analysé dans l'environnement TIMESQUARE [4], spécialement développé pour la modélisation et l'analyse avec CCSL. Néanmoins, cet outil n'est pas adapté pour évaluer et calculer les équation; telles que les équations de puissance et de température). Notre but était donc de pouvoir co-simuler le temps discret (logique), spécifié dans les FSMs et exécuté par CCSL, avec le temps continu des équations mathématiques. Dans nos travaux, nous choisissons *Scilab* [81], un logiciel open source gratuit de calcul numérique qui fournit un puissant environnement de développement pour les applications scientifiques et l'ingénierie. Nous développons ainsi, un "connecteur" entre TIMESQUARE et *Scilab* pour évaluer les équations actives. Nous nommons ce solveur T2 BACKEND FOR SCILAB, que nous abordons son implémentation le long de ce chapitre.

A.2 Co-Simulation conjointe

La figure A.2 présente un aperçu de la co-simulation. Trois domaines dépendants sont présents : celui d'en haut est le modèle PRISMSYS, décrit auparavant dans A.1, celui d'en bas décrit la co-simulation TIMESQUARE et *Scilab* et celui de milieu définit la

sémantique d'exécution du comportement du modèle PRISMSYS. Nous nous intéressons dans cette section au domaine d'en bas, le T2 BACKEND FOR SCILAB. Les deux premiers domaines sont présentés dans [5].

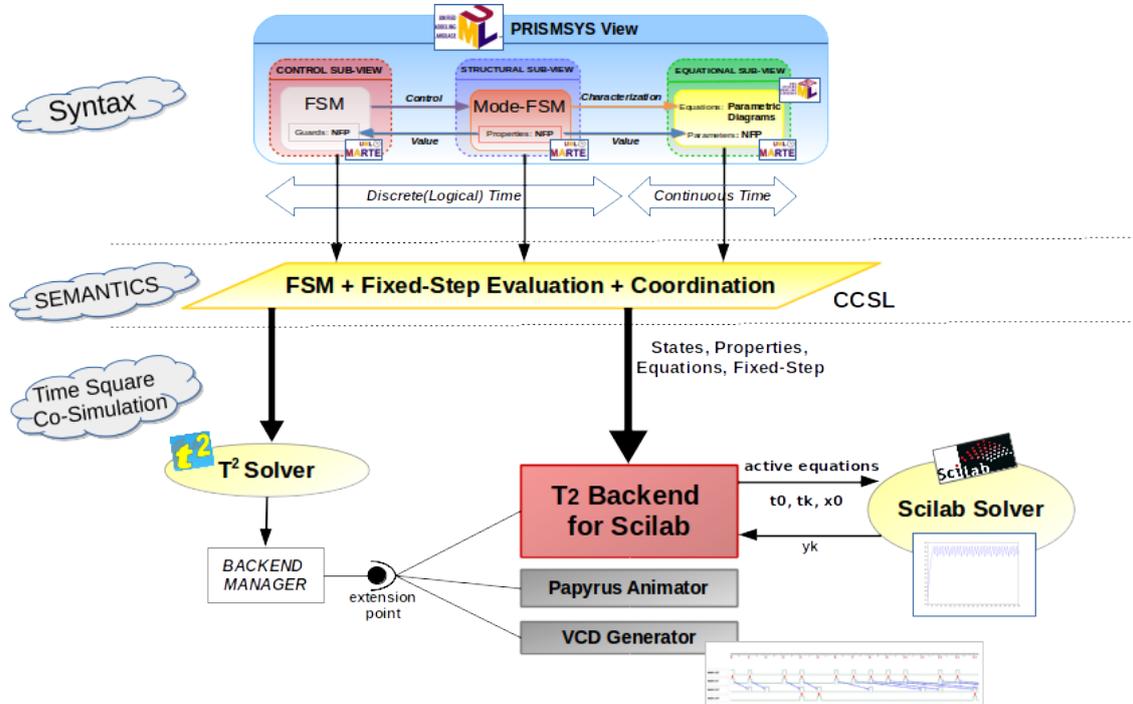


FIGURE A.2: Aperçu de T2 BACKEND FOR SCILAB

TIMESQUARE offre la possibilité d'ajouter des *backends* définis par l'utilisateur déclenchant des actions spécifiques sur des occurrences d'événements ou relations. Ces nouveaux *backends* peuvent être ajoutés au *backend manager* en passant par un point d'extension d'Eclipse¹. Durant la simulation, le *backend manager* reçoit le statut de chaque horloge (elle tique ou pas) à chaque pas de simulation. Il reçoit également le statut des relations (coïncidence et causalité) aussi bien que le statut des assertions (violé ou pas). Un développeur peut créer son propre *backend* sensible à certains de ces événements. TIMESQUARE est déjà distribué avec quelques *backends* tels que le *VCD backend* (qui dessine le diagramme temporel des horloges durant la simulation) et le *Papyrus Animator* (qui anime les éléments d'un modèle UML spécifié en *Papyrus*²). De la même façon, en utilisant le *backend manager* et en connectant au point spécifique d'extension, nous implémentons le T2 BACKEND FOR SCILAB comme un *plugin* d'Eclipse pour lier le solveur

1. <https://www.eclipse.org/>

2. <http://www.eclipse.org/papyrus/>

de TIMESQUARE au solveur de *Scilab*. Ce *backend* reçoit le statut des horloges (tique ou pas) à chaque pas de simulation et par conséquent configure le solveur de *Scilab* selon le modèle PRISMSYS et le conduit à évaluer les équations durant la simulation. L'utilisateur peut visualiser la valeur des propriétés non-fonctionnelles grâce à la fenêtre graphique de *Scilab* appelée par le solveur de *Scilab*.

