Modélisation et Caractérisation d'une Plate-Forme SOC Hétérogène : Application à la Radio Logicielle.

Samuel Rouxel

N° d'ordre : 77

Thèse

présentée et soutenue publiquement le

5 décembre 2006

par

Samuel Rouxel

pour obtenir le grade de

DOCTEUR ÈS SCIENCES de l'Université de Bretagne Sud

Spécialité : Sciences et Sciences de l'Ingénieur
Mention : Électronique et Informatique Industrielle

Modélisation et Caractérisation de Plates-Formes SoC Hétérogènes : Application à la Radio Logicielle

---

devant le jury composé de :

M*me*	Fabienne Nouvel	Maître de Conférences (HDR), IETR, Rennes	Rapporteur
M.	Charles André	Professeur, I3S, Sophia Antipolis	Rapporteur
M.	Christophe Moy	Enseignant chercheur, Supelec, Rennes	Examinateur
M.	Joël Champeau	Maître de Conférences, ENSIETA, Brest	Examinateur
M.	Jean-Luc Philippe	Professeur, LESTER, Lorient	Directeur de thèse
M.	Guy Gogniat	Maître de Conférences, LESTER, Lorient	Examinateur

Laboratoire d'Électronique des Systèmes Temps Réel (LESTER) CNRS FRE2734
Université de Bretagne Sud

Résumé

L'évolution des systèmes électroniques en terme de miniaturisation, de puissance et de ressources de calcul disponibles devient telle que des applications de plus en plus complexes peuvent être implantées sur un même circuit. Pour développer de tels systèmes (architecture matérielle et application), les outils et méthodes de conception actuels se placent à un haut-niveau d'abstraction (niveau système). Ces outils permettent de modéliser et de spécifier les systèmes, afin de vérifier et de tester leurs fonctionnalités. Ils évaluent également les performances de ceux-ci et renseignent le concepteur sur le respect ou non des contraintes spécifiées. Ces outils se concentrent principalement sur l'aspect fonctionnel des applications. Or à l'heure actuelle, afin de ne pas avoir à redévelopper à chaque fois certaines fonctions (cryptage, filtrage, etc.) suivant la cible visée, les industriels et universitaires développent des composants préconçus, les IP, matériels (fonctions déjà synthétisées) ou logiciels (codes sources déjà écrits). Certaines de ces IP sont paramétrables et leurs fonctionnalités sont garanties. Il est donc indispensable de prendre en compte ces composants dans un flot de conception de système sur puce (SoC), et de proposer des outils qui utilisent leurs caractéristiques afin de réduire davantage le temps de validation du système global ; ceci permet de valider très tôt dans le cycle de conception les choix architecturaux du système à concevoir.

Les travaux de cette thèse se sont déroulés dans le cadre du projet RNRT A3S, et intègrent cette notion de composants au sein d'une méthodologie de conception d'une plate-forme SoC (System on Chip), basée sur le langage de modélisation UML (Unified Modeling Language). Cette méthodologie propose un environnement de modélisation qui repose sur le profil UML A3S, appuyant l'approche d'architectures basées sur les modèles (MDA). Elle permet en outre de modéliser et de spécifier un système en décorrélant dans un premier temps l'application logicielle et l'architecture matérielle pouvant la supporter. Le modèle applicatif obtenu (PIM) est indépendant de la plate-forme matérielle (modèle architectural). C'est seulement dans un second temps, après avoir modélisé une ou plusieurs applications et plates-formes matérielles que le concepteur peut tester un ou plusieurs choix d'implémentation. Le modèle devient alors dépendant de la plate-forme matérielle (modèle PSM). La méthodologie inclut par ailleurs des règles de vérifications et de validation des systèmes (respect des contraintes structurelles et temps réel). L'outil développé, mettant en œuvre cette méthodologie exploite le profil A3S, et permet ainsi de modéliser, de spécifier des systèmes complexes et de vérifier les modélisations effectuées en UML, et renseigne le concepteur sur la faisabilité du système développé (ordonnançabilité, taux d'occupation), après l'analyse de l'outil XAPAT intégré. Cette conception basée sur les modèles implique un niveau de représentation des éléments qui les compose, en adéquation avec le niveau de conception désiré. Elle implique également une manipulation aisée pour les architectes systèmes, en les assistant par une sémantique riche pour permettre l'analyse et la validation des systèmes.

Le domaine applicatif retenu est celui des systèmes Radio Logicielle. Une chaîne UMTS a

notamment permis la validation de l'outil en confrontant les résultats estimés de l'outil, à ceux mesurés sur une plate-forme temps réel hétérogène (multi-DSP, multi-FPGA). Une partie du travail s'est concentré sur l'identification des composants utiles à la conception des systèmes SoC, et de leurs caractéristiques, en adéquation avec le niveau d'abstraction considéré. Ces composants sont de natures différentes (logiciels, matériels), et disposent de caractéristiques différentes suivant le niveau de configuration (modèle PIM, modèle PSM). Une autre partie des travaux a porté sur la définition des modèles UML, et donc du profil, qui définissent la sémantique des différents composants identifiés en fonction de la configuration (PIM, PSM), ainsi que leurs relations. Une réflexion a été nécessaire afin d'élaborer les diverses règles de vérification et modèles d'exécution qui permettent d'informer le concepteur de ses erreurs et de la faisabilité du système modélisé. Un modèle de système d'exploitation a également été inclus, enrichissant la liste des éléments (composants) déjà définis et démontrant l'extensibilité du profil.

Mots clefs : *UML, prototypage rapide, codesign, approche MDA, PIM, PSM, application Radio Logicielle, vérification de cohérence, ordonnancement temps réel.*

Remerciements

Cette thèse est le résultat d'un travail de 3 ans effectué au sein du Laboratoire des Systèmes Électroniques TEmps Réel (LESTER) de l'Université de Bretagne Sud. Je remercie les directeurs successifs, Monsieur *Éric Martin*, Professeur des Universités et aujourd'hui président de l'Université de Bretagne Sud, et Monsieur *Emmanuel Boutillon*, Professeur des Universités et actuel directeur du LESTER, pour m'avoir accueilli dans ce laboratoire.

Je remercie particulièrement mon directeur de thèse Monsieur *Jean-Luc Philippe*, de m'avoir proposé cette thèse qui a eu du mal à arriver, mais qui m'a beaucoup apportée. Je le remercie pour sa disponibilité, son accompagnement et ses conseils bénéfiques.

Je tenais à remercier chaleureusement mon co-encadrant de thèse, *Guy Gogniat*, pour la qualité de son encadrement, son soutien, son écoute, sa gentillesse et ses encouragements.

Je remercie Madame *Fabienne Uzel-Nouvel* et Monsieur *Charles André* d'avoir accepté d'être les rapporteurs de ce manuscrit et membre du jury.

Je remercie également les *membres du projet A3S* avec lesquels j'ai eu plaisir à travailler pour ma première expérience de projet de recherche national et qui ont participé au bon déroulement de cette thèse.

Je tiens aussi à remercier l'ensemble des membres du laboratoire *LESTER* pour leur accueil, leur gentillesse, leur courtoisie, et leur bonne humeur.

J'ai également partagé mon quotidien avec des collègues de bureau qui ont fait que ces trois années se sont encore mieux déroulées. Je tenais donc à remercier en particulier *Bertrand Le Gal* et *Samuel Evain* pour leur amitié et leur soutien précieux.

Enfin, un grand merci à ma *famille*, à mes *amis* et à *Nathalie*, pour leur présence et leur soutien.

Table des matières

Table des figures

Introduction

Télécommunication d'hier et d'aujourd'hui

"La communication est l'acte par lequel chaque conscience sort d'elle-même et dépasse son intériorité pour s'ouvrir à autrui". Cette citation de J. Russ [5] illustre que les personnes existent socialement à travers la communication, ce qui explique le besoin qu'ont eu les hommes de tous temps à communiquer entre eux. L'outil majeur dont l'homme dispose pour échanger avec les autres est le langage. Cette faculté naturelle chez l'homme, nécessite néanmoins une codification, l'utilisation d'un protocole, qui correspond chez lui au langage. En effet, l'homme peut échanger en parlant, mais pour se comprendre il faut parler la même langue, utiliser le même protocole.

Au fil du temps, les moyens de communication des hommes ont évolué au même rythme que l'évolution de leur moeurs et des besoins qui y sont liés. Jadis, proportionnellement plus sédentaires que nomades, les êtres humains communiquaient au moyen d'un outil unique, qui est l'organe de la voix. La portée de celle-ci a comme contrainte majeure de fixer la distance séparant deux interlocuteurs. C'est pourquoi bon nombre de personnes ont réfléchi, au travers du temps, à différents moyens et outils, permettant de communiquer, si possible en instantané, tout en s'affranchissant des distances. Les premières solutions trouvées furent tout d'abord de crier plus fort, puis de trouver des alternatives à la parole. Ces alternatives sont basées sur la mise en place d'une paire **outil-protocole**, où l'information à transmettre est mise en forme (suivant un protocole) et transmise via un support (outil), autre que la parole. Ce fut le cas des indiens d'Amérique qui communiquaient à travers des signaux de fumées (avant l'arrivée des espagnols), la vision permettant de voir plus loin des signaux et donc de communiquer à des distances supérieures. Dans cet exemple, le support est le nuage de fumée, et le protocole correspond à la fréquence des signaux déterminant l'information codée. Un autre peuple, celui d'Afrique utilisait des tam-tam pour dialoguer. Le principe des allumages de feux dans des tours distantes de plusieurs kilomètres pour transférer une information vint plus tard, suivi par les sémaphores et le télégraphe optique (Chappe en 1790). Ce dernier repose également sur des tours, surmontées de bras articulés qui transmettent des signaux codés de proche en proche sur de longues distances. Chacun de ces moyens a un nombre de codes limité qui ne permet pas d'avoir de vraies discussions. Avec la découverte de l'électricité, les chercheurs expérimentent et mettent au point des protocoles et outils permettant de s'échanger des messages à très longue distance quasiment en instantané. La naissance du télégraphe en 1838 par Wheaston associé au morse (1844) de Samuel Morse, permis d'envoyer le message "What hath God wrought[1]" sur une distance de 60Km via le réseau télégraphique. Ensuite l'onde radio est née (issue des lampes), permettant d'envoyer des signaux sans support fixe. Les évolutions technologiques issues de la recherche ont fait progresser en parallèle les moyens de communication avec et

[1] "De quels bienfaits Dieu ne nous a t-il pas gratifié"

sans support physique, en voyant se succéder l'apparition du téléphone (1876), du télégraphe sans fil (1899), du téléphone sans fil (1906), ainsi que les protocoles utilisés par ces différents outils. Les performances n'ont cessé de croître à chaque évolution, que ce soit la qualité des signaux ou le volume d'informations transmis, répondant ainsi aux désirs des utilisateurs. En effet, au fil du temps, la démographie des peuples, leur répartition géographique, le mode de vie des personnes et le développement industriel, ont entraîné des demandes diversifiées des modes de communication. Ces demandes se sont concrétisées au fil de l'évolution technologique.

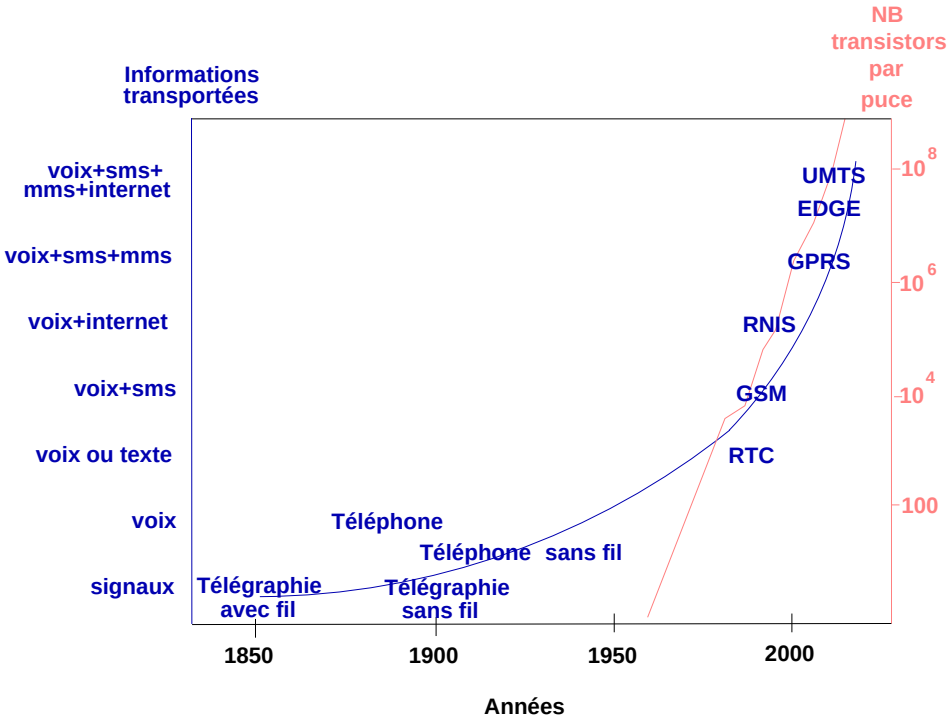


FIG. 1 – Évolution des technologies de communications associées aux performances technologiques des puces

La technologie est le support fournissant l'ensemble des propriétés et des fonctionnalités des outils de communications. C'est pourquoi, comme le montre la figure 1, l'augmentation du potentiel des composants électroniques (nombre de transistors dans les processeurs) permet de développer et de supporter des protocoles offrant plus de services à l'utilisateur. Avec le développement de la téléphonie mobile, et des standards de communication tels que : le GSM, le GPRS, l'EDGE et l'UMTS, avec des débits de plus en plus élevés, ce sont respectivement des textos (SMS), des messages multimédias (MMS), l'Internet et de la vidéo que l'on peut désormais transmettre en plus de la voix. De plus, différents standards sont disponibles suivant la mobilité des personnes. Le Wi-Fi (Wireless Fidelity) est notamment disponible dans certains espaces publiques et bon nombre d'entreprises ont leur propre réseau interne.

Devant la multitude des standards radio existants et la mobilité accrue des personnes, la naissance d'appareils ayant la capacité de s'adapter automatiquement à l'environnement dans lequel il se trouve, ou de réaliser une fonctionnalité voulue (téléphone, télévision, GPS, navigateur web, etc.) est nécessaire. Or chaque standard radio a ses propres caractéristiques, ce qui signifie que les parties matérielles et logicielles sont spécifiques au système. C'est pourquoi la gestion de multiples standards requière l'utilisation d'une plate-forme matérielle adaptative et extensible (scalable) pour gérer ceux existant et ceux à venir. Une solution sur laquelle les

chercheurs travaillent depuis quelques dizaines d'années est la **Radio Logicielle**.

La Radio Logicielle

La Radio Logicielle (RL) est apparue lorsque l'armée américaine, dans les années 70, s'est intéressée à l'interopérabilité entre diverses bandes de communication (HF, VHF, UHF ...), chaînes de communication (AM et FM) et formats de données militaires. Elle a commencé ses recherches sur une technologie Radio Logicielle avec le projet SPEAKeasy [6]. Ensuite le concept est apparu dans le domaine public grâce au professeur Joe Mitola III qui, par ses travaux, a présenté l'architecture de la Radio Logicielle, où les fonctions radio sont implémentées en logiciel [7]. Cette architecture permet de reconfigurer des ondes radio programmables (type de modulation et largeur de bande que l'on peut modifier) afin de s'adapter aux différents standards de communication actuels et à venir, grâce à son aspect reconfigurable. La Radio Logicielle idéale ne reste cependant qu'un concept, car sa réalisation nécessiterait une numérisation de la chaîne radio (présentée figure 2) juste après l'antenne. Or à l'heure actuelle, les propriétés des convertisseurs analogique/numérique existant ne le permettent pas (14 bits à 800Méch/s pour les plus performants [8]). Il est aujourd'hui nécessaire de passer par une numérisation en fréquence intermédiaire (à 440 Mhz pour le GSM). C'est pourquoi on utilise les termes de "Radio Logicielle Restreinte" (RLR) ou de Software Defined Radio (SDR) .