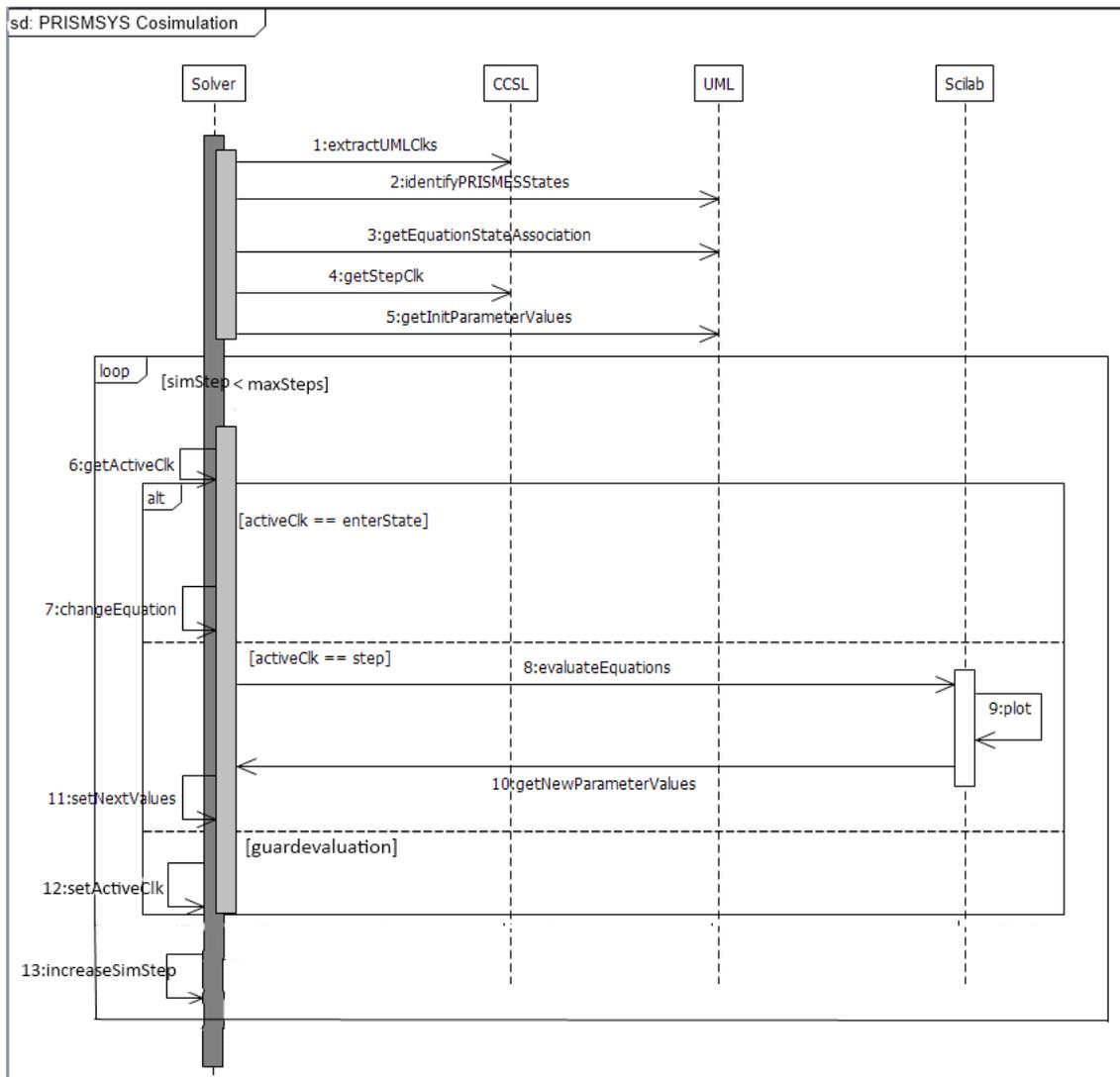


FIGURE A.3: Simulation de la vue thermique dans *Scilab* en utilisant T2 BACKEND FOR SCILAB

La figure A.3 présente un digramme de séquence résumant l'exécution de notre modèle. Les lignes de vie : *Solver* représente le solveur T2 BACKEND FOR SCILAB, *CCSL* représente la spécification CCSL, *UML* représente le modèle PRISMSYS et *Scilab* représente l'outil *Scilab*. Une fois la simulation démarre, le *Solver* extrait, de la spécification CCSL, l'horloge spécifiant le pas d'exécution et les horloges associées avec les états du

modèle UML. Également, le solveur extrait, du modèle UML, les équations associées avec ces états et les valeurs initiales des paramètres de ces équations. Une fois ces informations collectées, le solveur observe les horloges actives. A chaque fois un état est atteint, le solveur ajoute les équations associées avec cet état actif. Quand l'horloge de pas d'exécution tique, les équations actives sont évaluées par *Scilab*. Le résultat de cette évaluation est tracé par une fenêtre *Scilab* et enregistré dans T2 BACKEND FOR SCILAB. Pour chaque occurrence de pas d'exécution, les gardes sont évaluées. Si la garde évaluée est vraie, T2 BACKEND FOR SCILAB force le changement d'état du contrôleur.

A.3 Étude de cas : Contrôleur thermique d'un processeur

Le cas d'étude présent montre comment un contrôleur thermique peut être spécifié en utilisant T2 BACKEND FOR SCILAB. La température, une propriété non-fonctionnelle, doit être surveillée afin d'optimiser la consommation d'énergie. Dans ce cas d'étude, nous nous intéressons au CPU qui est la source de chaleur. Pour ce faire, nous définissons un modèle PRISMSYS basé sur une architecture mono-processeur. Dans la figure A.4, nous présentons une version simplifiée de la vue thermique de ce modèle. Dans cette figure, la sous vue structurelle représente une abstraction du CPU d'un point de vue thermique. Elle décrit une propriété non-fonctionnelle représentant la température (T). La sous vue équationnelle définit les équations qui caractérisent l'évolution de la température du CPU. La sous vue comportementale spécifie un contrôleur thermique, qui change l'activité du CPU en fonction de la température pour éviter de le surchauffer.