La formalisation de cette technologie s'effectue à travers un forum qui rassemble les différents acteurs de ce domaine pour partager les avancées réalisées. Il s'agit du SDR forum (SDRF), constitué d'un regroupement d'organisations effectuant des recherches dans le domaine de la SDR [9]. Les travaux présents dans ce mémoire y ont été présentés à plusieurs reprises. Le SDRF présente la Radio Logicielle comme une radio fournissant : un contrôle logiciel des standards de communication (techniques de modulation), des opérations large bande ou bande étroite, des fonctions de sécurité, des paramètres de formes d'onde des standards actuels et à venir. Cette définition a été approuvée par l'Union Internationale des Télécommunications ITU chargée de coordonner et de normaliser les réseaux et services mondiaux de télécommunication. Plusieurs autres définitions existent en fonction de la manière de considérer les différentes réalisations de la Radio Logicielle [10]. Voici quatre réalisations possibles :

- la Radio Matérielle : Toute la chaîne radio est réalisée en matériel. La seule manière d'implémenter une nouvelle chaîne est d'utiliser des switchs ou de changer manuellement la carte,
- la Radio Contrôlée par Logiciel : Seules les fonctions de contrôle sont réalisées de façon logicielle et peuvent donc être modifiées. Il est impossible de modifier le type de modulation ou la bande de fréquence sans modifier le matériel,
- la Radio Logicielle Restreinte : Ce système fournit une plage de fréquence configurable contrôlée de manière logicielle sans modifier le matériel. Les antennes sont séparées

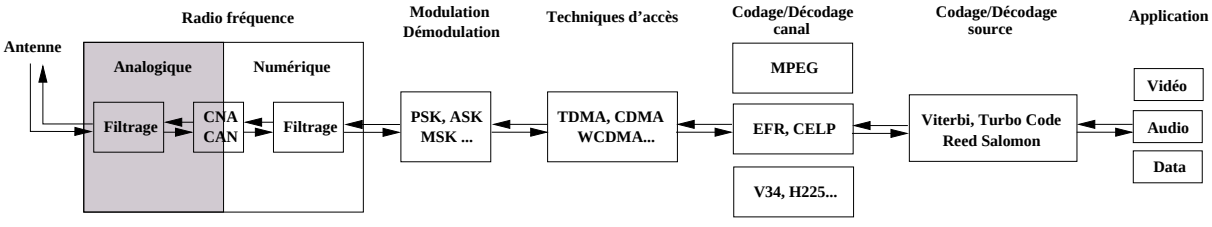


FIG. 2 – Chaîne radio

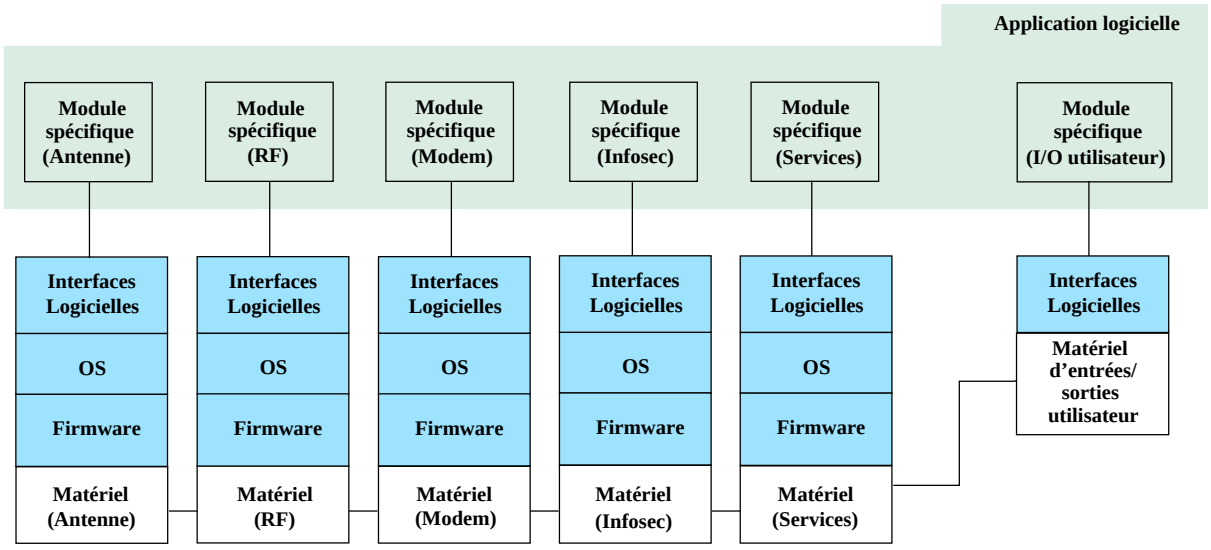


FIG. 3 – Implémentation typique de l'architecture "libre" matérielle/logicielle du SDRF

en avant du filtrage large bande, de l'amplification et des convertisseurs analogique-numérique au niveau de la chaîne de réception. C'est l'inverse au niveau de la chaîne de transmission. Si l'on veut pouvoir couvrir un ensemble de fréquences de différents standards il faut basculer sur l'antenne appropriée couvrant la gamme de fréquences désirées. Elle peut stocker différentes configurations, où se reconfigurer par voie aérienne (*over-the-air*),

- la Radio Logicielle Idéale : Ce système reprend la Radio Logicielle Restreinte mais en supprimant l'amplification analogique afin d'obtenir une numérisation dès la sortie de l'antenne,
- la Radio Logicielle "Ultime" également nommée "Radio Cognitive" ou "Radio Intelligente". Encore utopique, ce concept voudrait un composant de taille réduite, consommant peu, sans antenne externe, capable de s'intégrer facilement et de fournir l'information désirée dans le bon format. Un composant qui s'auto-adapterait à l'environnement, se reconfigurerait au moyen de signaux de contrôle qui lui parviendraient via son connecteur. Il serait pourvu de ressources de calcul et de mémorisation importantes afin de gérer une large gamme de services pour l'utilisateur.

Les caractéristiques de cette technologie sont à l'image du potentiel offert. L'architecture de la Radio Logicielle est complexe, car elle réclame une plate-forme matérielle **flexible**, qui puisse être reconfigurable. Elle se veut également **extensible**. Il doit être possible de venir configurer des nouveaux standards de communications et y ajouter des fonctionnalités supplémentaires (nouveaux services, nouvelles applications offertes aux utilisateurs). Elle doit être aussi **indépendante** du matériel, dans le sens où il ne faut pas changer de matériel pour avoir des standards de communication différents, les modifications sont effectuées de manière logicielle. Elle demande également une forte capacité de traitement, de l'ordre de quelques giga opérations par seconde (50 à 3000 MOPS pour le filtrage), pour réaliser tous les calculs nécessités par les applications. En ce qui concerne sa **maintenabilité**, elle est assez aisée grâce à la configuration "tout" logicielle.

La Radio Logicielle est un domaine de recherche très actif depuis ces dernières années, beaucoup d'aspects ont été traités (ex : support matériel) tandis que d'autres restent encore à

définir (ex : l'architecture de communication logicielle (SCA)). Différentes solutions architecturales ont été explorées. Que ce soit des solutions en couches [11], des architectures traditionnelles ou orientées noyau à base de macro [12], elles gravitent toutes autour des mêmes vues, à savoir des objets ou composants, logiciels ou matériels, qu'il suffit de reprogrammer, de reconfigurer, et dont les configurations sont contenues dans des mémoires ou téléchargées (par voie aérienne ou bornes spécifiques). Le lecteur intéressé pourra se référer aux articles cités pour plus de détails. Une implémentation typique de l'architecture de tels systèmes décrite par le SDR Forum est présentée figure 3 [10]. Cette implémentation en couches (matérielle, middleware, logicielle) permet de dissocier totalement l'application de l'architecture. Chaque partie matérielle possède ses propres API (Application Programming Interfaces) fournissant l'ensemble des primitives utiles à l'application pour utiliser au mieux les ressources matérielles dont elle dispose. L'application logicielle s'exécute donc, via un système logiciel (firmware) sur du matériel.

Mise en œuvre de la Radio Logicielle

La mise en œuvre d'un système de Radio Logicielle réclame une ossature logicielle, reposant sur une architecture matérielle riche (en terme du nombre de ressources) et diversifiée. L'évolution technologique de ces dernières années a permis d'augmenter la complexité, et donc la puissance, des composants électroniques. En effet la réduction croissante de la taille physique des transistors (65nm pour les pentium D 9XX [13]), alliée à la réduction de la consommation des puces (même si cela reste à améliorer), ont permis la naissance d'un nouveau concept de composants que sont les systèmes sur puce ou System On Chip (SoC). Le terme "système" dans un SoC désigne, une application complète fonctionnant sur un seul et même composant, **et** l'ensemble des ressources matérielles de natures différentes (processeurs, mémoires, ressources de communication, FPGA), utiles à son bon fonctionnement. Jusqu'alors composées de multiples composants discrets, les architectures matérielles des systèmes à venir auront pour principal support d'exécution les SoC (tel le MP211 de NEC présenté ultérieurement dans le document). Les derniers FPGA de chez Xilinx (Virtex-5 LX) ou Altera (Stratix II GX) peuvent également être considérés comme des SoC reconfigurables, de part l'intégration de processeurs (matériels et/ou logiciels), de bancs mémoires, de ressources de communication, et la reconfigurabilité qu'ils proposent. Or, les applications Radio Logicielle nécessitent de telles ressources pour exécuter les différents filtrages, modulation/démodulation, codage/décodage, etc., d'une chaîne radio telle que présentée sur la figure 2. L'architecture matérielle hétérogène offerte par les SoC répond donc aux exigences d'implémentation de la Radio Logicielle.

Cette évolution technologique a été accompagnée de modifications dans le processus de conception des systèmes. En effet les contraintes de temps de mise sur le marché (Time to Market) des produits, sont restées fortes pour limiter les coûts et rester compétitif. Aujourd'hui le cycle de conception d'un SoC dure 12 mois [14]. L'important est donc d'éviter les erreurs de conception ou de les détecter au plus tôt dans le cycle de conception afin de ne pas relancer une étape du processus. L'importance des applications, associées aux pressions temporelles, ont amené les concepteurs à ré-utiliser des fonctions régulièrement présentes dans différentes conceptions. Ces fonctions où blocs fonctionnels, plus ou moins complexes, déjà développés en matériel ou logiciel sont les IP (Intellectual Property).

Pour faciliter le développement d'application RL, des programmes de défense américains : le JTRS (Joint Tactical Radio System) et le JPEO (Joint Program Executive Office), rassemblent un consortium d'entreprises chargé de développer une architecture pour ces composants ra-

dio. Ces deux programmes aident le SDR Forum et l'OMG (Object Management Group) [15] à standardiser une architecture, retenue comme structure logicielle des systèmes Radio Logicielle. Cette architecture nommée SCA (Software Communication Architecture) [16] spécifie les interactions entre les différents composants matériels et logiciels d'une radio et fournit les commandes logicielles de contrôle.

Toutes ces modifications nécessitent que les méthodologies de conception s'adaptent en conséquence pour rester performantes.

Contexte des travaux de thèse

A la fin des années 90, le secteur des télécommunications a subi un véritable essor dû à l'explosion de la téléphonie mobile et la démocratisation de l'Internet. Mais à partir de 2001, la situation de ce secteur s'est dégradée avec l'effondrement du marché arrivé à saturation. Les opérateurs avaient prévu une hausse continuelle de la demande sur la même lancée et avaient investi en conséquence. Dans ces circonstances, la réaction des industriels a été de baisser le régime des chaînes de production et de réduire l'investissement de leurs services de Recherche et Développement. En France même si le GSM est un standard mondial, le retard sur la mise en place des infrastructures des terminaux nouvelle génération (UMTS) comparativement aux pays étrangers, notamment le Japon, est mesurable. Dans cette crise du secteur, marquée par un fort retrait de l'innovation des services mobiles, il a fallu réagir et anticiper les terminaux et services à venir, afin de rester compétitif. Le Réseau National de Recherche en Télécommunications (R.N.R.T.) a lancé un appel à projet dans ce sens en 2002. Il garde les mêmes directives depuis, en mettant l'accent sur le développement de l'ingénierie des modèles (MDE : Model Driven Engineering). Une des orientations proposées est l'anticipation de la diversité des terminaux et objets communicants. Cette anticipation passe par la maîtrise de la conception matérielle et logicielle de ces objets. Le projet Adéquation Architecture - Application Système (A3S), dans lequel s'inscrit une partie des travaux présentés dans ce mémoire, propose d'apporter une méthodologie de conception basée sur un langage extensible de modélisation (UML). Ce projet pré-compétitif, labellisé par le RNRT, rassemble des partenaires appartenant aux deux mondes, logiciels et matériels. Il regroupe Thales Communication S.A (Colombes) qui apporte son expertise dans le domaine de la RLR, Mistubishi Electric ITE (Rennes) qui apporte son savoir faire en matière de réalisation de systèmes de téléphonie mobile, la société Softeam (Paris-Rennes) experte dans la modélisation UML, et le Laboratoire d'Electronique des Systèmes TEmps Réel (LESTER)(Lorient) qui développe des méthodologies pour la conception sous contraintes de systèmes et circuits électroniques. L'objectif de ce projet était la réalisation d'un prototype d'outil logiciel avec UML, permettant à un concepteur d'applications Radio Logicielle de tester différents scénarios d'implémentation des composants de son application sur un modèle de plate-forme SoC, afin de valider ses choix. Le langage UML proposait déjà quelques solutions de modélisations de systèmes temps réel, sous la forme de modèles. Mais ceux-ci étaient incomplets et ne permettaient pas de tout exprimer. En revanche, malgré ces modèles, aucun logiciel permettant de les exploiter, dans le cadre de la conception de SoC, n'avait encore été développé. Il est à souligner que le langage UML était alors en pleine transition avec la standardisation en cours d'UML 2.0 [17] succédant à UML 1.4. Cette évolution a été néanmoins prise en compte dans le projet A3S, qui intègre les modifications d'UML 2.0 à UML 1.4, afin de pouvoir migrer facilement vers UML 2.0 par la suite.

Problématique abordée

Les possibilités architecturales des plate-formes matérielles offertes au concepteur sont multiples et variées. NEC propose par exemple un composant dédié aux applications de la téléphonie mobile : le MP211 [18]. Il comprend 3 processeurs ARM, un DSP, 4 zones de mémoire RAM auxquels se rajoutent 8 blocs supplémentaires 512Kb de RAM, ainsi que de la logique servant aux accélérateurs graphiques etc. Les possibilités offertes en terme d'application sont nombreuses (réception numérique terrestre, vidéophonie, téléphonie etc.). De plus, les applications, notamment Radio Logicielle, intègrent la reconfiguration et la modification des fonctions qui les composent. Le concepteur doit donc pouvoir tester aisément et tôt dans le cycle de conception, diverses possibilités d'implémentation des composants logiciels d'une application, sur une ou plusieurs plates-formes matérielles ciblées. La problématique est donc la suivante : peut-on proposer aux concepteurs, un environnement de modélisation leur permettant de disposer d'éléments caractérisés afin de concevoir, à la fois les applications logicielles qu'ils souhaitent mettre en œuvre, les plates-formes matérielles dont ils disposent, et de tester leurs choix d'implémentation sur chacune, sans aller jusqu'au bout du processus de conception ? Cet environnement peut-il supporter une approche de conception d'architecture basée sur des modèles ? Trop souvent les outils actuels ne visent qu'un seul type d'application ou se concentrent sur des architectures homogènes. Ils ne tiennent pas toujours compte de l'éventuelle utilisation d'un système d'exploitation. Or il est important pour un concepteur disposant de plusieurs plates-formes, de connaître rapidement la faisabilité d'une réalisation logicielle, quelle qu'elle soit, sur une plate-forme matérielle, en explorant les solutions d'implémentation. Cette garantie lui permet de s'engager dès le départ vers une solution capable de satisfaire l'ensemble des contraintes du système.

Contributions

Dans le monde de la Radio Logicielle, les concepteurs, utilisent dans leurs flots de conception, plusieurs outils et langages différents pour modéliser et spécifier leurs systèmes Radio Logicielle. Les travaux décrits dans ce document ont permis la réalisation d'un prototype logiciel autour d'un langage de modélisation extensible (UML) pour modéliser et spécifier de tels systèmes. Plus précisément, une sémantique particulière, adaptée au niveau de représentation des SoC, a été développée à travers le profil UML A3S. En outre, un travail de réflexion concernant le niveau de représentation des architectures matérielles et logicielles choisies a été mené. Ce travail a donné suite à l'identification précise des différents éléments matériels et logiciels et de leurs caractéristiques respectives, qui servent à modéliser le système. L'ensemble de ces éléments a permis d'établir le profil A3S, qui prend en compte la spécificité de la Radio Logicielle. Ce profil enrichit le profil Radio Logicielle [19] en cours de standardisation par l'OMG, et a été de nouveau enrichi après la clôture des travaux liés au projet A3S par l'ajout des systèmes d'exploitation au profil. De plus, un ensemble de règles, ainsi qu'un outil d'analyse permettent au concepteur d'identifier les erreurs commises, et l'instruisent sur la faisabilité du système qu'il désire implémenter. Les règles établies localisent les incohérences dans les valeurs que le concepteur a renseignées, ainsi que les erreurs ou oublis commis lors de la modélisation des architectures matérielles et logicielles de ses systèmes. L'outil d'analyse traduit ce qui a été modélisé et renseigné afin de répondre sur l'ordonnançabilité et la faisabilité du système. Ce prototype a été la première réalisation concrète des concepts MDA présentés au SDR Forum en 2004 [20], 2005 [21].

Plan du mémoire

Ce document présente dans une première partie une exploration des différentes méthodologies de conception existantes utilisées pour concevoir les systèmes embarqués temps réel. Cette partie porte une attention particulière à l'utilisation du langage UML dans la conception des SoC et la prise en compte de l'aspect temps réel par ce même langage.

La seconde partie illustre la façon dont on peut concevoir avec le langage UML. Il expose notre propre méthodologie basée sur UML. Le lecteur peut suivre le flot de conception A3S, découvrir le modèle développé ainsi que les outils à utiliser.

La suite du document traite des systèmes modélisés qui valident la méthodologie. Dans un premier temps, la troisième partie décrit une application Radio Logicielle. Une chaîne UMTS (partie émetteur et partie récepteur) est présentée, modélisée et les résultats de faisabilité commentés. Dans un second temps, la quatrième partie traite de systèmes qui intègrent et utilisent un système d'exploitation. Ces expériences confirment, de part les cas d'utilisation et les résultats retournés, la validité et la pertinence du modèle de système d'exploitation. Enfin la conclusion retrace les travaux effectués et ouvre la voie sur une poursuite des travaux intégrant les évolutions récentes et à venir. Il est en effet possible d'extrapoler sur l'intégration d'un modèle d'intergiciel (middleware).

Chapitre I

Méthodologies de conception et langage UML

Les successions d'innovations rapides de ces 25 dernières années dans le domaine électronique, sont le résultat de méthodologies de conception en adéquation avec les technologies des composants physiques qui supportent ces applications. Une méthodologie de conception, dans le domaine de l'informatique et de l'électronique, est l'ensemble des méthodes et des outils, utilisés pour mener à bien la conception de systèmes électroniques temps réel. Ce premier chapitre présente donc après avoir introduit ce qu'est un SoC, les différentes méthodes et langages dont le concepteur dispose pour traduire un système. Il s'attarde notamment à promouvoir les possibilités offertes par le langage UML dans la conception des SoC, en proposant un état de l'art spécifique. Cet état de l'art s'intéresse également, à la fin de ce chapitre, à la prise en compte en UML de l'aspect temps réel (via les systèmes d'exploitation), partie intégrante des systèmes électroniques actuels.

I.1 Les méthodologies de conception des SoC

I.1.1 Que se cache-t-il derrière un SoC ?

L'objet de l'étude concerne la modélisation des systèmes monopuces (SoC) dans un cadre défini (Radio Logicielle) mais non restrictif. Il est donc important de définir ce que les chercheurs et les industriels de l'électronique appellent un SoC. Dans la littérature concernant les SoC, il semble qu'aucune définition formelle n'ait été donnée. Le concept de SoC découle des progrès technologiques des puces électroniques. La réduction de la taille des transistors engendre en effet, une concentration supérieure de transistors sur une même surface de silicium. Cette concentration autorise désormais la présence de plusieurs composants de natures différentes sur une unique puce. Une carte électronique, autrefois composée de multiples composants discrets (processeur, mémoires, ASIC, etc...), se voit désormais dotée d'un composant central intégrant tous les autres. C'est pourquoi le SoC est généralement présenté comme un ensemble de blocs hétérogènes présents sur la même surface de silicium. Pour être plus précis, ces blocs peuvent être des IP matérielles pré-vérifiées qui communiquent au moyen de protocoles complexes [22].