Le comportement thermique du CPU est décrit par deux modes : *COOLING* et *HEATING*. Chaque mode correspond à une évolution thermique différente. L'état *COOLING* indique que la température du CPU décroît. Tandis que l'état *HEATING* signifie que la température augmente. Pour définir l'évolution thermique, une équation est associée à chacun de ces états (point 2 dans la figure A.4). Ainsi, nous associons l'état *HEATING* avec l'équation $dT/dt = -0.1 * (T - 100)$ et l'état *COOLING* avec l'équation $dT/dt = -0.1 * (T - 30)$, en supposons que 100 est la température maximale du CPU et 30 est sa température minimale. L'unité de la température est le ° C.

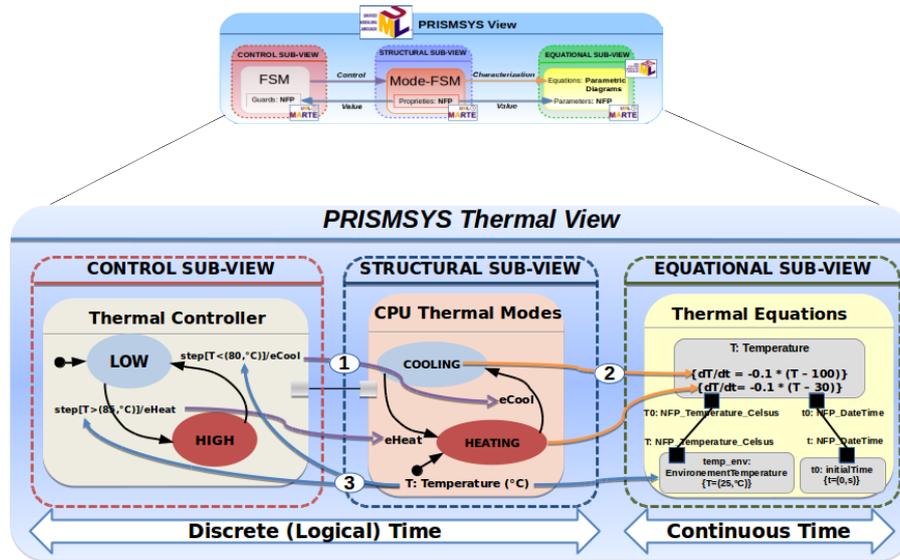


FIGURE A.4: Cas d'étude : Contrôleur thermique d'un CPU

Le changement entre les deux modes se produit lorsqu'une transition est tirée. Les transitions sont sensibles aux événements associés envoyés par la sous vue comportementale (point 1 dans la figure A.4). Ce dernier contient un contrôleur thermique dont la tâche est de contrôler l'évolution de la température du CPU. Son comportement est décrit par une autre machine d'états à deux états : *LOW* et *HIGH*. *LOW* indique que la température du CPU est inférieure à la température recommandée. *HIGH* exprime que la température dépasse une valeur maximale recommandée. Contrairement à la FSM-mode de la sous vue structurelle, la transition de la FSM du contrôleur est sensible à l'évaluation des gardes. Dans ce cas, les gardes sont évaluées si la température surpasse 85°C ou est inférieure à 80°C (point 3 dans la figure A.4). Une fois une transition est déclenchée, un événement spécifique est envoyé à la mode-FSM pour changer ses modes et ainsi activer l'équation à évaluer.

La figure A.5 montre l'évolution (tracée en *Scilab*) de la température *T* en fonction du temps *t*. Quand la simulation commence, à $t=0$, T2 BACKEND FOR SCILAB extrait les horloges associées avec les modes *COOLING* et *HEATING* et le pas d'horloge. T2 BACKEND FOR SCILAB identifie également les équations thermiques spécifiées dans le modèle. Dans notre modèle, nous assumons que la température ambiante est de 25°C . Ainsi, la simulation de TIMESQUARE commence à 25°C et les états *HEATING* et *LOW* (qui sont les états initiales) sont activés.

Quand la température dépasse 85° , la transition correspondante est déclenchée, l'état change à *HIGH* et l'événement *eCool* est envoyé à la mode-FSM du CPU. Cet événement cause la sortie du mode *HEATING* pour entrer dans l'état *COOLING*, et par conséquent la désactivation de l'équation $dT/dt = -0.1 * (T - 100)$ et l'activation de l'équation $dT/dt = -0.1 * (T - 30)$. Dans les pas de simulation suivante, l'équation associée à *COOLING* est évaluée et la température diminue. Quand la température est inférieure à 80° , la machine d'état du contrôleur change à l'état *LOW*. Notons que dans la figure [A.5](#), la température dépasse un peu le seuil prédéfini par la garde. Ceci est dû à la surveillance périodique de la température et l'évaluation périodique des gardes. Pour limiter ce phénomène, nous pouvons diminuer le pas d'exécution.

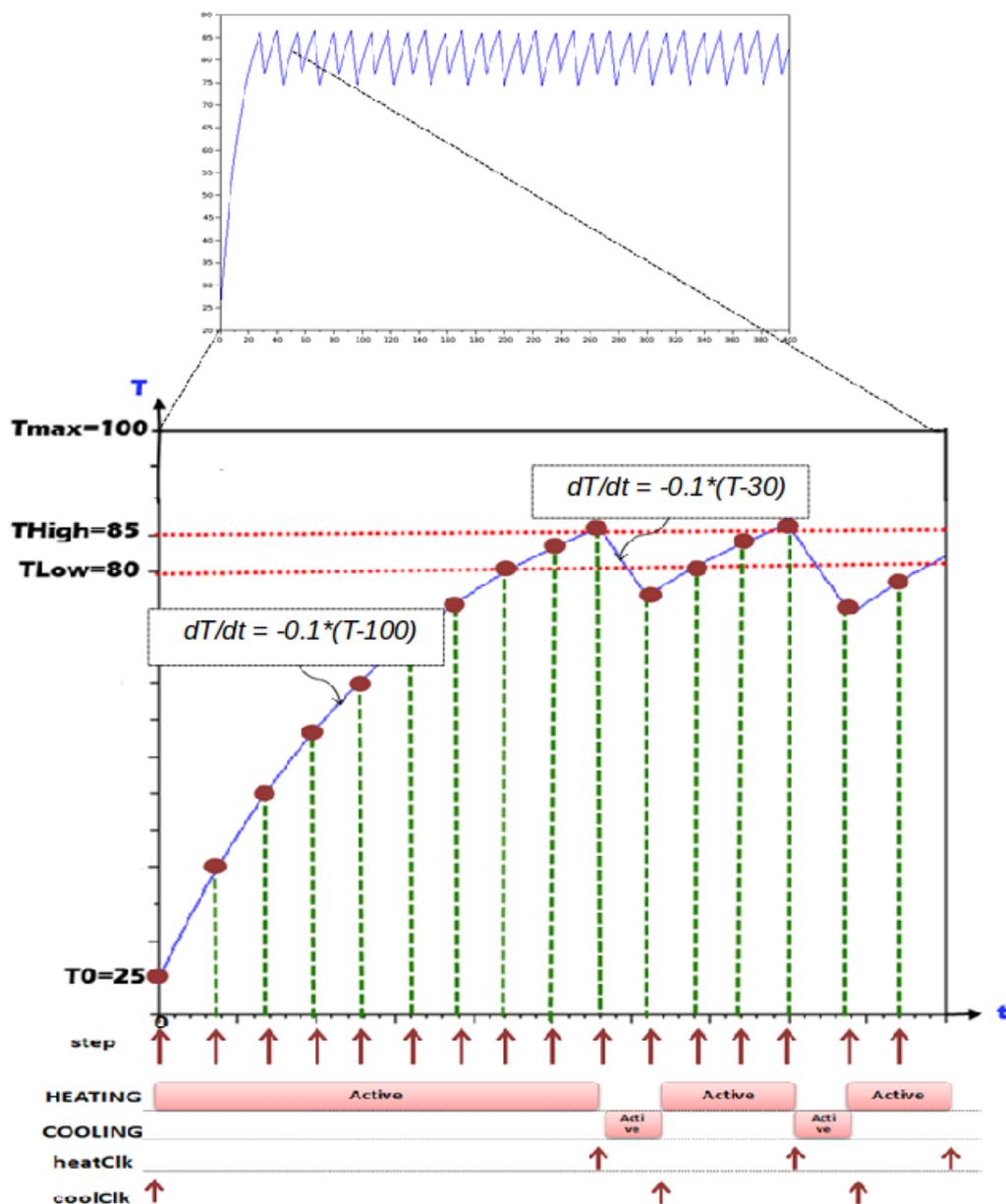


FIGURE A.5: Diagramme de séquence de la simulation de T2 BACKEND FOR SCILAB

Table des figures

| | | |
|------|--|----|
| 1.1 | Évolution proportionnelle du nombre de cœurs par rapport à l'énergie, la fréquence et la performance pour les SoCs [3] | 3 |
| 1.2 | Illustration générale des contributions | 7 |
| 2.1 | Exemple d'un système embarqué hétérogène : Téléphone portable | 14 |
| 2.2 | Modèle de cycle de développement en V | 15 |
| 2.3 | Allocation | 17 |
| 2.4 | KPN à deux processus | 21 |
| 2.5 | Modèle SDF à deux tâches | 21 |
| 2.6 | Une chronologie partielle des principaux modèles de <i>Process Network</i> | 22 |
| 2.7 | Puissance Consommée | 25 |
| 2.8 | Graphe de tâches sous Platform Architect MCO | 32 |
| 2.9 | Graphe de tâches sous Platform Architect MCO | 33 |
| 2.10 | Structure des différents niveaux de modélisation de l'approche MDA | 35 |
| 3.1 | Vue d'ensemble de notre approche MU _V ARCH | 45 |
| 3.2 | Extrait du méta-modèle de MU _V ARCH | 49 |
| 3.3 | Méta-modèle de <i>BackboneView</i> | 50 |
| 3.4 | Exemple d'un OPP | 51 |
| 3.5 | Méta-modèle d'une vue dans MU _V ARCH | 52 |
| 3.6 | Spécialisation de la vue de performance | 54 |
| 3.7 | Spécialisation de la vue de puissance | 55 |
| 3.8 | Spécialisation de la vue thermique | 56 |
| 3.9 | Spécialisation de la vue d'application | 57 |
| 3.10 | Relation d'association dans MU _V ARCH | 59 |
| 3.11 | Relation d'abstraction dans MU _V ARCH | 60 |
| 3.12 | Approche du solveur SMT existant [74] | 63 |
| 3.13 | Relation d'Allocation dans MU _V ARCH | 65 |
| 3.14 | Allocation globale pour les tâches | 66 |
| 3.15 | Allocation globale pour les ressources | 66 |
| 3.16 | Méta-modèle du contrôleur général dans MU _V ARCH | 67 |
| 4.1 | Architecture technique de notre atelier d'instrumentation de MU _V ARCH | 73 |
| 4.2 | ODROID-XU3 équipé d'un processeur <i>big.LITTLE</i> | 75 |