L'architecture matérielle d'un SoC ressemble donc à celle présentée sur la figure I.1. Elle se compose de l'ensemble des composants habituels, nécessaires aux exigences d'un système complexe. Le SoC combine à la fois des parties numériques et analogiques. Il dispose de ressources

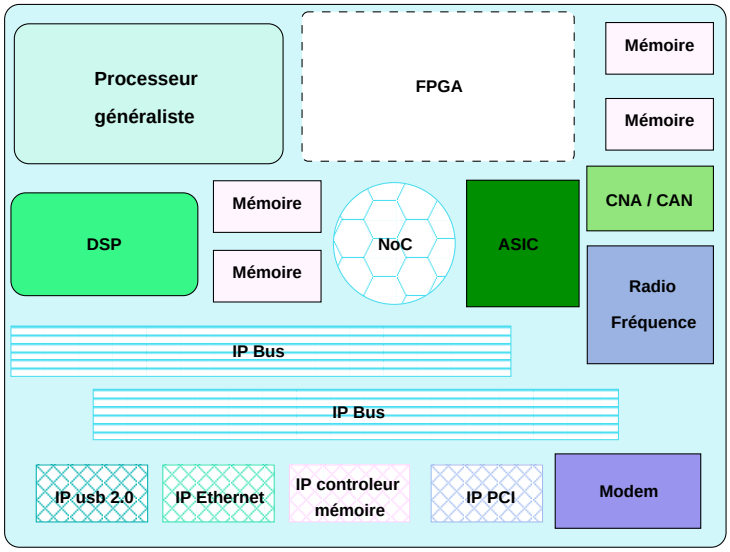


FIG. I.1 – Architecture générique d'un SoC

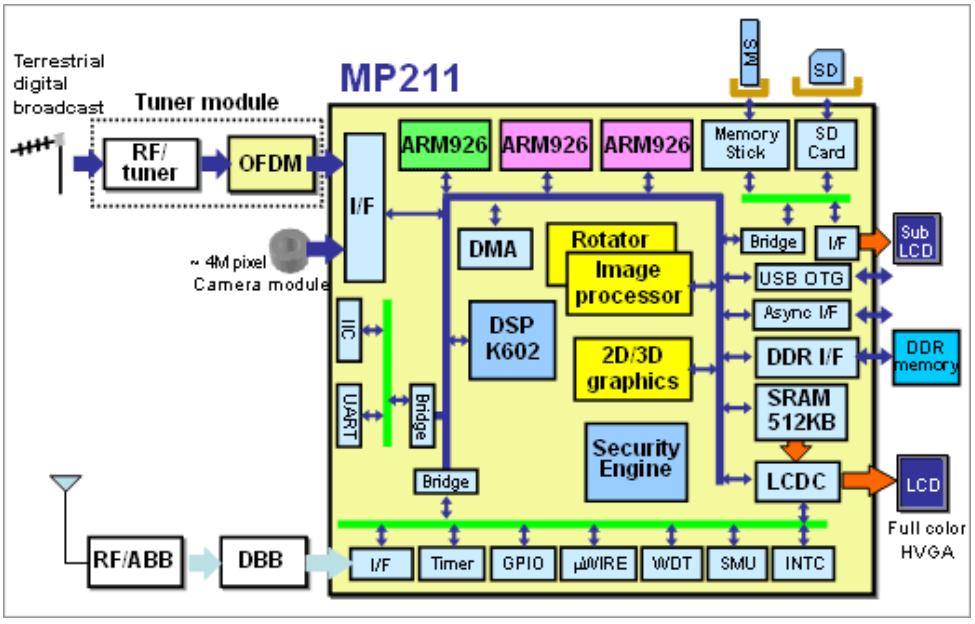


FIG. I.2 – Architecture matérielle du MP211

de calcul conséquentes, spécifiques (DSP, ASIC) ou non (processeur généraliste), auxquelles sont adjointes des ressources de mémorisation proportionnées (en taille et en nature). L'hétérogénéité ne pourrait être complète sans la présence de composants reconfigurables (FPGA) offrant de la flexibilité au système. Tous ces composants ne coexisteraient pas, s'il n'y avait une structure de communication adaptée et/où adaptable entre eux (ex : un réseau sur puce (NoC)). Le SoC peut également intégrer des capteurs, des micro-actionneurs et des composants optiques, on parle alors de Microsystèmes Opto-Electro-Mécaniques (MOEMS en anglais). Même si le SoC devient le composant majeur d'une carte électronique, il est néanmoins amené à échanger avec l'extérieur (capteurs externes, clavier, écran, etc.), c'est pourquoi il dispose d'interfaces d'entrées/sorties (E/S). Pour donner un exemple concret, la figure I.2 détaille l'architecture du processeur MP211 de NEC, spécialisé pour les applications de la téléphonie mobile [18].

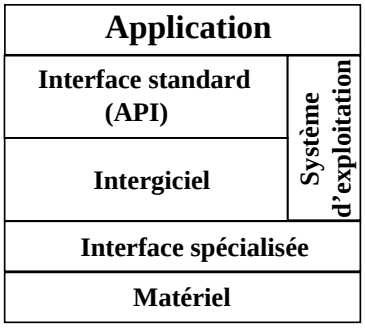


FIG. I.3 – Architecture globale d'un SoC

Devant la complexité et la granularité des blocs utilisés, Gupta [23] présente la notion de SoC comme une conception au niveau système d'une implantation microélectronique. Il traduit bien là, la dimension "système" de l'intégration matérielle. Un système ne se restreint cependant pas, à une architecture matérielle. Le matériel n'est que le support d'exécution d'une application. Dans le cas des SoC, les applications sont aussi complexes que les architectures sur lesquelles elles reposent. Cette complexité se traduit par un ensemble de couches intermédiaires séparant l'application logicielle de l'architecture matérielle. L'architecture générale d'un système monopuce présentée figure I.3, montre les différentes couches matérielles, matérielles/-logicielles, et logicielles, séparant la couche physique de l'application. Les SoC sont issus, en partie, de l'assemblage de blocs IP, favorisant ainsi la ré-utilisabilité de blocs pré-conçus. Un des challenges de la conception de ces systèmes réside dans l'interfaçage des différents blocs matériels et les procédures de communication (transfert des signaux de contrôle et de données) durant l'exécution de l'application. Ce sont ces deux problématiques qui expliquent la présence des différentes couches d'interfaces (spécialisée, intergiciel, OS, API) entre le matériel et le code de l'application. Par exemple, pour qu'une application puisse tirer le meilleur parti des ressources matérielles disponibles, notamment des processeurs programmables, ceux-ci doivent pouvoir fournir des primitives de pilotage logicielles qui lui sont propres (driver, firmware). La figure I.4 illustre l'architecture globale mise en œuvre dans le cas du MP211 [24]. Des drivers permettant le pilotage du matériel sont fournis, ainsi le système d'exploitation dispose des routines logicielles pour utiliser les différentes ressources et donc abstraire le matériel vis à vis de l'application. Les différentes applications dédiées à l'affichage, au traitement d'images, et aux traitements de l'information, peuvent, grâce au *middleware* (intergiciel) et à l'OS, se partager de manière transparente les différentes ressources disponibles sur le MP211. L'intergiciel aide à la transparence car il propose des abstractions logicielles à l'OS pour simplifier le développement

des applications réparties [25].

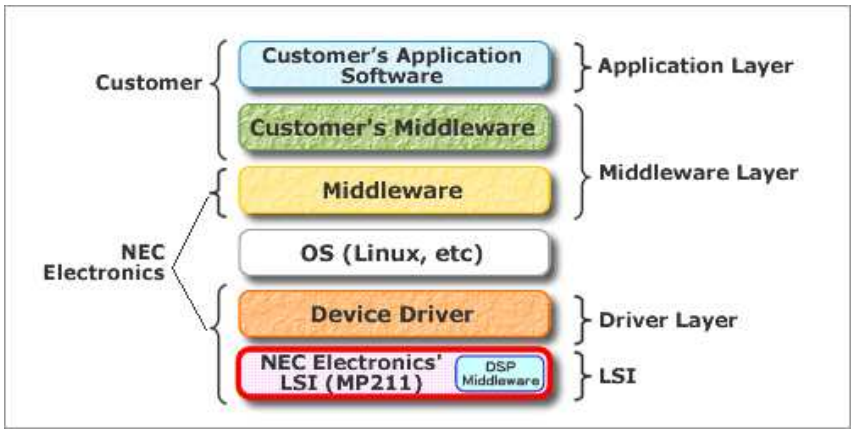


FIG. I.4 – Vue générale de l'architecture du MP211

Le SoC est donc un système complexe qui rassemble des compétences métiers différentes. Une définition d'un SoC appropriée serait de le considérer comme un composant matériel constitué de ressources matérielles et logicielles hétérogènes suffisantes pour l'exécution d'applications complètes. Il peut être considéré à des niveaux d'abstraction différents et donc être représenté par des modèles différents. La conception de ces systèmes ne peut être résolue par des méthodes classiques où la conception du matériel précède celle du logiciel. Car dans le cas des SoC, ces méthodes sont incompatibles avec les contraintes de temps de mise sur le marché actuelles. La solution repose sur des méthodes de conception conjointes du matériel et du logiciel (codesign) liées à la ré-utilisation de design de haut-niveau (IP).

I.1.2 Comment concevoir un SoC ?

L'aboutissement d'un produit, quel qu'il soit, est issu d'une idée de départ. Cette idée mûrit, se développe jusqu'à ce qu'un produit puisse être défini. Pour arriver au produit final un certain nombre d'étapes ordonnées doivent être respectées et franchies. Ce processus s'applique donc naturellement dans le cadre du développement de composants électroniques. Si l'on s'intéresse aux cas des composants numériques uniquement, jusqu'aux années 60, la technologie ne permettait d'obtenir, que des circuits MSI (Medium Scale Integration) contenant quelques portes logiques (de 10 à 100). Ce n'est qu'à partir de 1970 que sont apparus les premiers microprocesseurs 8 bits, puis les microcontrôleurs (VLSI : Very Large Scale Integration), utilisés dans des petits systèmes temps réel monocircuits. Ces évolutions, ajoutées à la diversité des composants proposés, ont tout d'abord amené le concepteur d'un système à prendre en compte la possibilité de pouvoir réaliser n'importe quelle fonction en programmant un composant qui le permet (microprocesseur). L'architecte matériel continuait toujours de réaliser les composants matériels tandis qu'un spécialiste logiciel programmait les applications. L'apparition des composants configurables et reconfigurables par programmation (EPLD, CPLD, FPGA) ont ensuite ouvert le champs des possibilités, en permettant l'implantation physique d'une fonction par configuration de blocs matériels (cellules logiques + bascules D). A ces blocs de faibles granularités, s'ajoutent désormais dans les FPGAs, des processeurs physiques et synthétisables (respectivement le PowerPC et le processeur MicroBlaze dans un Virtex-II pro de chez Xilinx [26]). Le système peut alors potentiellement tenir sur un seul composant.

Les méthodes de conception se sont toujours adaptées, avec un temps de retard, pour intégrer au mieux les nouvelles contraintes d'utilisation des nouveaux composants, et contenir le temps de conception. Tant qu'il y avait peu de transistors, les dessins des masques de gravure se faisaient manuellement. Lorsque les circuits MSI sont apparus, les dessins ont été automatisés via des logiciels informatiques compilant des représentations schématiques d'association de portes (Max+ d'Altéra [27]). Les systèmes ont continué de se complexifier avec les processeurs et les FPGAs, et les méthodes de conception ont élevé le mode de représentation des éléments du système. Du niveau physique, les systèmes ont été représentés au niveau logique, puis RTL (Register Level Transaction), puis comportemental et structurel, et désormais ils se représentent directement au niveau système. Cette élévation du niveau d'abstraction n'a été permise que par l'accompagnement de l'utilisation d'outil de génération automatique de masques (Cadence [28]), de synthèse logique (Design Compiler [29], les outils Synplify [30]), de synthèse de haut-niveau (CatapultC [31], Defacto [32], Spark [33], GAUT [34]), qui réduisent le temps de conception tout en garantissant les performances des systèmes. Cependant ces outils nécessitent généralement de nombreux interfaçages, dus aux différents formalismes de représentation (en entrées et en sorties) qu'ils utilisent. Chacun d'eux intervient à des étapes contigües du flot de conception. Le flot de conception d'un SoC, représenté sur la figure I.5, précise l'ensemble des étapes, et la manière dont elles sont réalisées à l'heure actuelle, nécessaires pour l'obtention des masques qui serviront à produire le circuit final.

La première étape du concepteur est de définir et de traduire les besoins du système qu'il souhaite fabriquer afin de dégager les contraintes globales de fonctionnement. Il identifie les traitements qu'il doit mettre en œuvre et les ressources matérielles qu'il pense judicieux d'utiliser pour réaliser son circuit.

La seconde étape est la traduction du fonctionnement du système via la définition du séquencement des traitements et des transmissions des signaux de données et de contrôle. Elle s'effectue aux moyens de modélisations concernant l'application du système, comme dans un flot de conception classique, mais aussi par des modélisations de l'architecture matérielle. Cette étape peut subir plusieurs raffinements, en fonction de la précision souhaitée, des contraintes connues, et du type de modélisation utilisée. Dans les deux cas, avec et sans raffinement, le concepteur s'attarde à spécifier les différentes caractéristiques et contraintes imposées aux éléments de représentation logicielle et matérielle dissociés.

Une fois ces deux étapes réalisées, l'étape d'exploration architecturale, visant à obtenir la meilleure adéquation d'implantation matérielle/logicielle des différents traitements, peut avoir lieu. Ce partitionnement peut être manuel ou automatisé. Dans le second cas il s'effectue par le biais de calcul de métriques. Ces métriques matérielles (dynamisme (au sens de la reconfiguration, programmation)), scalabilité, etc.) et logicielles (parallélismes, orientation mémoire, etc.) associées à des bibliothèques d'estimation, détermine une implantation optimale aux vues des spécifications exprimées [35]. Ces choix effectués, un ensemble d'analyses, de simulation et de co-simulation, reposant sur des modèles comportementaux peuvent être réalisés. Les résultats renseignent alors sur les performances et la fonctionnalité effective du système. Ils identifient les erreurs commises et permettent au concepteur d'effectuer les modifications nécessaires en amont du flot.

L'ultime étape avant l'obtention des masques de gravure et le passage en fonderie, est la validation du système après la synthèse du système pour les parties matérielles, et après compilation pour les parties logicielles.

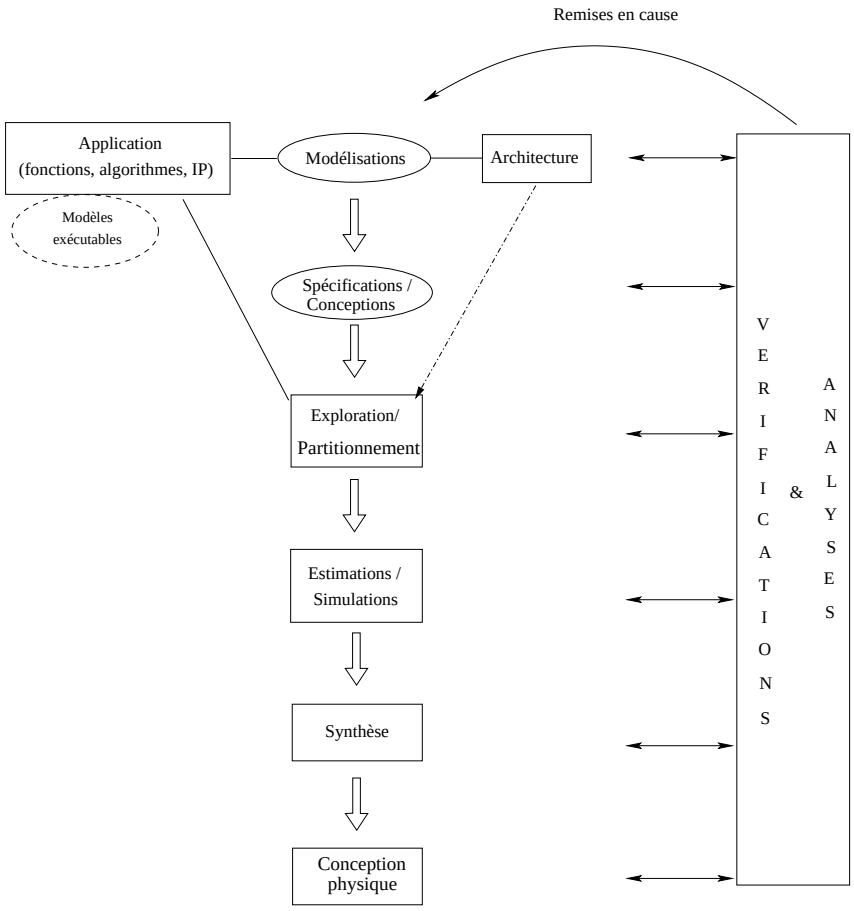


FIG. I.5 – Flot de conception d'un SoC

I.1.3 Méthodologies et langages de conception au niveau système

Chaque avancée technologique des composants électroniques est suivie du développement de nouvelles méthodologies de conception adaptées. Ces évolutions de conception technologique, appelées *Linchpin technologies*, nécessaires à l'amélioration des gains de productivité, élèvent le niveau d'abstraction des représentations (portes, RTL, IP)[1]. L'avènement des systèmes sur puce remplace petit à petit les méthodologies de conception basées sur des blocs (*Block-Based Design* (*BBD*)) (qui ont supplanté celles dirigées par le temps (*Time-Driven Design* (TDD)) et l'espace (*Area-Driven Design* (ADD)) par des méthodologies de conception basées sur des plates-formes (*Platform-Based Design* (PBD))[1]. Cette évolution, retracée sur la figure I.6, illustre la granularité croissante des composants de base des architectures matérielles, qui augmente les possibilités de ré-utilisation de blocs pré-conçus offerts aux concepteurs pour la réalisation de ses systèmes.