| | | |
|------|---|-----|
| 4.3 | Fichier “odesign” de spécification de notre éditeur | 77 |
| 4.4 | Extrait de l’éditeur réalisé avec Sirius | 79 |
| 4.5 | Modèle de la vue architecturale de <i>big.LITTLE</i> | 81 |
| 4.6 | Modèle de la vue Performance | 82 |
| 4.7 | Modèle de la vue Puissance | 83 |
| 4.8 | Modèle de la vue Thermique | 84 |
| 4.9 | Modèle de la vue d’application | 85 |
| 4.10 | Modèle global de <i>big.LITTLE</i> crée et édité avec Sirius | 86 |
| 4.11 | Template Acceleo pour la génération du fichier “.res” | 89 |
| 4.12 | Fichier “.res” généré depuis Acceleo | 89 |
| 4.13 | Extrait du fichier JSON de l’ordonnancement calculé généré par le solveur SMT | 91 |
| 4.14 | Extrait de l’analyseur (<i>Parser</i>) Java JSON | 92 |
| 4.15 | Modèle global de <i>big.LITTLE</i> crée et édité avec Sirius | 93 |
| 4.16 | Représentation des allocations des ressources | 94 |
| 4.17 | Représentation des allocations des tâches | 95 |
| 4.18 | Représentation en FSM du contrôleur général | 96 |
| 5.1 | Illustration des différentes contributions de la thèse | 99 |
| A.1 | Vue dans PRISMSYS | 104 |
| A.2 | Aperçu de T2 BACKEND FOR SCILAB | 105 |
| A.3 | Simulation de la vue thermique dans <i>Scilab</i> en utilisant T2 BACKEND FOR SCILAB | 106 |
| A.4 | Cas d’étude : Contrôleur thermique d’un CPU | 108 |
| A.5 | Diagramme de séquence de la simulation de T2 BACKEND FOR SCILAB | 110 |

Liste des tableaux

| | | |
|-----|--|----|
| 4.1 | Niveaux Voltage/Fréquence pour les cœurs big et LITTLE | 76 |
| 4.2 | OPPs des cœurs big et LITTLE employés dans notre exemple | 80 |
| 4.3 | Coûts des tâches en Méga-Cycles | 85 |

Abréviations

| | |
|----------------|---|
| AAA | A déquation A lgorithme A rchitecture |
| ADL | A rchitecture D escription L anguage |
| CCSL | C lock C onstraint S pecification L anguage |
| CD | C lock D omain |
| CPS | C yber P hysical S ystem |
| DSML | D omain S pecific M odeling L anguages |
| DFG | D ata F low G raph |
| DVFS | D ynamic V oltage and F requency S caling |
| ESL | E lectronic D esign L evel |
| FMI | F unctional M ock-up I nterface |
| FMU | F unctional M ock-up U nit |
| FSM | F inite S tate M achine |
| IDM | I ngénierie D irigée par les M odèles |
| ISO | I nternational S tandardization O rganization |
| IP | I ntellectual P roperty |
| MARTE | M odeling and A nalysis of R eal-time E mbedded systems |
| MOF | M eta O bject F acility |
| MoC | M odel of C omputation |
| MuVArch | M odélisation multi V ues A rchitecture |
| OMG | O bject M anagement G roup |

| | |
|--------------|---|
| OPP | O perating P erformance P oints |
| PBD | P latform B ased D esign |
| PD | P ower D omain |
| RTL | R egister T ransfer L evel |
| SMT | S atisfaisabilité M odulo T héories |
| SoC | S ystem o n C hip |
| SysML | S ystems M odeling L anguage |
| TLM | T ransaction L evel M odeling |
| TD | T hermal D omain |
| UML | U nified M odeling L anguage |
| UPF | U nified P ower F ormat |
| VUML | V iew B ased U nified M odeling |

Bibliographie

- [1] J.S. Kilby. Invention of the integrated circuit. *Electron Devices, IEEE Transactions on*, 23(7) :648–654, Jul 1976. ISSN 0018-9383. doi : 10.1109/T-ED.1976.18467.
- [2] Semiconductor Industry Association. International technology roadmap for semiconductors. 1999.
- [3] Steve Carlson. Low power solutions. April 2015. URL <http://www.eda.org/edps/Papers/1-1%20Steve%20Carlson.pdf>.
- [4] Julien DeAntoni and Frédéric Mallet. Timesquare : Treat your models with logical time. In *TOOLS (50)*, volume 7304, pages 34–41. Springer, 2012.
- [5] Carlos Ernesto Gómez Cárdenas. *Modeling functional and non-functional properties of systems based on a multi-view approach*. Theses, Université Nice Sophia Antipolis, December 2013. URL <https://tel.archives-ouvertes.fr/tel-00931001>.
- [6] ISO/IEC/IEEE. Systems and software engineering – architecture description. *ISO/IEC/IEEE 42010 :2011(E) (Revision of ISO/IEC 42010 :2007 and IEEE Std 1471-2000)*, pages 1–46, 1 2011.
- [7] Johan Eker, Jörn W. Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Stephen Neuendorffer, Sonia Sachs, and Yuhong Xiong. Taming heterogeneity - the ptolemy approach. In *Proceedings of the IEEE*, pages 127–144, 2003.
- [8] Edward A. Lee. Cyber-physical systems - are computing foundations adequate? In *Position Paper for NSF Workshop On Cyber-Physical Systems : Research Motivation, Techniques and Roadmap*, October 2006. URL <http://chess.eecs.berkeley.edu/pubs/329.html>.