Au départ la méthodologie ADD se focalisait sur la minimisation de l'espace de silicium que devait occuper un système. Elle concernait les ASIC de petite taille (<250 000 portes) et s'appuyait sur les tableaux de Karnaugh, algèbre de Boole et la minimisation booléenne pour parvenir à limiter la surface occupée. La logique se complexifiant, la conception menée par le temps (TDD) a pré-value sur la précédente. Des outils de layout ont fait leur apparition renvoyant des estimations précises de performances temporelles et de surface. L'utilisation de compilateurs et la vérification fonctionnelle au niveau RTL prédisent et analysent à plus haut niveau les performances prévisibles. La complexité toujours croissante a alors amené les concepteurs à

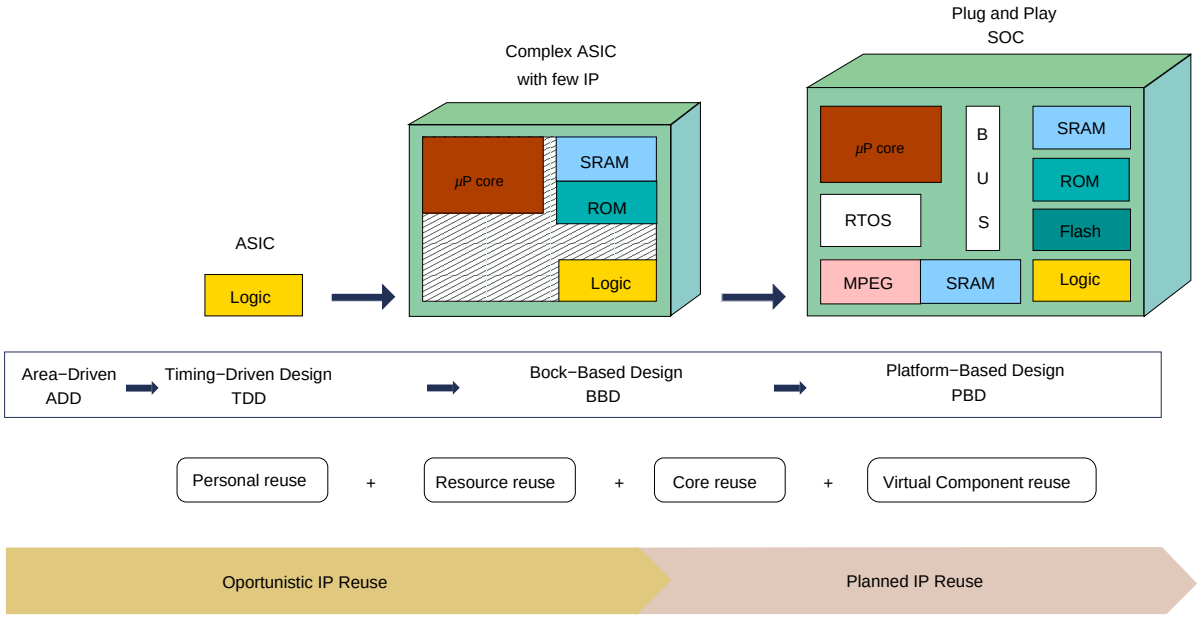


FIG. I.6 – Evolution des méthodologies de conception [1]

ré-utiliser des blocs fonctionnels déjà conçus au sein d'une méthodologie BBD. Ils intègrent les blocs de manières hiérarchiques en respectant une implantation *top-down*. Le comportement du système est modélisé à haut-niveau. Il est analysé par des méthodes de codesign pour obtenir le meilleur compromis d'implantation logicielle/matérielle. Ce partitionnement est réalisé sur des blocs fonctionnels de niveau RTL. Les outils d'analyse de haut niveau simulent le comportement du système et le valide à partir des résultats d'outils de vérification type HDL (langage de description matériel) avec lesquels ils s'interfacent, allant jusqu'à la synthèse RTL. La répartition physique des blocs se fait en tenant compte de leur interfaçage tout en optimisant les coûts en surface, en temps, et en puissance. La dernière méthodologie utilisée, basée sur les plates-formes, accentue la réutilisation des conceptions déjà réalisées à travers l'utilisation de blocs pré-conçus déjà éprouvés, donc prédictibles, possédant des interfaces standardisées. Ces blocs fonctionnels sont très peu modifiés contrairement à ceux de la méthodologie BBD qui demandent à être adaptés aux besoins des applications spécifiques. La complexité des ASIC traités par la méthode PBD va au-delà du million et demi de portes supportées par la BBD. Les *"Linchpin technologies"* mises en place pour la méthodologie PBD s'organisent autour de deux activités : (1) la création des blocs et (2) l'intégration sur la puce. La première (1) reprend les méthodologies TDD et BBD, avec des blocs possédant des interfaces standardisées aisément ré-utilisables dans d'autres conceptions. La seconde (2) se focalise sur la conception et la vérification de l'architecture et de l'interfaçage entre blocs. La PBD se concentre autour d'une architecture de bus standardisée tout en minimisant le nombre de création d'interfaces et de modifications de blocs. Ce sont des outils de conception haut-niveau utilisant des méthodes de co-design qui permettent la modélisation au niveau système (sélection des IP), la réalisation du partionnement matériel/logiciel, et la validation en précisant les modèles de composants virtuels (VC : composant VLSI autonome prêt à être intégré, également dénommé IP). Les outils de layout s'intéressent au placement du bus et des blocs pour bien les intégrer suivant leurs relations. Des outils de validation se préoccupent des interfaces de toute nature à tous les niveaux de modélisation (inter bloc, bloc/bus, numérique/analogique etc.).

La modélisation PBD est la combinaison de deux efforts : conception "descendante" (*top-*

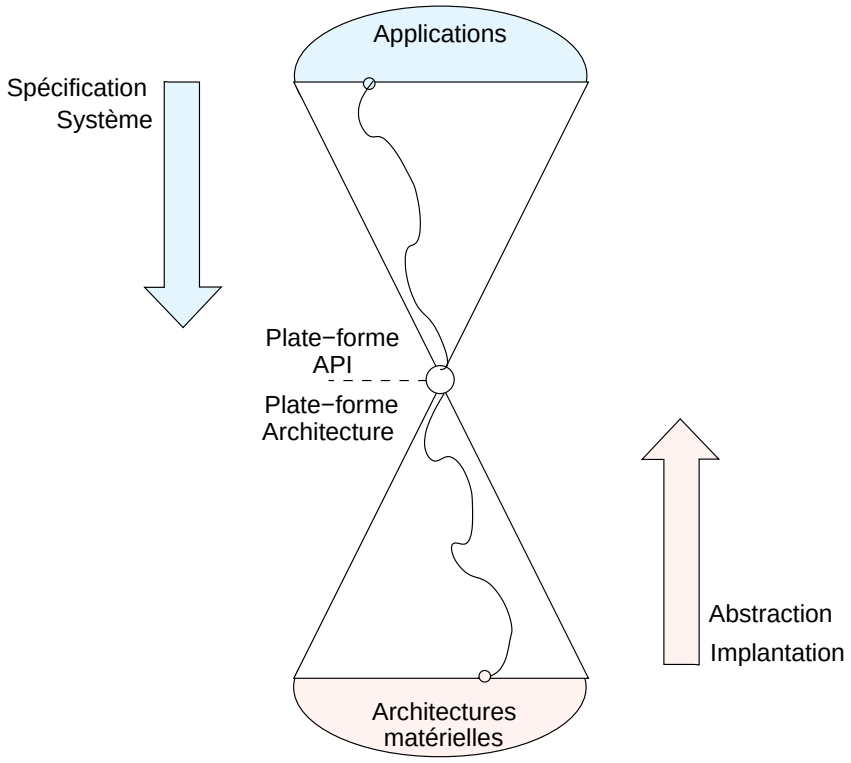


FIG. I.7 – Empilage de plates-formes

down) et conception "montante" (*bottom-up*), figure I.7. Le point de rencontre, *"meeting-in-the-middle"* est le résultat des raffinements successifs des spécifications du système, et de l'abstraction des implantations envisageables [36][37]. Le raffinement s'effectue par étapes où le système est représenté à des niveaux d'abstraction différents. Chacun de ces niveaux est considéré comme une **plate-forme** dotée d'informations suffisantes pour les couches inférieures. Ces informations permettent l'exploration de l'espace de conception qui précise et prédit les propriétés de l'implantation finale. L'empilage de plates-formes fait bien apparaître le découplage entre les processus de développement des applications et ceux dus à l'implantation de l'architecture.

Chaque *"Linchpin technologies"* s'appuie sur des langages et des modèles pour exprimer, décrire, spécifier, formaliser et analyser les systèmes électroniques temps réel embarqués. Beaucoup ne sont performants que dans un domaine applicatif spécifique. Si nous nous intéressons aux langages de modélisation et de spécification au niveau système, les parties matérielles et logicielles d'un système doivent pouvoir être spécifiées. Il existe trois approches de modélisation au niveau système possibles, la modélisation homogène, la modélisation hétérogène et une spécification multi-langages [38]. La modélisation homogène utilise un langage unique pour spécifier l'ensemble du système (partie matérielle et partie logicielle). La modélisation hétérogène utilise des langages spécifiques pour les parties matérielles et logicielles. La spécification multi-langages utilise des langages spécifiques pour les différentes sous-parties d'un système que peuvent se partager plusieurs équipes.

Les langages proposés pour la modélisation et la spécification au niveau système doivent permettre l'intégration de modèles hétérogènes et doivent pouvoir gérer différents niveaux d'abstraction. Ils peuvent être classés en plusieurs catégories [39][40] attachées à :

(1) la ré-utilisation des langages de description matérielle (HDL) existant, tel quel VHDL [41], Verilog [42], ou en leur apportant des extensions SystemVerilog[43],

(2) la ré-adaptation de langages de programmation logicielle (C/C++ [44], java [45]) pour disposer de moyens d'exprimer les caractéristiques du matériel, comme par exemple SystemC [46],

(3) la création de langage spécifique au niveau de modélisation comme Rosetta [47],

(4) l'utilisation des propriétés de langages standardisés qui ne sont spécifiques, ni au domaine de l'électronique, ni à un niveau de modélisation précis. C'est le cas du langage UML [48].

C'est cette quatrième catégorie qui va nous intéresser plus particulièrement en raison des impératifs de notre projet. Il s'agit pour nous de répondre aux préoccupations des systèmes de télécommunication (ici Radio Logicielle) notamment dans le domaine de leur conception et de leur développement. Les architectures Radio Logicielle se caractérisent par un découplage entre l'environnement d'exécution hétérogène (fonctions cablées, DSP, MCU) et les applicatifs (code des fonctions cablées où d'exécution sur processeur). Nous retrouvons ici la définition de plate-forme d'exécution hétérogène et la notion d'application où il faut vérifier à priori l'adéquation. Les spécialistes du domaine soulignent la nécessité d'avoir une suite d'outils de développement intéropérables suivant une méthodologie standardisée. Ils ne disposent en effet que de méthodologies et de processus de conception informels, utilisant des outils non spécifiques au domaine, et non intéropérables. Ils ne disposent pas non plus de langage commun.

Au vu des propriétés connues du langage UML (présentées dans la section I.2.1), nous pensons que c'est une solution aux préoccupations exposées. L'état de l'art de l'utilisation d'UML dans la conception des SoC, présenté ci-après, tend d'ailleurs à prouver l'intérêt des chercheurs à intégrer ce langage dans le processus de conception à haut-niveau des systèmes électroniques temps réel.

I.1.4 L'approche MDA

Le processus de conception d'un système ou d'une application est bien souvent le résultat d'un travail de plusieurs personnes d'une même équipe, ou de plusieurs équipes (comme c'est le cas dans les systèmes électroniques complexes d'aujourd'hui). Ces personnes ne disposent pas forcément des mêmes compétences et pourtant ils travaillent de manière indépendante ou non sur des parties du même projet, du même système. Ils doivent donc pouvoir se comprendre et interfacer leur réalisation avec celles des autres.

Le *Model Driven Architecture* (**MDA**) est une approche qui utilise des modèles pour le développement logiciel [49]. Son objectif est de réutiliser les modèles à travers les plates-formes technologiques. Il sépare les spécifications de l'application (opérations à effectuer) de l'infrastructure matérielle supportant l'application. C'est une approche qui permet le développement séparé des parties matérielles et logicielles d'un système. Elle offre la possibilité de choisir la plate-forme matérielle sur laquelle doit être implantée l'application et propose de transformer les spécifications de l'application (projection) sur l'architecture matérielle retenue. Le modèle applicatif est donc transformé en modèle intégré sur une plate-forme. Cette approche assure ainsi la **portabilité** des applications (modélisées de manière indépendante), l'**interopérabilité** des systèmes (possibilité de transformer les modèles pour les adapter, par exemple changer de langage) et la **réutilisabilité**, à la fois des applications modélisées qui peuvent être implantées sur d'autres plates-formes, et des plates-formes qui peuvent accueillir des applications différentes.

Le MDA définit 3 modèles d'abstraction différents, leurs intéractions, et leurs significations. Il s'agit du *Computation Independent Viewpoint* (CIM) qui ne s'intéresse qu'aux contraintes du système indépendamment de la structure de l'application, de sa réalisation et de son implantation (cahier des charges). Le second modèle est le modèle indépendant de la plate-forme (*Platform Independent Viewpoint* (PIM)) qui spécifie l'application par le biais d'informations non liées à son implantation sur une plate-forme déterminée. Le dernier modèle, (*Platform Specific Viewpoint* (PSM)) dérive les informations de la modélisation PIM et associe les contraintes spécifiques liées à l'utilisation de la plate-forme désirée. La transformation de modèles, illustrée figure I.8, permet le passage du modèle PIM vers le modèle PSM. Les modèles de l'approche MDA sont exprimés par des langages basés sur le MOF (Meta-Object Facility) (UML, CWM). Le MOF est un standard de l'OMG qui définit la spécification des métamodèles (structure de représentation des modèles). L'utilisation d'un langage MOF garantit la vérification, le parsing et la transformation des modèles avec les outils standardisés qui le supportent. Les transformations résultent de l'application de règles qui convertissent un modèle en un autre du même système. Ces règles peuvent s'appliquer pour transformer un métamodèle en un autre, pour utiliser des marqueurs d'un modèle qui guident les transformations, pour fusionner un modèle à un autre afin d'obtenir le modèle désiré.

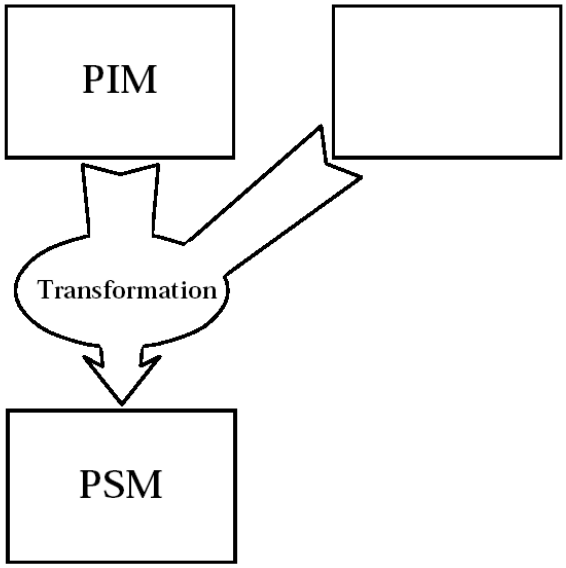


FIG. I.8 – Transformation de modèles

Cette approche convient donc au projet dont la conception est partagée entre divers équipes focalisées sur la réalisation d'une partie d'un même système. Elle offre l'interopérabilité qui permet de regrouper les réalisations des différentes étapes et de les fusionner via les règles de transformation déterminées. Elle peut tout à fait s'appliquer dans une méthodologie de conception PBD pour réaliser le processus de raffinement des plate-formes, exprimées à différents niveaux d'abstraction. Les différentes plates-formes correspondent alors aux modèles intermédiaires entre le PIM et le PSM, obtenus par des transformations successives. Son caractère standard, appuyé par l'utilisation d'un langage issu du MOF, tel qu'UML, conforte l'intérêt d'utiliser une telle méthode dans un processus de conception d'un SoC.

I.2 Le langage UML dans la conception de SoC

I.2.1 Qu'est-ce qu'UML ?

Le langage UML est un langage de modélisation standardisé et maintenu par l'OMG depuis 1997. Ce langage semi-formel, supporté de manière graphique, est défini par un métamodèle UML [48], lui même défini à partir d'un méta-métamodèle MOF (Meta Object Facility) [50]. Ce dernier décrit les informations utilisées par les métamodèles. Ces mêmes métamodèles décrivent la sémantique (définition et sens d'utilisation) des **éléments** employés dans les modèles et leurs relations. Le langage UML, centré sur la modélisation objet, est principalement utilisé pour représenter les applications informatiques de n'importe quel domaine (médecine, électronique, mécanique, banque, etc.). Il sert à spécifier la structure et le comportement des systèmes, grâce aux divers diagrammes graphiques qu'il propose. Ceux-ci permettent de recouvrir presque toutes les étapes de développement du cycle de conception des applications, de la description du système à la génération de code. Il propose dans la standardisation 1.4 (utilisée pendant le déroulement de cette thèse), un ensemble de 9 diagrammes, classés suivant des représentations statiques ou structurelles et dynamiques, qui offre la possibilité d'exprimer à différents niveaux de précisions, l'application à réaliser :

- Représentation statique ou structurelle
  - *le diagramme de classes* : est le diagramme principal d'UML. Il définit la structure (relation, attributs, classification) des entités manipulées par l'utilisateur. Ce diagramme est utilisé pour décrire l'architecture d'une application.
  - *le diagramme de package* : permet de hiérarchiser les modèles en regroupant différents diagrammes au sein d'un même package.
  - *le diagramme de cas d'utilisation* : est utilisé pour exprimer le besoin. Comme son nom l'indique, il exprime les différents cas d'utilisation possible de l'application à partir des acteurs identifiés. Il évoque notamment les fonctionnalités nécessaires aux utilisateurs du système.
  - *le diagramme de déploiement* : décrit l'architecture du système logiciel, comment sont installés les différents composants du système.
  - *le diagramme d'objets* : sont utilisés pour illustrer un contexte, à l'aide d'instances de classes définies dans un diagramme de classes.
- Représentation dynamique
  - *le diagramme d'activité* : exprime l'aspect dynamique d'un système par la description des points d'entrée, de sortie et l'enchaînement des processus.
  - *le diagramme d'états* : exprime les différents états d'un objet et les transitions entre les états. Les états peuvent être hiérarchiques et les transitions sont déclenchées sur événement.
  - *le diagramme de séquence* : exprime le caractère dynamique d'un système à travers la description des séquences d'interaction temporisées entre objets durant leur cycle de vie.
  - *le diagramme de collaboration* : de même que le diagramme de séquence, il décrit les interactions (échange de messages) entre objets mais sans précision temporelle, uniquement de manière globale.