- [9] Edward A. Lee. Cyber physical systems : Design challenges. Technical Report UCB/EECS-2008-8, EECS Department, University of California, Berkeley, Jan 2008. URL <http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-8.html>.
- [10] Document Number ANSI/GEIA EIA-632. Processes for engineering a system. *Government Electronics and Information Technology Association*.
- [11] UML2.0. Unified modeling language (UML), object management group, http://www.omg.org/technology/documents/modeling_spec_catalog.htm.
- [12] OMG. OMG Systems Modeling Language (OMG SysML), Version 1.3, 2012. URL <http://www.omg.org/spec/SysML/1.3/>.
- [13] Jean-Luc Voirin. 9.1.1 method & tools for constrained system architecting. *INCOSE International Symposium*, 18(1) :981–995, 2008. ISSN 2334-5837. doi : 10.1002/j.2334-5837.2008.tb00857.x. URL <http://dx.doi.org/10.1002/j.2334-5837.2008.tb00857.x>.
- [14] Peter H. Feiler, David P. Gluch, and John J. Hudak. The architecture analysis & design language (AADL) : An introduction. Technical Report CMU/SEI-2006-TN-011, Software Engineering Institute, Carnegie Mellon University, 2006. URL <http://www.aadl.info/aadl/currentsite/currentusers/notation.html>.
- [15] Object Management Group (OMG). The uml marte standardized profile. In *Proceedings of the 17th IFAC World Congress*, 2008.
- [16] E.A. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 17(12) :1217–1229, Dec 1998. ISSN 0278-0070.
- [17] Darko Kirovski and Miodrag Potkonjak. System-level synthesis of low-power hard real-time systems, 1997.
- [18] Ti-Yen Yen and Wayne Wolf. Sensitivity-driven co-synthesis of distributed embedded systems. In *Proceedings of the 8th International Symposium on System Synthesis*, ISSS '95, pages 4–9, New York, NY, USA, 1995. ACM. ISBN 0-89791-771-5. doi : 10.1145/224486.224488. URL <http://doi.acm.org/10.1145/224486.224488>.

- [19] K.S. Vallerio and N.K. Jha. Task graph extraction for embedded system synthesis. In *VLSI Design, 2003. Proceedings. 16th International Conference on*, pages 480–486, Jan 2003. doi : 10.1109/ICVD.2003.1183180.
- [20] Hyunok Oh and Soonhoi Ha. A hardware-software cosynthesis technique based on heterogeneous multiprocessor scheduling. In *Proceedings of the Seventh International Workshop on Hardware/Software Codesign, CODES '99*, pages 183–187, New York, NY, USA, 1999. ACM. ISBN 1-58113-132-1. doi : 10.1145/301177.301524. URL <http://doi.acm.org/10.1145/301177.301524>.
- [21] A. Arnold. *Systèmes de transitions finis et sémantique des processus communicants*. E.R.I. Etudes et recherches en informatique. Masson, 1992. ISBN 9782225827464. URL <https://books.google.fr/books?id=bBhnAAAACA AJ>.
- [22] Massimiliano Chiodo, Paolo Giusto, Harry Hsieh, Attila Jurecska, Luciano Lavagno, and Alberto Sangiovanni-Vincentelli. A formal specification model for hardware/-software codesign. Technical report, Berkeley, CA, USA, 1993.
- [23] Felice Balarin, Massimiliano Chiodo, Paolo Giusto, Harry Hsieh, Attila Jurecska, Luciano Lavagno, Claudio Passerone, Alberto Sangiovanni-Vincentelli, Ellen Sentovich, Kei Suzuki, and Bassam Tabbara, editors. *Hardware-software Co-design of Embedded Systems : The POLIS Approach*. Kluwer Academic Publishers, Norwell, MA, USA, 1997. ISBN 0-7923-9936-6.
- [24] David Harel. Statecharts : A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3) :231–274, June 1987. ISSN 0167-6423. doi : 10.1016/0167-6423(87)90035-9. URL [http://dx.doi.org/10.1016/0167-6423\(87\)90035-9](http://dx.doi.org/10.1016/0167-6423(87)90035-9).
- [25] Daniel D. Gajski, Frank Vahid, and Sanjiv Narayan. Speccharts : A vhdl front-end for embedded systems, 1003.
- [26] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information processing*, pages 471–475, Stockholm, Sweden, Aug 1974. North Holland, Amsterdam.
- [27] Edward A. Lee and et al. Synchronous data flow, 1987.

- [28] Richard M. Karp and Raymond E. Miller. Properties of a model for parallel computations : Determinacy, termination, queueing. *SIAM Journal on Applied Mathematics*, 14(6) :1390–1411, 1966.
- [29] Arvind and Vinod Kathail. A multiple processor data flow machine that supports generalized procedures. In *Proceedings of the 8th Annual Symposium on Computer Architecture*, ISCA '81, pages 291–302, Los Alamitos, CA, USA, 1981. IEEE Computer Society Press. URL <http://dl.acm.org/citation.cfm?id=800052.801882>.
- [30] David E. Culler. *Managing parallelism and resources in scientific dataflow programs*. PhD thesis, 1990. URL <http://opac.inria.fr/record=b1001871>. PHD.
- [31] S. Ritz, M. Pankert, and H. Meyr. High level software synthesis for signal processing systems. In *Application Specific Array Processors, 1992. Proceedings of the International Conference on*, pages 679–693, Aug 1992.
- [32] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete. Cyclo-static data flow. In *Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on*, volume 5, pages 3255–3258 vol.5, May 1995. doi : 10.1109/ICASSP.1995.479579.
- [33] P. Eles, K. Kuchcinski, Z. Peng, A. Doboli, and P. Pop. Scheduling of conditional process graphs for the synthesis of embedded systems. In *Proceedings of the Conference on Design, Automation and Test in Europe, DATE '98*, pages 132–139, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-8186-8359-7.
- [34] A. Girault, Bilung Lee, and E. A. Lee. Hierarchical finite state machines with multiple concurrency models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(6) :742–760, Jun 1999. ISSN 0278-0070.
- [35] Marc Geilen and Twan Basten. Reactive process networks. In *Proceedings of the 4th ACM International Conference on Embedded Software*, EMSOFT '04, pages 137–146, New York, NY, USA, 2004. ACM. ISBN 1-58113-860-1. doi : 10.1145/1017753.1017778. URL <http://doi.acm.org/10.1145/1017753.1017778>.
- [36] B. D. Theelen, M. C. W. Geilen, T. Basten, J. P. M. Voeten, S. V. Gheorghita, and S. Stuijk. A scenario-aware data flow model for combined long-run average and worst-case performance analysis. In *Formal Methods and Models for Co-Design, 2006*.

- MEMOCODE '06. Proceedings. Fourth ACM and IEEE International Conference on*, pages 185–194, July 2006. doi : 10.1109/MEMCOD.2006.1695924.
- [37] Julien Boucaron, Anthony Coadou, Benoît Ferrero, Jean-Vivien Millo, and Robert De Simone. Kahn-extended Event Graphs. Research Report RR-6541, INRIA, 2008. URL <https://hal.inria.fr/inria-00281559>.
- [38] Axel Jantsch and Hannu Tenhunen, editors. *Networks on Chip*. Kluwer Academic Publishers, Hingham, MA, USA, 2003. ISBN 1-4020-7392-5.
- [39] Ieee standard for standard systemc language reference manual. *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, pages 1–638, Jan 2012.
- [40] SoCLib, <http://www.soclib.fr/trac/dev>, 2012.
- [41] Brian Bailey. System level virtual prototyping becomes a reareal with ovp donation from imperas., 2008. URL <http://www.ovpworld.org/system-level-virtual-prototyping-becomes-a-reality-by-brian-bailey>.
- [42] W. Kruijtzter, P. van der Wolf, E. de Kock, J. Stuyt, W. Ecker, A. Mayer, S. Hustin, C. Amerijckx, S. de Paoli, and E. Vaumorin. Industrial ip integration flows based on ip-xact standards. In *Design, Automation and Test in Europe, 2008. DATE '08*, pages 32–37, March 2008. doi : 10.1109/DATE.2008.4484656.
- [43] Luca Benini, Alessandro Bogliolo, and Giovanni De Micheli. A survey of design techniques for system-level dynamic power management. *IEEE TRANSACTIONS ON VLSI SYSTEMS*, 8(3) :299–316, 2000.
- [44] Ben Abdallah Faten. *Study and optimisation of dynamically reconfigurable architectures - processors interaction*. Theses, Université Rennes 1, October 2009. URL <https://tel.archives-ouvertes.fr/tel-00438608>.
- [45] Taewhan Kim and Pilok Lim. Thermal-aware high-level synthesis based on network flow method. In *Hardware/Software Codesign and System Synthesis, 2006. CODES+ISSS '06. Proceedings of the 4th International Conference*, pages 124–129, Oct 2006.
- [46] Jingcao Hu, Youngsoo Shin, N. Dhanwada, and R. Marculescu. Architecting voltage islands in core-based system-on-a-chip designs. In *Low Power Electronics and Design*,

2004. *ISLPED '04. Proceedings of the 2004 International Symposium on*, pages 180–185, Aug 2004.
- [47] Ayse K. Coskun, Tajana Simunic Rosing, Kresimir Mihic, Giovanni De Micheli, and Yusuf Leblebici. Analysis and Optimization of MPSoC Reliability. *Journal of Low Power Electronics*, 2(1) :56–69, 2006. ISSN 1546-1998.
- [48] Jan Haase, Markus Damm, Dennis Hauser, and Klaus Waldschmidt. 225 :205–214, 2006.
- [49] Ayse K. Coskun, Richard Strong, Dean M. Tullsen, and Tajana Simunic Rosing. Evaluating the impact of job scheduling and power management on processor lifetime for chip multiprocessors. In *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '09, pages 169–180, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-511-6.
- [50] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1) :46–61, January 1973. ISSN 0004-5411. URL <http://doi.acm.org/10.1145/321738.321743>.
- [51] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli. Metropolis : an integrated electronic system design environment. *Computer*, 36(4) :45–52, April 2003. ISSN 0018-9162. doi : 10.1109/MC.2003.1193228.
- [52] S. Mohanty and V.K. Prasanna. Rapid system-level performance evaluation and optimization for application mapping onto soc architectures. In *ASIC/SOC Conference, 2002. 15th Annual IEEE International*, pages 160–167, Sept 2002.
- [53] Ingo Sander and Axel Jantsch. System modeling and transformational design refinement in forsyde [formal system design]. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 23(1) :17–32, 2004.
- [54] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. Streamit : A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 179–196, London, UK, UK, 2002. Springer-Verlag. ISBN 3-540-43369-4. URL <http://dl.acm.org/citation.cfm?id=647478.727935>.