Chacun de ces diagrammes manipule des éléments particuliers de la sémantique UML, ce sont des "*Class*", des instances de classe ("*Object*"), des "*Components*" etc.[48]. Les différents domaines d'activité qui utilisent UML, disposent des mêmes éléments UML (issus du méta-

modèle) pour modéliser leurs applications. Or les applications sont très variées, et les objets manipulés différents. UML offre donc un mécanisme de spécialisation des éléments liés aux classes d'applications, qui leur confère une sémantique adaptée à leur domaine. Cette personnification s'effectue par le biais de **profils UML** qui apportent des extensions au langage (*stereotype, tagged value, constraints*). Les **stéréotypes** permettent ainsi d'attribuer des caractéristiques particulières (paramètres) aux divers éléments du métamodèle par le biais de "*tagged value*". Cette notion de profil définit donc la sémantique particulière qui spécialise UML pour un domaine d'activité à travers un ensemble de diagrammes. Le profil formalise et apporte des règles d'utilisation pour les modèles. Plusieurs profils appliquables à la spécification UML 1.4 existent, que ce soit dans le domaine des systèmes électroniques temps réels [51] [52] [2] [53], de l'informatique [54] [55] ou encore de la biologie [56]). La prise en compte du temps réel avec UML, et notamment des profils associés, est traitée dans le paragraphe I.2.2.

Depuis 1997, le métamodèle UML a continuellement évolué afin de proposer une structure et des éléments offrant des possibilités de modélisation le plus en adéquation avec les besoins des concepteurs (programmation). Au début de cette thèse, l'UML 2.0 était encore dans le processus de standardisation à l'OMG. Un nombre limité d'outils UML prenaient tout juste en compte l'intégration des modifications apportées par le métamodèle 1.5 qui allait être remplacé. Dans cette configuration nous nous sommes donc orienté vers l'utilisation du métamodèle 1.4, intégré dans l'ensemble des outils UML du moment, le lecteur intéressé par les outils pourra se référer à la liste non exhaustive fournie par l'OMG sur son site [57]. Néanmoins les évolutions majeures de la version en cours de standardisation ont été pris en compte au sein du modèle développé, pour un passage plus aisé de notre profil (conforme au métamodèle 1.4) au métamodèle 2.0. Les évolutions de la version UML 2.0, concernent la modification du métamodèle pour intégrer de nouvelles notions et l'ajout de 4 diagrammes supplémentaires qui sont, outre l'officialisation du diagramme de package : le diagramme d'objets, le diagramme de structure composite, le diagramme de temps, et le diagramme global d'interaction [58]. Nous ne détaillerons pas ces diagrammes dans ce document, principalement axé sur la conception d'un profil conforme au métamodèle 1.4.

I.2.2 Adapter UML : Les Profils

La problématique de l'utilisation du langage UML dans la conception des systèmes temps réel embarqués est devenue une activité de recherche importante [59]. En effet, les caractéristiques de ce langage (présentées précédemment) font de lui un bon candidat à la modélisation et à la spécification à haut-niveau de ces systèmes. D'ailleurs, afin de faciliter son intégration au sein de méthodologies de conception, plusieurs profils ont été développés et standardisés pour intégrer les notions de temps réel des systèmes électroniques et permettre les vérifications et validations temporelles. C'est le cas des profils (1) ***"UML Profile for Schedulability, Performance and Time"***(SPT) [2] et (2) ***"UML Profile for Modeling QoS and Fault Tolerance Characteristics and Mechanisms"***(QoS) [53]. Le premier (1) apporte la sémantique des contraintes temps réel utiles aux analyses temporelles d'un système (analyse d'ordonnancement, analyse de performances) avec certaines caractéristiques de qualité de services introduites en tant qu'attribut. Il permet la description des propriétés temporelles et la prédiction du comportement du logiciel par détection des incohérences lors de la phase d'analyse, ceci avant même d'entamer la phase de développement. Il supporte également l'interopérabilité des outils de modélisation et d'analyse UML. Il est décomposé en 3 sous-profils présentés sur la figure I.9.

Le "*General Resource Modeling Framework*" sert de base aux autres sous-profils et déter-

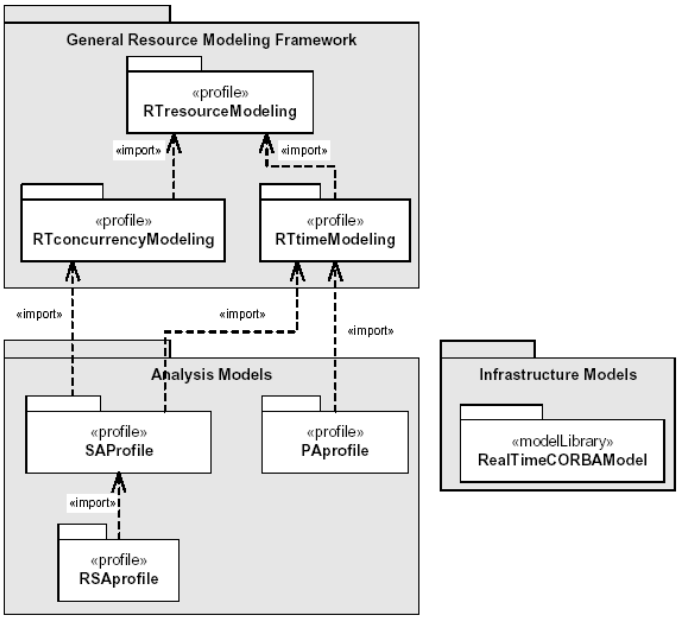


FIG. I.9 – Structure du profil SPT [2]

mine la sémantique des ressources proposées aux analystes systèmes et aux développeurs qui modélisent des systèmes et qui veulent vérifier leurs performances temporelles. Les annotations permettant la définition des ressources sont regroupées en deux catégories : "concurrentielles" pour la gestion du parallélisme, et "temporelles" lorsqu'il s'agit de modéliser le temps. Le second sous-profil, "*Analysis Model*", spécifie les différents types d'analyse pouvant être effectuées sur les ressources. Trois modèles d'analyses sont définis dans le sous-profil, un modèle d'analyse de performance (PAProfile), un modèle d'analyse d'ordonnançabilité (SAProfile) et un modèle d'analyse spécifique à l'ordonnancement des applications temps réel CORBA (RSAProfile). Le dernier sous-profil, "*Infrastructure Models*" est lié au monde CORBA.

Le second (2) profil décrit les caractéristiques permettant d'apporter des informations non fonctionnelles à un système en terme de qualité de service (QoS) (latence, sûreté, probabilité d'erreur, etc.), les restrictions qui leurs sont imposées (bornes), et ceci en fonction de plusieurs modes d'exécution. Il possède pour cela 3 "*packages*", "*QoSCharacteristics*", "*QoSConstraints*" et "*QoSDimensions*" . Les "*QoSCharacteristics*" représentent les caractéristiques de services quantifiables, non fonctionnels, tels que les délais, le débit, la disponibilité etc. Les "*QoSContraints*" imposent les restrictions aux "*QoSCharacteristics*" (temps de réponse maximum, nombre minimal d'erreur etc.). Les "*QoSLevels*" définissent les différents niveaux de qualité de chaque caractéristique et les passages possibles entre les différents niveaux. Cela permet de définir les modes de fonctionnement dégradés d'un système et les possibilités de passage d'un mode à l'autre. Ces profils ne reflètent toutefois qu'un seul aspect des systèmes, et ne sont pas suffisants pour exprimer l'intégralité des caractéristiques d'un système complet. De plus, les mécanismes d'annotation et d'utilisation des éléments ne sont pas simples et ils ne facilitent pas la construction et la compréhension des modèles.

C'est pourquoi certains profils réutilisent les profils existants, et étendent les concepts déjà développés pour offrir davantage de possibilités de développement, de flexibilité et faciliter la modélisation. C'est le cas du profil MARTE, ***"UML Profile for : Modeling and Analysis***

of Real-Time and Embedded Systems", intégré aux travaux de développement du programme commun de recherche "CARROLL", sur l'ingénierie logicielle dirigée par les modèles [51]. Thalès, le Commissariat à l'Energie Atomique (CEA) et l'Institut National de Recherche en l'Informatique et Automatique (INRIA), associés aux Universités de Carleton, ESEO, ENSIETA et aux industriels IBM, Alcatel et d'autres, répondent à l'appel à proposition (RFP février 2005) lancé par l'OMG [60] qui veut remplacer le SPT 1.1 (basé sur UML 1.4) au profit d'un profil plus complet et plus simple, conforme à la norme UML 2.0 et SysML. Ils proposent donc de ré-homogénéiser les profils existants (SPT, QoS), de combler certaines lacunes sémantiques pour étendre la modélisation au domaine de l'embarqué (unité de mesure de conversion, tuples, expressions temporelles etc.) et d'ajouter l'analyse des modèles des machines d'états. Le profil MARTE regroupe donc un ensemble de sous-profils comprenant des modèles de représentation temporels, de représentation des propriétés non fonctionnelles (formalisation UML 2.0 des profils SPT et QoS), des modèles d'analyses d'ordonnancement, des modèles de ressources d'exécution matérielles et logicielles. Ces sous-profils sont rassemblés dans la spécification illustrée sur la figure I.10.

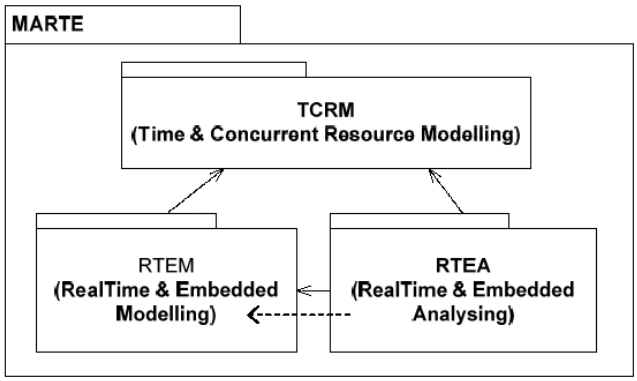


FIG. I.10 – Architecture haut-niveau de la spécification MARTE

Le *package* "*Time & Concurrent Resource Modelling*" deviendrait la nouvelle infrastructure standardisée temps réel embarquée de l'OMG. Elle contiendrait la sémantique utile à la modélisation des paramètres temps réel embarqués "*package RTEM*" (spécification des propriétés non fonctionnelles, des éléments du comportement statique, déterministe) et la sémantique nécessaire pour l'analyse de ces modèles,"*package RTEA*" (analyse des modèles de performances et d'ordonnancement). Ce profil n'existait pas lorsque le projet A3S a débuté, il a fait son apparition en cours de projet et son développement se poursuit toujours.

Selic et Rumbaugh vont jusqu'à transposer un langage de modélisation de conception d'architectures logicielles (Real-time Object-Oriented Modeling (ROOM)) [61] en UML pour en faire un profil dédié : ***"UML-RT"*** [52]. Le langage ROOM est un langage de définition d'architectures logicielles spécifiques, complexes, dirigées par les événements et potentiellement distribués. Le profil temps réel créé reprend tous les éléments du langage ROOM et fournit la sémantique UML permettant la modélisation structurale et comportementale d'un système temps réel complexe. Le système se décompose en composants hiérarchiques, les «*capsules*», qui interagissent entre eux par échanges de signaux entre leurs «*ports*» connectés via des «*connector*» qui indiquent le chemin des communications. Le comportement de chaque capsule est traduit par une machine d'état qui lui est associée et la séquence des signaux échangés est définit par

un «*protocol*» associé à chaque port, comme l'illustre la figure I.11. L'architecture fournie par le profil est cependant restreinte, comme les possibilités de modélisation de performances offertes. Cela reste donc un profil complémentaire au SPT.

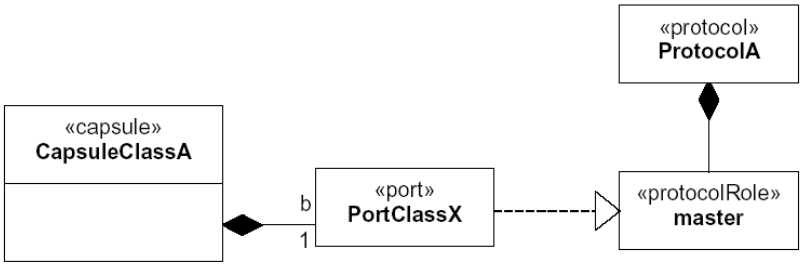


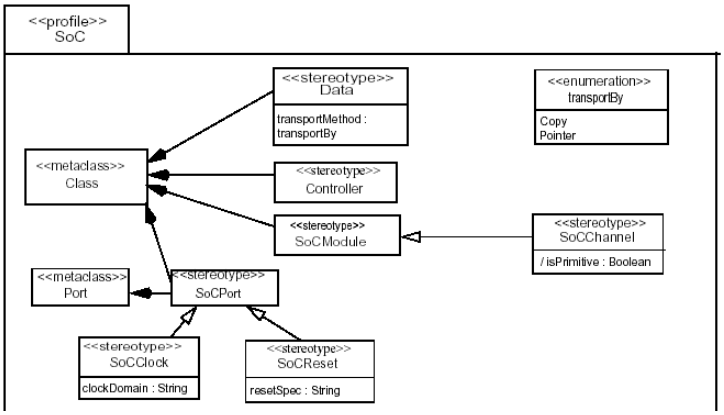
FIG. I.11 – Port et Protocol

STmicroelectronics exploite UML 2.0 pour générer leur spécification dans un autre langage, SystemC [46]. Il propose un profil UML SystemC (***"UML profile for SystemC"***) [62] qui spécifie (via les stéréotypes) les éléments UML correspondant à la sémantique utilisée dans le langage SystemC, afin d'avoir une dimension graphique qui modélise à un niveau système plutôt que de concevoir à bas niveau à travers le codage [39]. Il définit ainsi un ensemble de stéréotypes issus du langage SystemC («*sc_module*», «*sc_port*», «*sc_interface*», «*sc_channel*», sc_thread, etc.). Le profil spécifie 2 niveaux, correspondant pour le premier au cœur de SystemC (couche 0)(formalisme SystemC) et pour le second à des *ports*, *channels* et *interfaces* prédéterminées dans des librairies (couche 1)(signal, mutex, semaphore). Ces stéréotypes spécifiés sont utilisés à la fois dans des diagrammes de structure (Diagrammes de Classes et diagrammes de structures composite) pour modéliser les blocs et leurs assemblages (connections des «*sc_module*» via «*sc_channel*» et «*sc_prim_channel*») et dans les diagrammes comportementaux pour la modélisation fonctionnelle exprimée par les *process* et *channel* de la spécification SystemC. L'objectif de ce profil est de fournir aux ingénieurs matériels et logiciels un moyen d'améliorer le processus de conception d'un SoC à haut-niveau permettant la spécification, l'analyse, la construction, la visualisation et la documentation des parties matérielles et logicielles d'un SoC [63].

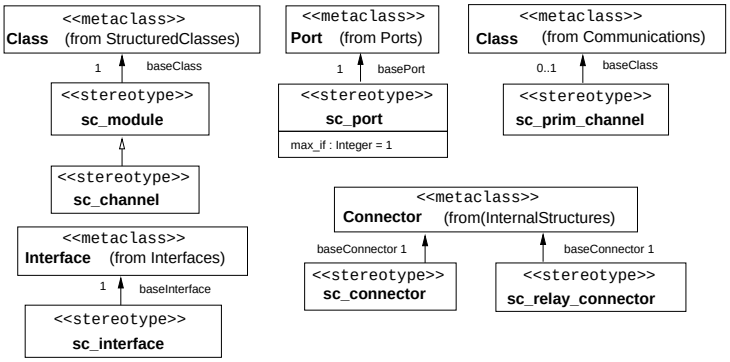
UML 2.0 se rapproche également d'un autre langage de modélisation, qui est lui, spécifique aux applications d'ingénierie logicielle, par le biais de l'adoption, toute récente, du profil ***"UML Profile for Systems Engeneering"*** (SysML) non encore finalisé [64]. L'ingénierie logicielle utilise en effet des approches similaires à la conception des systèmes temps réel embarqués et utilise de plus en plus UML. Une explication donnée par l'association française d'ingénierie système est que la conception de l'ingénierie système est fortement guidée par les besoins du passage à la programmation par objet, or les premières versions de SysML étaient peu adaptées à la modélisation de systèmes complexes [65]. D'où la création de ce profil qui nécessite quelques extensions de stéréotypes et extensions de diagrammes (diagramme de concepts, diagramme d'exigences, diagramme paramétrique) qui enrichissent les possibilités de spécifications, d'analyses, de conceptions, de vérifications et de validations d'une grande gamme de systèmes complexes.

Les groupes Fujitsu et IBM supportés par d'autres compagnies internationales (Canon, Toshiba, etc.) se sont eux lancés dans la spécification d'un profil pour SoC, pendant le projet A3S.

Ce ***"UML Profile for System on Chip"*** [66] vient tout juste d'être standardisé à l'OMG. Il s'agit d'apporter la sémantique permettant de décrire, un SoC par le langage UML 2.0 (modélisation structurelle, modélisation des communications, des opérations et des propriétés). Ce profil fournit pour cela la possibilité de représenter des modules et des canaux hiérarchiques qui caractérisent les SoC (ensemble de blocs interfacés par des ressources de communication). C'est un profil qui est très proche du profil UML SystemC de STmicroelectronics quant aux stéréotypes qui y sont spécifiés («*SoCModule*», «*SoCPort*», «*SoCInterface*», «*SoCChannel*», «*SoCProcess*») comme illustré sur la figure I.12. Il permet d'ailleurs la génération automatique de code SystemC. Sa proximité avec le code SystemC et sa génération lui permet de couvrir différents niveaux d'abstraction allant du niveau transactionnel (TLM) jusqu'au niveau registre (RTL).



(a) Extrait du profil UML for SoC



(b) Extrait du profil UML SystemC [62]

FIG. I.12 – Extraits des profils UML for SoC et SystemC

L'université de Tampere (Finlande) a développé son propre profil UML 2.0, ***"TUT-Profile"*** [3], en utilisant le mécanisme d'extension par stéréotype (et "*tagged value*") offert par le langage UML. TUT-Profile offre les éléments pour définir la structure et les paramètres de l'application et de la plate-forme ainsi que le mapping basé sur des blocs IP. L'application peut être modélisée par un ensemble de classes actives dont le comportement est spécifié à travers les diagrammes UML, tandis que la plate-forme est représentée de manière plus abstraite à travers des composants de librairies. Une application est composée de composants applications «*ApplicationComponent*» (composant fonctionnel) qui sont instanciés en tant que «*ApplicationProcess*», comme illustré sur la figure I.13. Les plates-formes sont constituées suivant la même démarche.

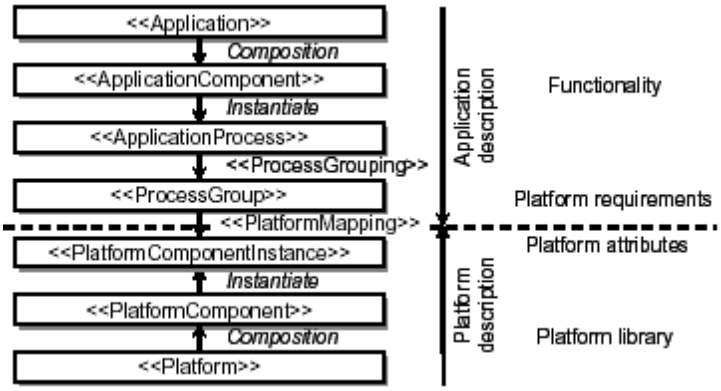


FIG. I.13 – Hiérarchie du profil TUT-Profile [3]

Les composants de l'application tout comme ceux de la plate-forme peuvent être caractérisés par l'intermédiaire de valeurs fournies aux "*tagged value*" allouées à leurs stéréotypes respectifs. Ces "*tagged value*" correspondent à des contraintes non-fonctionnelles utilisées pour les spécifications, l'automatisation de la conception (*mapping*), l'analyse et l'implantation. Pour exemple, un «*ApplicationProcess*» possède les "*tagged value*" : *priority, code memory, datamemory, realtimetype, processtype*. Ces caractéristiques peuvent servir à l'analyse d'ordonnancement et à des calculs de partitionnement. D'ailleurs d'autres stéréotypes sont définis pour l'analyse de l'exploration architecturale, à savoir des «*processinggroup*» qui sont utilisés pour regrouper plusieurs «*ApplicationProcess*» à implanter sur le même composant matériel. Les composants de la plate-forme possèdent également des propriétés exprimées à travers les "*tagged value*" et s'interfacent entre eux par des éléments de communication. L'aspect système d'exploitation n'est pas explicitement évoqué mais il se retrouve à travers un stéréotype de type d'ordonnancement («*scheduling*») et dans certains paramètres de "*tagged value*" définis (priority). Cet aspect laisse peu de liberté quant aux choix d'OS et sa prise en compte dans une analyse de performances temporelles. L'originalité de ce profil est de limiter l'utilisation des diagrammes pour la modélisation, et la prise en compte dans le modèle, d'annotations issues des résultats de performances.

Parallèlement aux aspects strictement temporels, certains domaines applicatifs cherchent à adopter une solution de modélisation commune de leurs systèmes. C'est le cas du domaine particulier de la Radio Logicielle, où une spécification est en cours de standardisation (non finalisée), et propose tout un ensemble d'éléments permettant de répondre à une partie des problèmes de conception temps réel de ces systèmes (modélisation PIM + mécanismes de transformation PSM pour cible CORBA). Cette spécification ***"PIM and PSM for Software Radio Component"*** [19] comprend notamment le *UML Profile for SWRadio* (profil SDR) qui permet la modélisation d'un système Radio Logicielle PIM. Il étend le langage UML 1.4 avec une sémantique spécifique au domaine Radio Logicielle non seulement pour modéliser, mais aussi pour permettre la génération automatique de fichier de description (fichier XML) et de code (squelette) pour valider le système durant la conception et le développement d'un environnement de simulation Radio Logicielle. Ce profil prend en compte les 3 compétences nécessaires à la conception des systèmes Radio Logicielle au travers des trois *packages* du profil :

(1) les développeurs d'applications et de composants logiciels, *"Applications and Devices package"*,

(2) les fournisseurs d'infrastructures de communication, *"Infrastructure package"*,

(3) les fournisseurs de plates-formes matérielles Radio Logicielle, *"Communication Equipment package"*.

Le premier *package* définit des stéréotypes qui étendent les "*component*" et "*interface*" (éléments du métamodèle UML) pour inclure les concepts nécessaires au développement des applications Radio Logicielle. Il fournit un ensemble d'interfaces et de composants utilisés pour le développement d'applications forme d'onde. Il englobe les stéréotypes «*Resource*» (utilisés pour modéliser un composant de l'application PIM), «*Device*» (utilisé pour modéliser un composant de l'application PSM), «*DeviceDriver*», et l'API logicielle Radio Logicielle «*SWRAPI*», afin de pouvoir interfacer les composants sur étagères (Commercial Off The Shell). Le *package* d'infrastructure fournit quant à lui la sémantique des composants de service et de communication du domaine radio (intergiciel CORBA). Ce sont des stéréotypes "*component*" et "*interface*" là aussi, qui sont définit : «*Device Manager*», «*Radio Manager*» etc.. Le *package* (3) formalise les différents composants matériels qui composent l'équipement matériel des systèmes Radio Logicielle. Il inclut les stéréotypes «*RF Device*», «*I/O Device*», «*Security Device*», «*Antenna*», «*Frequency Converter*» etc., disposant chacun de paramètres utiles pour le comportement de la forme d'onde déployée. Ce profil reste restrictif au domaine de la Radio Logicielle et propose des éléments de représentation de la plate-forme matérielle encore trop généralistes pour bien spécifier ce type d'application. De plus, le développement des systèmes d'exploitation à partir de ce profil sont limités à l'intergiciel CORBA (Common Object Request Broker Architecture) qui gère l'intégration des applications distribuées sur les architectures hétérogènes.

Les méthodologies de conception évoluant vers la conception PBD, les auteurs de [67] ont développé un ***"profil PBD"***, décrivant la sémantique utile à la modélisation de la structure et du comportement de systèmes embarqués à différents niveaux d'abstraction (à commencer par le niveau conceptuel). Cette sémantique reprend les éléments du langage UML 1.4 et du profil SPT auxquels ils ajoutent des classificateurs spécifiques au domaine de l'embarqué ainsi que des relations spécialisées entre stéréotypes. Ils proposent des stéréotypes qui spécialisent les "*class*", "*component*", "*nodeinstance*", et les relations d'"*association*" pour apporter la sémantique des systèmes qui identifie les éléments fonctionnels, les modèles de calcul, les éléments de l'architecture, et qui leur attribue certaines caractéristiques en conséquence. Parmi ces stéréotypes nous retrouvons le sétérotype de «*Resource*», attribué aux éléments de la plate-forme matérielle. Il se spécialise en «*Bus*», «*Memory*», «*Processor*», dont les attributs sont annotés par des "*tags*". Chacun peut disposer de «*port*» comme interface d'interaction avec les autres «*Process*». Le stéréotype «*Process*» spécifie les objets de calcul (fonctions). Les fonctions de communications à implanter et à raffiner sont stéréotypées «*medium*». Les communications inter-process, représentées par des relations d'associations, sont elles stéréotypées «*Communicate*». Elles sont spécialisées pour définir le modèle de calcul associé à la communication («*SDF Communicate*», «*Kahn Process Networks Communicate*»...). D'autres stéréotypes sont définis pour spécifier le type d'algorithme à appliquer pour l'accès aux ressources partagées. La notion de services et d'utilisateur de services est aussi présente pour des extensions futures. Un exemple d'utilisation est donné sur la figure I.14. La modélisation structurelle des différents éléments d'un système peut donc être réalisée via les éléments stéréotypés du profil PBD (les "*class*" représentent les fonctions, les "*component*" représentent l'implantation logicielle, les "*nodeinstance*" pour la ressource physique qui exécute...). Les stéréotypes définis dans le profil résument la fonctionnalité d'un système à deux actions au lieu de trois. La première consiste à regrouper la

décomposition fonctionnelle avec la définition du modèle d'exécution. Ils utilisent pour cela les stéréotypes de classe «*Process*» et d'interconnexion «*Communicate*» dotés d'attributs relatifs (nom, type de port etc.). La seconde concerne la spécification du comportement par raffinement de ces classes stéréotypées, par un diagramme d'activité ou de machine à états, suivant le modèle d'exécution. Le profil ainsi obtenu peut être utilisé dans une méthodologie (Metropolis) à plusieurs étapes du flot de conception et fournir les informations utiles aux outils d'analyses et de synthèses appropriés. C'est un profil qui met en évidence les composants matériels.

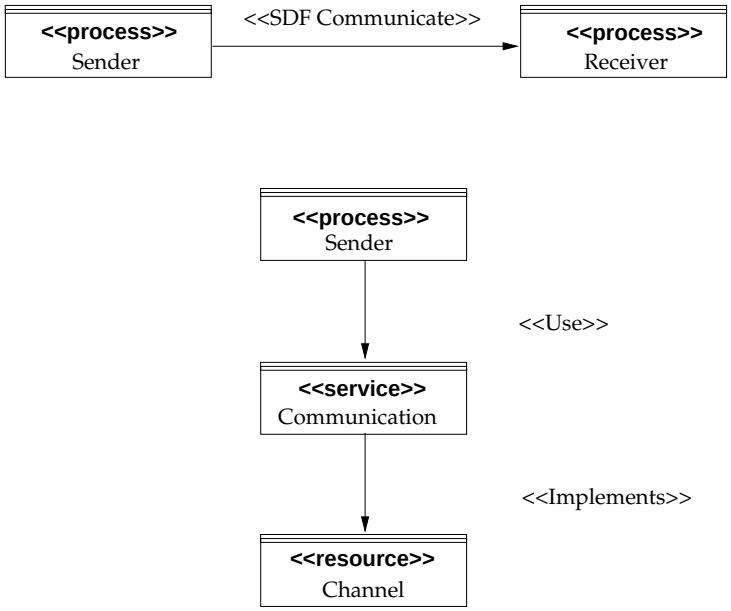


FIG. I.14 – Relations entre stéréotypes

L'effervescence autour du langage UML et la naissance des profils qui l'adapte au domaine spécifique des systèmes temps réel embarqués, montrent le besoin d'uniformiser dans un standard existant, le formalisme utilisé dans la conception de ces systèmes. Les profils seuls ne sont cependant pas suffisants pour estimer les performances d'un système et valider sa conformité à un cahier des charges fixé. Ils s'inscrivent d'ailleurs dans des méthodologies de conception au niveau système, où ils tiennent une place plus où moins importante.

I.2.3 Intégrer UML dans les méthodologies de conception SoC

La pression liée à la productivité, les enjeux financiers et le besoin de s'assurer de la validité des systèmes électroniques temps réel pendant leur cycle de conception, amènent les chercheurs à réfléchir et à développer de nouvelles méthodes et langages de conception. UML est un de ces langages qui motive de nombreux chercheurs à définir son formalisme, la façon de l'utiliser et de l'intégrer au sein de méthodologies de conception nouvelles ou existantes. Nous allons donc nous intéresser ici à présenter un état de l'art de ces méthodologies, utilisant ou non, les profils précédemment présentés.

Dans le cas de la conception des SoC, la représentation se fait à haut-niveau d'abstraction. Une des possibilités de représenter cette abstraction, autre que mathématiquement, est l'utilisation de langages graphiques tels que le langage UML. Cette élévation du niveau d'abstraction,

amène un certain nombre de chercheurs à utiliser ce langage comme une abstraction supplémentaire venant au-dessus de leur méthodologie originale. Les chercheurs de l'Université Nationale de Singapour utilisent d'ailleurs UML pour transposer l'infrastructure de communication et les caractéristiques temporelles de SystemC en UML. Ils peuvent ainsi générer du code SystemC, simulable et synthésisable, à partir de diagrammes de classes et de diagrammes d'états. Ils utilisent des stéréotypes pour identifier les classes en tant que bloc SystemC (module, channel, interface). Chaque diagramme de classes modélise la structure d'un composant (bloc) et les diagrammes d'états sont utilisés pour modéliser son comportement. Ils ont développé deux flots pour produire du SystemC, l'un à partir des outils UML de Rational (RoseRT) [68], qui produit un SystemC au niveau RTL, et l'autre à partir de l'outil Rhapsody [69] qui produit un SystemC niveau TLM et comportemental. Les deux réalisations sont très similaires à savoir une modélisation en UML (RoseRT, Rhapsody) via les diagrammes, suivie d'une conversion en xmi, qui est parsée (RT2SystemC, via un template pour Velocity [70]) pour générer du SystemC (Synopsys System Studio). La figure I.15 illustre le flot suivi à partir de l'outil Rhapsody. Le SystemC généré est toutefois fortement contraint à l'outil de synthèse utilisé [71] et la modélisation des systèmes devient très vite compliquée et volumineuse dès lors que les systèmes comprennent plusieurs composants.

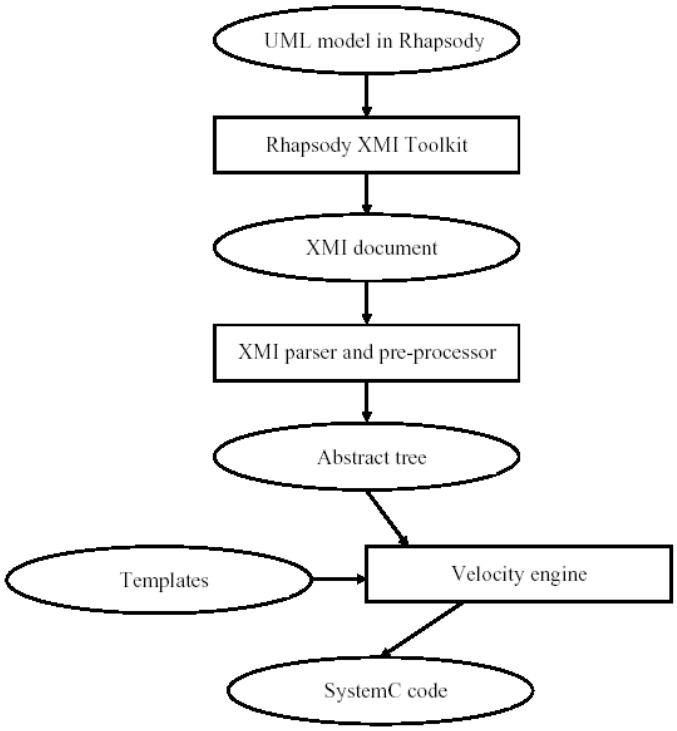


FIG. I.15 – Flot de l'implantation avec l'outil Rhapsody

Les laboratoires de Fujitsu utilisent UML 1.4 au sein d'un processus de conception des SoC à haut-niveau orienté objet (SLOOP : System Level Object-Oriented Process) [72]. Ce processus leur permet d'évaluer les performances et la fonctionnalité au niveau système à partir de modélisation UML. Ils utilisent pour cela des stéréotypes UML adaptés des méthodologies ROOM et du logiciel RoseRT. Leur méthodologie de conception repose sur 4 modélisations de niveau d'abstractions incrémentaux, illustrées sur la figure I.16 permettant l'implantation

des parties matérielles et logicielles d'un SoC, à l'aide d'exécutable C++ et SystemC. Chaque modélisation est en effet suivie d'une implantation de son modèle, en C++ pour le modèle conceptuel, et en SystemC pour les 3 autres. Le modèle conceptuel (diagramme de cas d'utilisation) est le point départ de la méthodologie, il capture les contraintes fonctionnelles et non-fonctionnelles aux travers des diagrammes de cas d'utilisation et de séquence (représentent les scénarios des diagrammes de cas). Il est suivi du modèle fonctionnel, qui présente la structure de l'application (diagramme de structure composite) indépendamment de l'architecture physique (ensemble de processus communiquant), et du modèle architectural (diagramme de déploiement), composé de ressources physiques (processeur, DSP, FPGA) et de communications (bus, mémoire) présentes en bibliothèque et paramétrables (ordonnancement multi-tâches, largeur de bus, latence de transfert). Le modèle d'exécution considéré est un réseau de processus de Kahn (dont le modèle est issu du diagramme de structure). Il permet de calculer les charges de travail des communications (nombre de jetons) et des calculs (nombre d'invocations des fonctions) de l'application. Les résultats de ces calculs sont pris en compte et associés aux valeurs des paramètres des composants matériels pour l'analyse des performances après mapping dans le modèle de performance. Ce modèle, en effet, *map* chaque composant du diagramme de structure composite sur un composant du diagramme de déploiement et permet ainsi l'évaluation des performances. Dans cette méthodologie l'absence de vérifications des modèles fait cruellement défaut pour valider les résultats obtenus.