- [55] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Paul Johnson, Jae-Wook Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpfen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. The raw microprocessor : A computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22(2) :25–35, March 2002. ISSN 0272-1732. doi : 10.1109/MM.2002.997877. URL <http://dx.doi.org/10.1109/MM.2002.997877>.
- [56] David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F. Brown III, and Anant Agarwal. On-chip interconnection architecture of the tile processor. *IEEE Micro*, 27(5) :15–31, September 2007. ISSN 0272-1732. doi : 10.1109/MM.2007.89. URL <http://dx.doi.org/10.1109/MM.2007.89>.
- [57] Sander Stuijk, Marc Geilen, and Twan Basten. Sdf3 : Sdf for free. In *Proceedings of the Sixth International Conference on Application of Concurrency to System Design, ACSD '06*, pages 276–278, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2556-3. doi : 10.1109/ACSD.2006.23. URL <http://dx.doi.org/10.1109/ACSD.2006.23>.
- [58] Kees Goossens, John Dielissen, and Andrei Radulescu. Aethereal network on chip : Concepts, architectures, and implementations. *IEEE Des. Test*, 22(5) :414–421, September 2005. ISSN 0740-7475.
- [59] Edward A. Lee, C. Hylands, J. Janneck, J. Davis II, J. Liu, X. Liu, S. Neuendorfer, S. Sachs M. Stewart, K. Vissers, and P. Whitaker. Overview of the ptolemy project. Technical Report UCB/ERL M01/11, EECS Department, University of California, Berkeley, 2001. URL <http://www.eecs.berkeley.edu/Pubs/TechRpts/2001/3947.html>.
- [60] Y. Sorel. Syndex : System-level cad software for optimizing distributed real-time embedded systems. *Journal ERCIM News*, 59 :68–69, October 2004. URL <http://www-rocq.inria.fr/syndex/publications/pubs/ercim04/ercim04.pdf>.
- [61] E. Belhaire, E. Bourennane, G. Bouvier, D. Demigny, P. Garda, L. Kessal, L. Lacasagne, F. Lohier, M. Paindavoine, Y. Sorel, L. Torres, and S. Weber. *Méthodes et architectures pour le traitement du signal et des images en temps réel*, chapter 5 : Y.

- Sorel, Méthodologie AAA et logiciel SynDEx, pages 79–108. IC2. Hermes, 2001. URL <http://www-rocq.inria.fr/syndex/publications/pubs/ic201/ic201.pdf>.
- [62] Y. Sorel. Massively parallel systems with real time constraints, the algorithm architecture adequation methodology. In *Proceedings of Conference on Massively Parallel Computing Systems, MPCs'94*, Ischia, Italy, May 1994. URL <http://www-rocq.inria.fr/syndex/publications/pubs/mpcs94/mpcs94.pdf>.
- [63] Douglas C. Schmidt. Model-driven engineering. *IEEE Computer*, 39(2), February 2006. URL <http://www.truststc.org/pubs/30.html>.
- [64] Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained : The Model Driven Architecture : Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003. ISBN 032119442X.
- [65] OMG. Meta object facility (mof) 2.0 core specification. object management group , v2.0, October 2003.
- [66] Eclipse Foundation. Eclipse modeling framework (emf), 2016. URL <https://www.eclipse.org/modeling/emf/>.
- [67] Eclipse Foundation. Ecore tools, 2013. URL <http://www.eclipse.org/ecoretools/>.
- [68] Eclipse Foundation. Object constraint language (ocl), 2014. URL <https://wiki.eclipse.org/OCL>.
- [69] Eclipse Foundation. Sirius, 2016. URL <https://www.eclipse.org/sirius/>.
- [70] Mahmoud Nassar. Vuml : a viewpoint oriented uml extension. In *ASE*, pages 373–376. IEEE Computer Society, 2003. ISBN 0-7695-2035-9. URL <http://dblp.uni-trier.de/db/conf/kbse/ase2003.html#Nassar03>.
- [71] P. Roques. *SysML par l'exemple - Un langage de modélisation pour systèmes complexes*. eBooks Informatique Eyrolles. Eyrolles, 2011. ISBN 9782212850062. URL <https://books.google.fr/books?id=HQncWTDH120C>.
- [72] Frédéric Boulanger, Christophe Jacquet, Cécile Hardebolle, and Elyes Rouis. Modeling heterogeneous points of view with modhel'x. In Sudipto Ghosh, editor, *Models*

- in Software Engineering*, volume 6002 of *Lecture Notes in Computer Science*, pages 310–324. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-12260-6.
- [73] F. Boulanger and C. Hardebolle. Simulation of multi-formalism models with model’x. In *Software Testing, Verification, and Validation, 2008 1st International Conference on*, pages 318–327, April 2008.
- [74] Emilien Kofman, Robert De Simone, and Amani Khecharem. Multicore SMT scheduling of periodic task systems with energy minimization. In *Workshop on Highly-Reliable Power-Efficient Embedded Designs*, Barcelone, Spain, March 2016. URL <https://hal.inria.fr/hal-01282264>.
- [75] ARM. big.little technology : The future of mobile, 2013. URL https://www.arm.com/files/pdf/big_LITTLE_Technology_the_Futue_of_Mobile.pdf.
- [76] ODROID. Odroid-xu3, 2015. URL <http://odroid.com/dokuwiki/doku.php?id=en:odroid-xu3>.
- [77] Mike Anderson. Scheduler options in big.little android platforms, 2015. URL http://events.linuxfoundation.org/sites/events/files/slides/GTS_Anderson.pdf.
- [78] Linaro. Energy aware scheduling [eas], 2015. URL <https://wiki.linaro.org/WorkingGroups/PowerManagement/Resources/EAS>.
- [79] Eclipse. Acceleo, 2012. URL <http://www.eclipse.org/acceleo/>.
- [80] Charles André. Syntax and Semantics of the Clock Constraint Specification Language (CCSL). Research Report RR-6925, 2009. URL <https://hal.inria.fr/inria-00384077>.
- [81] Stephen L. Campbell, Jean-Philippe Chancelier, and Ramine Nikoukhah. *Modeling and simulation in Scilab/Scicos with ScicosLab 4.4*. Springer, New York, 2010. ISBN 978-1-4419-5526-5. URL <http://opac.inria.fr/record=b1130798>.