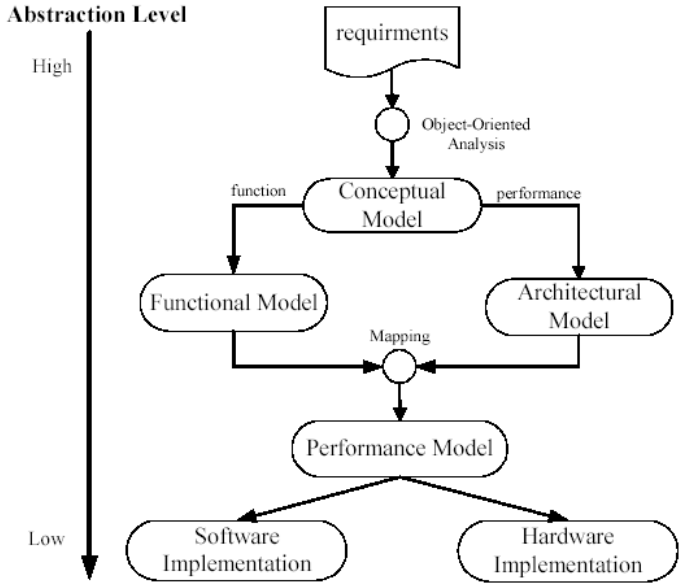


FIG. I.16 – Flot de l'outil SLOOP

NEC et l'Université de Lugano, élèvent également le niveau d'abstraction des modélisations des systèmes embarqués au niveau système par le biais d'UML 2.0. Leurs équipes ont intégré, dans un premier temps, UML (via l'outil Rhapsody UML) à leur environnement de co-design matériel/logiciel (ACES), pour la modélisation et la spécification de la partie logicielle uniquement [73][74]. Elles mettent notamment en avant l'indépendance du langage UML vis à vis de l'implantation sur une plate-forme et utilisent les diagrammes d'objets, diagrammes d'états et de séquence, pour extraire un modèle à événements discrets de l'application, point d'entrée de leurs

outils d'analyses et de synthèse, comme illustré sur la figure I.17. Le diagramme d'objets représente les liens de communication entre les éléments fonctionnels de l'application (événements). Un stéréotype de partitionnement a été introduit pour préciser quels sont les blocs fonctionnels dont l'implantation peut être matérielle ou logicielle. Les autres blocs non stéréotypés sont obligatoirement implantés en logiciel. Un diagramme d'états est associé à chaque objet fonctionnel pour décrire son comportement. Le diagramme de séquence fixe les contraintes temporelles des objets par des annotations. Le fichier exporté est ainsi utilisé par une interface web qui analyse et qui permet au concepteur d'effectuer un partitionnement manuel des blocs sur une plate-forme matérielle prédéterminée dont l'architecture est de type monoprocesseur avec un bus connectant une mémoire, un processeur et un accélérateur matériel. L'outil d'analyse génère ensuite le programme exécutable, les interfaces matérielles/logicielles adéquates basées sur les informations de partitionnement et de communication spécifiées par le concepteur. Le formalisme SystemC issue du langage UML permet la co-simulation du comportement du système et permet l'estimation des performances. NEC a ensuite fait évoluer le flot en y ajoutant la spécification de l'architecture matérielle à l'aide d'un diagramme de déploiement [75]. Ce diagramme est réalisé à partir d'une plate-forme de base identique à celle décrite précédemment, disponible en bibliothèque. L'évolution réside dans la possibilité de faire évoluer la plate-forme matérielle ciblée, soit par connexion d'autres plates-formes du même type en définissant les composants d'interconnexion, soit en ajoutant des composants individuels. Ces composants individuels sont de même nature que ceux qui composent la plate-forme, et sont contraints à être connectés à un bus. Cette méthodologie permet de parcourir de nombreuses étapes du flot de conception, allant de la spécification haut-niveau jusqu'à la co-simulation. Cependant, elle est restrictive à un type d'architecture donné et limité. Même s'il souhaite l'indépendance du modèle UML à l'implantation, la qualification des objets du diagramme d'objets avec le stéréotype «*partitionning*», entraîne déjà un choix d'implantation, notamment pour les objets non stéréotypés.

Le langage UML est prépondérant dans les méthodologies de conception PBD. Plusieurs outils mettent en évidence son utilisation de différentes manières (avec ou sans profil). Les travaux d'Oliveira [76], par exemple, proposent d'explorer l'espace de conception architectural à partir de résultats d'estimation fournis dès l'étape de modélisation effectuée avec le langage UML 2.0 et le profil SPT. Le flot proposé adopte l'approche MDA. Le système applicatif est tout d'abord décrit par un ensemble de cas d'utilisation donnant lieu à divers scénarios qui peuvent être annotés afin de préciser les plus performants. La structure de l'application et son comportement sont respectivement spécifiés à l'aide de diagramme de classes, de diagramme d'objets et de diagrammes de séquence. Ces derniers renseignent le nombre d'exécution de boucles et leurs conditions d'exécution. Toutes ces informations permettent de générer la signature de l'application, qui comprend à la fois les informations de structure et une pseudo trace. Vient ensuite le mapping de l'application sur un modèle d'architecture homogène précaractérisée (taille mémoire, consommation d'énergie, fréquence d'utilisation) effectué à partir d'un diagramme de déploiement. Des estimations de performances peuvent alors être obtenues pour chacun des cas d'utilisation et pour plusieurs architectures différentes afin d'être comparées et ainsi parcourir un espace d'exploration pour dégager la solution la plus pertinente. Leur approche est intéressante pour savoir quelle est la solution la plus avantageuse, cependant la précision des résultats d'estimations est variable concernant les temps d'exécution (exprimés en nombre de cycles) ainsi que l'énergie (exprimée sous la forme de bascules qui commutent). L'intégration d'autres modèles au niveau des plates-formes est à venir (intégrant : RTOS, API, driver, composant d'application).

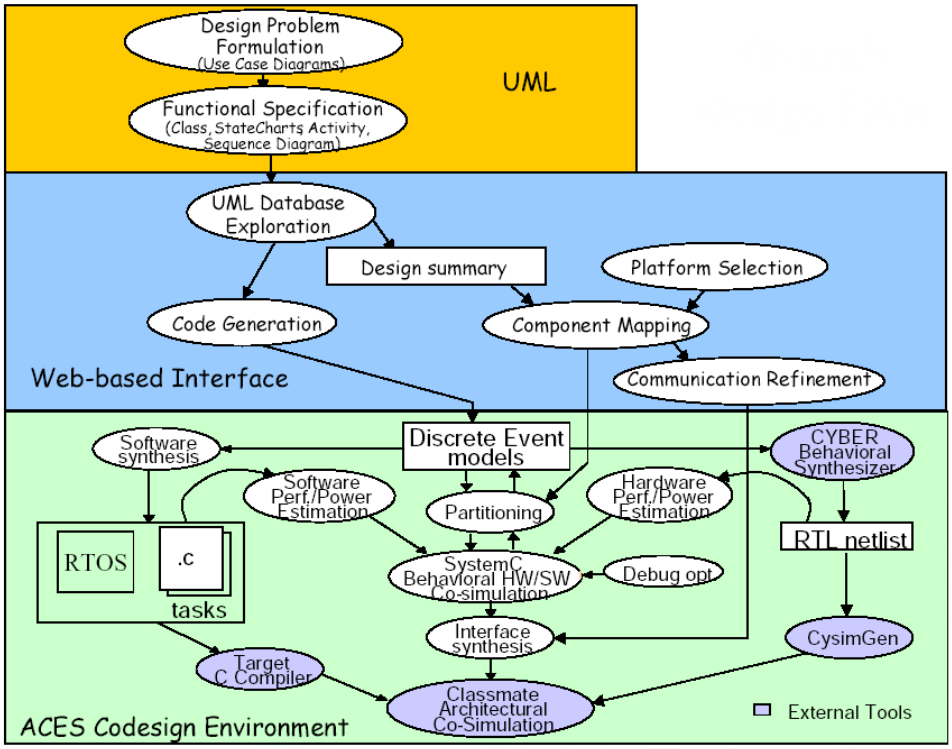


FIG. I.17 – Flot de l'environnement ACES

L'Université des Sciences Appliquées de Mittweida (Allemagne) a elle aussi adopté une méthodologie PBD, avec une approche MDA pour la conception des applications sur architectures reconfigurables. Leur conception est centrée autour du modèle de compilateur UML pour les architectures reconfigurables (MOCCA). Cette approche permet de partitionner, d'estimer et d'implanter le système à partir de modules matériels et logiciels. Beierlein et son équipe automatisent, par cette méthode entièrement orientée objet (logiciel et matériel), la synthèse matérielle à partir de modèles UML à haut-niveau d'abstraction et du langage d'action de MOCCA [77]. Le flot de conception donné sur la figure I.18 qui semble simple, masque une complexité répartie dans les différents niveaux d'abstraction à détailler pour obtenir les fichiers exécutables et les fichiers de configuration matériel. C'est une méthodologie pour générer, analyser et compiler du code. Le modèle PIM est obtenu après avoir fourni l'ensemble des types de données spécifiques à l'application (int, char, float) à travers un diagramme de classes stéréotypé. Il en est de même pour le modèle de plate-forme ciblée où les types de données attendues sont spécifiés en fonction de la cible. La structure de l'application est spécifiée à l'aide de classes et d'interfaces dont les relations se distinguent par des liens de dépendance, de généralisation et d'implantation dans les diagrammes de classes. Les classes sont annotées par le langage d'action. Son comportement est obtenu par des opérations et diagramme de machine d'états. La plate-forme matérielle est composée d'une architecture simple (processeur et FGPA avec chemin de communication) modélisée par un diagramme de déploiement. Le mapping est partiellement automatisé ou tout manuel. Dans un premier temps, le concepteur prend la décision de réaliser la fonction en logiciel ou en matériel (.exe, .bit) puis dans un second temps, il choisit sa cible sur le diagramme de déploiement. Le compilateur MOCCA analyse le système et génère les fichiers d'implantation (logiciel et matériel). C'est donc une méthodologie qui ne concerne que des plates-formes particulières centrées sur les composants reconfigurables. Elle demande

beaucoup de spécifications utiles au compilateur.

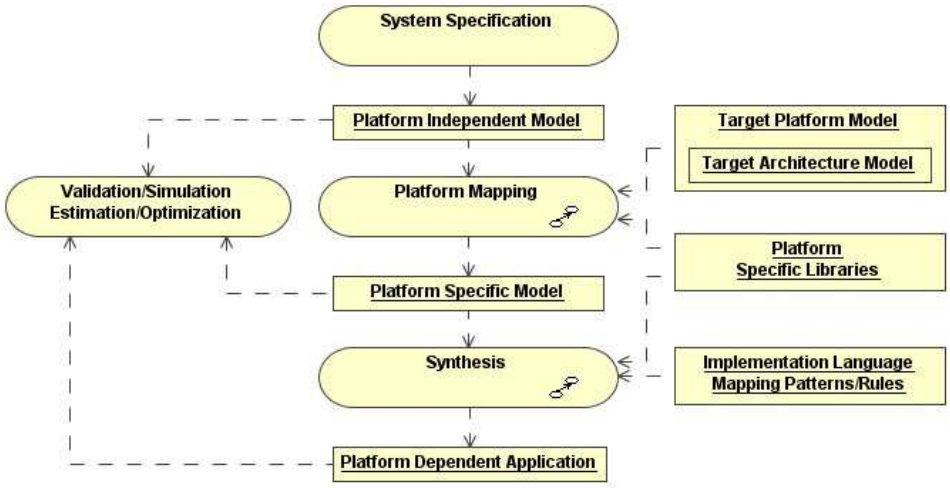


FIG. I.18 – Flot de codesign basé sur UML

La méthodologie de conception basée sur Metropolis [78][79] et le profil PBD [67] est une référence souvent citée dans le domaine. Metropolis propose un environnement de conception pour des systèmes hétérogènes qui supporte la simulation et l'analyse formelle. Il fournit une infrastructure basée sur un modèle dont la sémantique est assez générale pour supporter les différents modèles de calcul existants (librairie de MoC) et prendre en compte les nouveaux. Ses points d'entrées sont des spécifications textuelles ou graphiques des contraintes de conceptions non fonctionnelles, de l'application fonctionnelle et de l'architecture. Il a la particularité (qui l'est de moins en moins) de séparer à différents niveaux de granularité : la fonctionalité de l'architecture, le calcul de la communication et le comportement des performances, ceci, afin de mettre en avant la ré-utilisabilité. Metropolis dans la méthodologie exposée ici, et dont le flot est présenté sur la figure I.19, utilise ses outils d'analyses et de synthèse. Le profil PBD est utilisé pour toutes les spécifications UML et il est également intégré à Metropolis pour qu'il puisse comprendre les spécifications et procéder ainsi à l'analyse et à la synthèse. Un diagramme de cas d'utilisation est employé pour formaliser le problème (Design Problem Formulation) et ainsi représenter les services du système et ses exigences. L'utilisation des stéréotypes offerts par le profil conduit à définir dans un diagramme de classes, la structure du système applicatif (attribution des stéréotypes «*process*» et «*Communicate*»), puis de la raffiner afin de faire apparaître le modèle de calcul sous-jacent ainsi que les interfaces de communication des fonctions (le nombre et la directivité de leurs ports). Le comportement, lui est spécifié à différents niveaux d'abstraction grâce aux différents diagrammes qu'offre le langage UML 1.4. Les diagrammes de cas d'utilisation décrivent les services (niveau CIM), les diagrammes d'intéractions traduisent les intéractions entre composants système, et les diagrammes d'activité et de machines à états spécifient le comportement individuel des composants (ce peut être aussi des annotations textuelles comprenant des fractions de code). L'étape de spécification de la plate-forme capture les éléments matériels («*Process*») et leurs relations («*Communicate*»), ainsi que les mécanismes d'interaction entre les objets représentés. Elle comprend à la fois la spécification de l'architecture matérielle et le choix de réalisation du mapping. Elle précise les "services" offerts par les composants matériels (ex : un bus peut posséder un protocole avec une certaine qualité de service) tout en les associant aux besoins des fonctions. L'ensemble de la spécification réalisée gra-

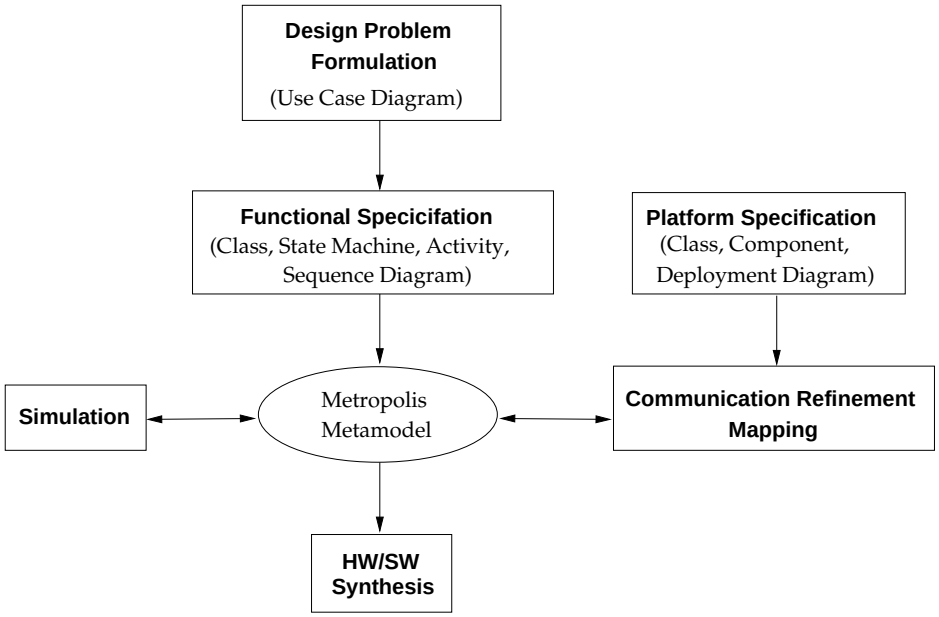


FIG. I.19 – Flot de conception d'une plate-forme UML

phiquement est également traduite dans le métamodèle Metropolis pour en permettre l'analyse. L'étape de raffinement qui précède l'analyse raffine les communications à partir d'un modèle spécifié à l'aide de ressources logiques (protocole, canaux virtuel...) contenues en librairie (Network Platform) afin de faire comprendre à Metropolis l'implémentation des intéractions entre objets. Metropolis considère un système réprésenté par une *netlist*. Les composants de la netlist peuvent être des *processes, media, ports, interface*, d'où la ressemblance avec les stéréotypes utilisés. Un système d'exploitation n'est pas représenté en tant que tel. Il peut être représenté par un réseau entre deux «*process*» implantés sur le même CPU. Le cœur du réseau est alors un «*media*» qui implante le RTOS. La spécification et la modélisation du système avec cette méthodologie est très fastidieuse. Il n'y a aucune vérification qui garantit que les modélisations effectuées soient correctes. Un oubli ou une erreur étant très vite réalisée, c'est l'ensemble du processus qu'il faut reprendre si cela se produit.

L'université de l'institut des sciences et techniques de Manchester (UMIST) s'est également intéressée à l'utilisation d'UML 1.4 dans sa méthode orientée objet pour la conception et le développement des systèmes embarqués [4]. En effet leurs travaux proposent une approche itérative et incrémentale PBD avec la modélisation d'objets matériels et logiciels sur puce (HASoC) basée sur la méthode de codesign orientée objet MOOSE [80]. L'originalité de cette méthode vient du fait que l'on dissocie le modèle de la plate-forme en deux parties : la partie proprement matérielle et la partie interface logicielle/matérielle (RTOS, drivers, machine virtuelle). Edwards et Green améliorent la méthode MOOSE et apportent une notation graphique ouverte (UML) associée au profil UML-RT (présenté à la section I.2.2). La première étape du flot, présentée sur la figure I.20, est semblable à celle des autres méthodes précédemment évoquées. Il s'agit de la spécification des exigences du système dans un langage naturel (description des contraintes, fonctions requises). Vient ensuite la modélisation "uncommited" qui va permettre d'obtenir un modèle exécutable . Elle est réalisée à travers des diagrammes de cas d'utilisation associés à des diagrammes de séquence et des machines d'états. Le diagramme de cas d'utilisation identifie les classes et objets à créer. Le diagramme de séquence décrit le comportement des objets (leur intéractions). A partir de ce diagramme, les objets actifs et passifs sont identifiés

et stéréotypés en capsules «*capsules*» qui englobent les objets passifs (hiérarchie). Le modèle de classe issu des classes identifiées à travers le diagramme de cas d'utilisation, représente la structure statique de l'application, alors que le modèle d'objets représente son comportement pendant l'exécution. Le comportement plus précis de chaque capsule est spécifié à travers un diagramme de machines à états. La modélisation "commited" revient à préciser l'interface de communication entre les objets, et à effectuer les choix d'implantation. Dans le cas ou certaines interfaces n'aient pas été déterminées dans les spécifications, c'est à ce niveau qu'elles sont introduites, dans le cas contraire cette étape se limite à effectuer le partitionnement en affectant un stéréotype spécifique (ex : «*sw capsule Mips*»). La plate-forme matérielle est représentée à la fois par un modèle d'architecture matérielle (HAM) via un diagramme de déploiement mais aussi par son modèle d'interface logicielle associée (SHIM), à savoir différentes couches d'abstraction correspondant aux *driver*, au modèle du système d'exploitation temps réel et au modèle de la machine virtuelle. La machine virtuelle supporte l'exécution du "commited model". Ce modèle d'interface logicielle est défini par un diagramme de composants spécifiant les mécanismes utilisés pour les communications entre éléments du modèle "commited" implantés en logiciel et ceux implantés en matériel. L'architecture matérielle est modélisée à partir d'un diagramme de déploiement dont les divers composants (processeur, mémoire, interconnexion, interface, etc.) possèdent une description matérielle synthétisable. La dernière étape, celle de l'intégration effectue le *mapping* entre le modèle de plate-forme et le modèle "commited" pour vérifier la satisfaction ou non du respect des contraintes imposées (la validation du comportement étant effectuer sur le modèle "uncommited").

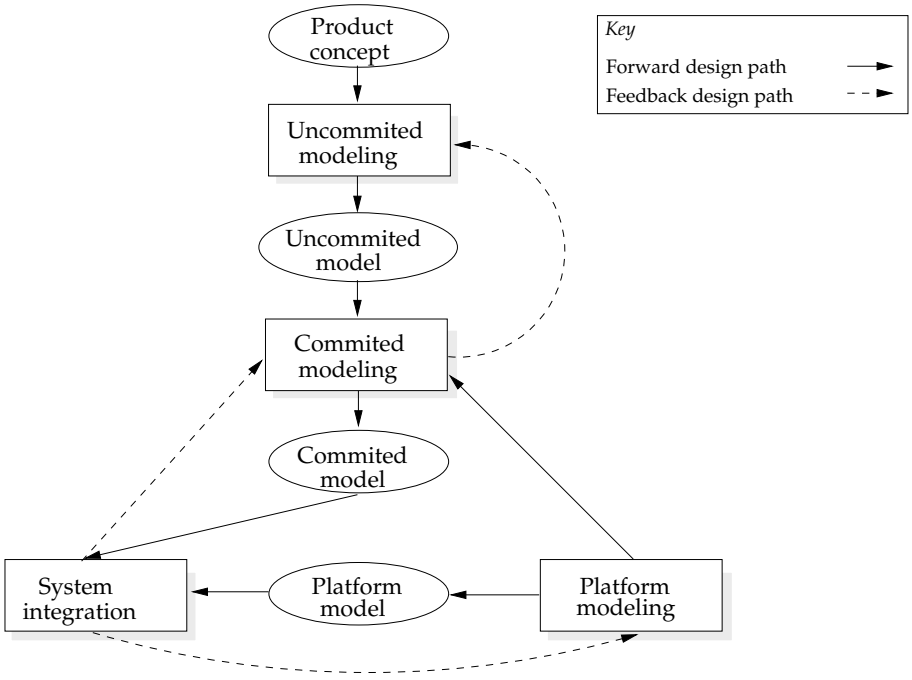


FIG. I.20 – Cycle de HASoC [4]

L'université de Tampere (Finlande) est la première à ma connaissance à proposer un flot de conception complet d'un SoC, qui à partir de modélisations et spécifications de haut-niveau (en UML 2.0), implante physiquement un système (prototypage FPGA) et renvoie des annotations sur ses performances. A l'origine du profil TUT ([3]), les chercheurs de cette université ont

développé une méthodologie non PBD, à base de classes actives qui utilisent ce profil, et qui a été intégré à l'outil Koski. Ils ont cependant l'approche MDA dans leur processus de conception. Le flot, tiré de [3], est présenté sur la figure I.21 et décompose la conception avec UML 2.0 en trois parties que l'on retrouve quasi systématiquement : la modélisation de l'application, de l'architecture et du mapping.

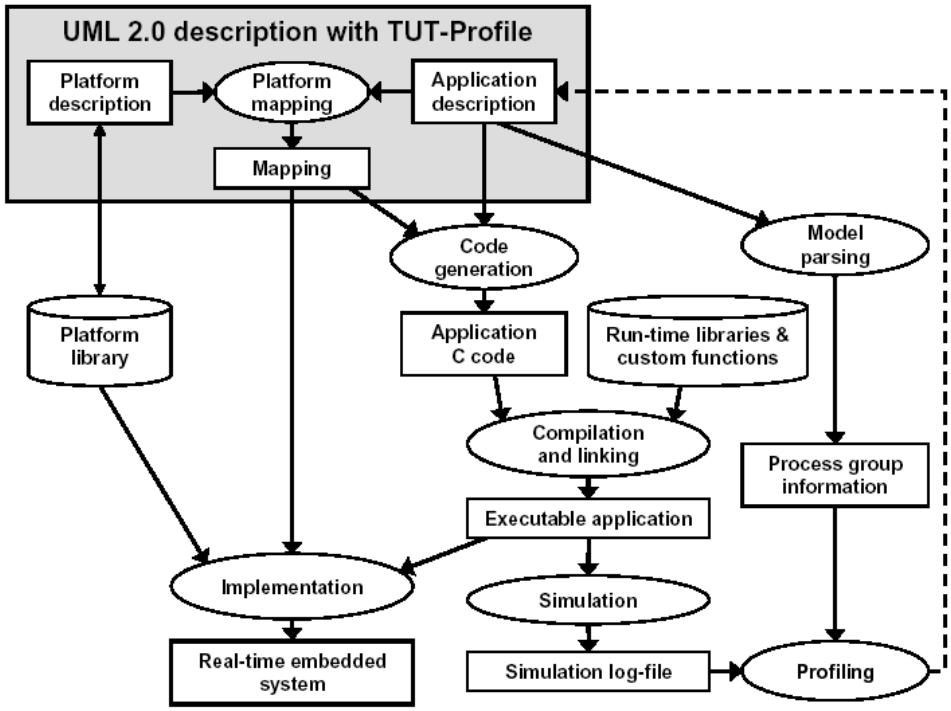


FIG. I.21 – Flot de conception basé sur le profil UML TUT

Une particularité de la méthodologie est qu'elle est centrée autour des applications. Le concepteur peut se limiter à modéliser l'application uniquement, les étapes de modélisation de l'architecture et du mapping étant automatisables grâce aux bibliothèques de composants matériels déjà synthétisés (processeur, réseaux de communication, interface logicielle, implantation d'algorithmes logiciels, etc.). Le logiciel d'outillage UML utilisé pour la modélisation en UML 2.0 est Tau G2 de Telelogic [81] qui génère automatiquement du code C pour l'application.

La modélisation UML est utilisée pour avoir un niveau de modélisation au niveau système. Elle vient en amont des étapes d'analyse, de génération de code, et de simulation et de synthèse. La première modélisation effectuée concerne la description de l'application. Ces éléments sont spécifiés hiérarchiquement dans un diagramme de classes stéréotypé. La structure de l'application est modélisée par un diagramme de structure composite proposé par UML 2.0 et son comportement est exprimé à travers des diagrammes d'états/transitions dont le modèle d'exécution correspond à des communications asynchrones étendues aux machines à états finis [82]. Ce sont ces machines à états qui permettent le calcul des performances et la vérification fonctionnelle du système. Les différents états, transitions et actions sont stéréotypés («*TimedSequence*»,«*TimedTransition*»,«*TimedMessage*») pour paramétrer les temps d'exécution des conditions et passage de messages [83].

L'architecture matérielle est construite à partir d'une librairie de composants matériels stéréotypés d'après le profil TUT. Sa modélisation est réalisée à partir d'un diagramme de structure composite. Les composants matériels utilisés sont paramétrés avec les "*tagged value*" définies

en fonction des stéréotypes existants dans le profil TUT. Les interconnexions entre les éléments de traitements sont toutefois limitées à des segments HIBI avec des wrapper HIBI [84].

L'étape de mapping revient à connecter l'application à l'architecture et s'effectue en 2 parties : la première regroupe les composants fonctionnels que le concepteur décide d'implanter sur le même composant (stéréotype «*thread*» si l'implantation est logicielle, «*hardware*» si l'implantation est matérielle), et la seconde exécute les décisions d'implantation sur le matériel (*mapping*). Le regroupement des fonctions à implanter en logiciel peut se faire sous la forme d'un thread, qui dans le cas d'utilisation d'un OS se verra affecter d'un niveau de priorité lui correspondant. Ils sont toutefois réduit à deux niveaux (high et low). Le partitionnement peut ensuite avoir lieu en choisissant quel composant matériel exécutera les groupements effectués. Dans le cas où cette étape est automatisée, le groupement s'effectue suivant les critères de charges de calcul, des communications entre les «*processes*» et de la taille des groupes.

L'outil Tau G2 peut alors générer du code C servant pour l'implantation du système. L'application est ensuite construite à partir de ce code généré auquel s'ajoute du code spécifique issu de bibliothèques (code concernant les communications), ce qui permet d'obtenir, une fois compilé, un code exécutable pour vérifier fonctionnellement, par simulation, l'application. Le profiling est réalisé au cours de la simulation et est focalisé sur les communications et l'activité des machines à états à partir des valeurs des "*tagged value*" renseignées. Ce profiling renvoie un rapport de performance sur un diagramme de séquence. L'implémentation s'obtient par compilation du code C obtenu par le compilateur de la cible désigné, pour la partie logicielle, et par la génération d'un en-tête VHDL auquel sont incorporées les sources VHDL des composants en bibliothèque. Les accès entre le logiciel et le matériel étant réalisés par appel de fonctions C externes. Les outils mis en œuvre pour réaliser ce flot sont nombreux, il y a un *parser* et un *profiler* d'application UML, mais aussi un parser d'architecture et de mapping, tous utiles à l'exploration automatique d'architecture. Il y a également deux outils d'annotation de retour des performances de l'application et des modélisations de l'architecture et du mapping, utiles à la simulation fonctionnelle et aux optimisations (exploration automatique). Il est à noter que l'ensemble du profil a été écrit et testé sur une seule application propriétaire. Cette méthode nécessite de développer toutes les interfaces.

Je tiens également à évoquer l'environnement de dévelopement intégré GASPARD, du Laboratoire d'Informatique Fondamentale de Lille (LIFL), qui a évolué en GASPARD2 pour la comodélisation graphique de SoC [85]. Il adresse tout particulièrement les applications de calcul distribué et parallèle (application de traitement du signal). Il intègre les résultats de recherche des équipes du LIFL mais aussi ceux des projets européens grâce auxquels il évolue (projets européen ITEA Sophocles, puis Prompt2Implementation). UML 2.0 est le point d'entrée des spécifications de l'application et de l'architecture, précédant un ensemble de transformations de modèles (UML2.0->ISP, UML2.0->ARCH, ISP->SystemC) aboutissant à la génération de code C++ [86]. Nous pouvons noter la représentation de l'architecture matérielle modélisée par des composants vue sous deux angles : un aspect fonctionnel (actif (DSP,FPGA), passif (mémoire, capteur), interconnexion), et un aspect structurel (élémentaire (composant unique IP), composé (processeur interconnecté à une mémoire), répétitif (motif de composants interconnectés)).

D'autres travaux de recherche intéressants gravitent autour du langage UML et de la conception. Il s'agit de la suite d'outil Xmodelink de CATS [87][88] qui permet de générer du code SystemC (comportement et fonction principale) servant à être synthétisé pour la partie matérielle et à fournir du code exécutable pour la partie logicielle. Le processus de conception est également basé sur les modèles. Les analyses statiques et dynamiques de l'application lo-

gicielle sont respectivement traduites à travers les diagrammes de cas d'utilisation (exprime concurrence), de séquence (comportement) ou de collaboration UML. L'architecture matérielle est quant à elle spécifiée à l'aide d'un outil graphique de "composants" SystemC. Cependant par manque d'informations diffusées et accessibles, nous ne pouvons porter un regard critique sur la méthode utilisée. La société STmicroelectronics utilise leur profil UML SystemC dans une suite logiciel ou l'outil Entreprise Architect de Sparkx Systems [89] lui permet de réaliser leurs diagrammes UML. Les diagrammes de classes lui permettent de décrire les modules, les interfaces et les canaux hiérarchiques. Les diagrammes de structures composite représentent les structures internes des plates-formes spécifiées. Ils précisent les relations d'interconnexion entre les différentes parties d'un bloc conteneur (channel, module). Les diagrammes de machines à états décrivent le comportement fonctionnel des process SystemC [39]. Et les diagrammes d'objets fournissent l'instanciation des plates-formes du diagramme de structure composite tout en précisant les attributs des objets. Le flot de conception au niveau système d'un SoC intégrant ce profil fait partie des travaux à venir.

Le domaine Radio Logicielle est tout particulièrement intéressé par le langage UML pour l'utiliser dans les flots de conception. La société Zeligsoft s'est d'ailleurs tournée vers UML 2.0 notamment pour la conception des architectures de communications logicielles (SCA)[16] à travers le développement de leur propre outil Component Enabler (CE) [90] similaire visuellement à ce qui a été développé dans A3S. La SCA définit le déploiement et la configuration d'une architecture dédiée à la Radio Logicielle. Elle spécifie comment assembler les composants d'un système et comment les déployer sur les plates-formes. L'approche adoptée repose sur les modèles (MDA) et permet, à partir des modèles d'applications et de plates-formes, de générer des fichiers de description xml conformes à la spécification SCA qui représente le profil du domaine. Ces fichiers peuvent être réutilisés comme point d'entrée puisqu'il sont conformes. L'accent de l'outil est mis sur la communication inter-composants. Il introduit la notion de "composant logique". Si on prend l'exemple d'un composant reconfigurable FPGA (comme typé *loadabledevice*), et d'un processeur généraliste (typé *executabledevice*) connectés sur la même carte, la configuration du FPGA nécessite le téléchargement du bitstream de configuration. Ce téléchargement contient une configuration (assimilée à la fonction à réaliser) qu'il faut exécuter par le biais par exemple d'un bout de programme logiciel provenant du processeur ou d'un composant dédié à cet effet connecté entre les deux composants. Ce bout de programme fait partie de l'architecture de communication et peut donc être représenté par un composant distinct, un "composant logique" (*logical device*). Zeligsoft apporte cette représentation qui fait partie intégrante du SCA. Chaque composant matériel doit disposer d'un "composant logique". Le concepteur peut spécifier un composant physique ou le "composant logique" va être déployé comme typé *deployondevice*, s'il ne le fait pas le "composant logique" est vu comme faisant partie du composant physique en tant que composant comme typé *devicemanager*. A priori il applique la notion de profil, non pas au langage UML mais aux fichiers de description xml qu'il est possible de générer d'après la spécification SCA, et qui décrivent l'ensemble de la modélisation et de la spécification du système. Les documents plus techniques éclaircissant les zones d'ombres sur la réalisation et le fonctionnement de l'outil sont toutefois rares, du fait du produit commercial. UML 2.0 sert donc à simplifier la création de ces fichiers xml qui peuvent s'échanger, et être réutilisés. L'architecture matérielle et le potentiel des modélisations sont cependant limités à la spécification du SCA.

Enfin, Viel [91] se focalise sur un point particulier qui est l'analyse d'un diagramme de séquence afin de déterminer l'ensemble des communications entre les différents objets (graphe de dépendance des communications) ainsi que les mécanismes de synchronisation à mettre en

place pour le bon fonctionnement du système. Et ainsi prendre en compte ces mécanismes dans le calcul des performances.

Comme nous pouvons le constater à travers cet état de l'art, le domaine est vaste et de nombreuses initiatives voient le jour. Cette effervescence est saine dans la mesure où nous sommes encore au balbutiement de ces technologies pour le domaine des systèmes temps réel embarqués. Il est donc essentiel d'explorer de nombreuses pistes avant de converger vers des solutions plus stables et opérationnelles. Le profil MARTE est d'ailleurs l'exemple qui illustre le mieux cette convergence, avec le regroupement des deux profils SPT et QoS. Avant de proposer une synthèse de ces travaux, il me semble important de positionner rapidement les travaux développés dans cette thèse.

I.2.4 Profil et méthodologie A3S

Les travaux de cette thèse ont été réalisés pour une bonne partie dans le cadre du projet RNRT A3S. La méthodologie A3S développée entre donc dans le cadre des objectifs définis par le projet, qui, nous le rappelons, étaient de proposer aux concepteurs d'applications Radio Logicielle un environnement UML lui permettant de tester différents scénarios d'implantation de composants de son application sur des modèles de plates-formes matérielles. Un outil de prototypage rapide des systèmes temps réel embarqués et en particulier ceux du domaine de la Radio Logicielle a été développé. Il offre au concepteur l'opportunité de modéliser et de spécifier à haut-niveau d'abstraction son application et son architecture matérielle à partir d'une banque d'IP existantes en sa possession. Cet outil repose essentiellement sur le langage UML 1.4 auquel s'ajoute la sémantique du profil A3S développé pour répondre aux besoins de modélisation haut-niveau, requis pour exprimer ces systèmes. Le profil A3S hérite des profils SPT et QoS, ainsi que du profil SDR, afin d'exprimer un certain nombre de contraintes temps réel déjà présentes dans ces profils. Néanmoins, ces trois profils ne suffisent pas pour pouvoir exprimer l'ensemble des contraintes nécessaires aux vérifications de cohérences de ces plates-formes ainsi qu'aux analyses de faisabilité permettant de confirmer au concepteur que ses choix d'implantation respectent le cahier des charges fixé. De même, le profil SDR reste incomplet pour exprimer convenablement les plates-formes matérielles existantes. C'est pourquoi le profil A3S apporte la sémantique supplémentaire concernant les éléments matériels et logiciels, auquel a été rajoutée la sémantique concernant la prise en charge des systèmes d'exploitation. La méthodologie de conception adoptée est PBD avec une approche MDA. Un modèle indépendant de la plate-forme représente en effet l'application, un autre modèle représente l'architecture matérielle et enfin un modèle de plate-forme dépendant de l'architecture spécifie les choix d'implantation des éléments de l'application sur l'architecture. Les éléments de l'application sont vus comme des IPs, seul l'aspect non fonctionnel est pris en compte. Les éléments de l'architecture sont vus comme des circuits intégrés individuels (avec les caractéristiques des datas sheet). Des mécanismes de vérifications permettent la vérification de cohérences de chacun des modèles réalisés, auxquels s'associent des calculs de performances. La modélisation du système ne requiert que 2 des diagrammes du langage UML, le diagramme d'activité et le diagramme de déploiement. Les informations détaillant plus particulièrement la réalisation du profil ainsi que les vérifications font l'objet du prochain chapitre.

I.2.5 Synthèse comparative

Les divers travaux présentés dans la section précédente montrent la diversité des besoins des concepteurs en matière de sémantique pour pouvoir exprimer toute la complexité d'un système électronique temps réel et de son fonctionnement. Les systèmes à concevoir sont très variés, à la fois dans les types d'applications à développer ainsi que les possibilités matérielles existantes. L'utilisation du langage UML permet de converger vers un unique langage de représentation graphique de ces systèmes. La représentation visuelle limite les erreurs d'interprétation et facilite, en général, la compréhension. Encore faut-il être d'accord sur ce qu'il est utile de représenter et sur la manière de le représenter. Une particularité du langage UML est d'offrir la possibilité de définir des sémantiques particulières et de les soumettre à standardisation pour relever les lacunes ou les incohérences des éléments représentés, et proposer à l'ensemble de la communauté une solution commune répondant au besoin du plus grand nombre. La variété des profils UML touchant aux systèmes électroniques, s'explique par l'évolution des besoins (en lien avec l'évolution technologique) et la prise de recul concernant la façon dont pouvait être représentés les systèmes. C'est pourquoi, le profil MARTE [51] tant à remplacer le profil SPT ainsi que le profil QoS, et pourrait intégrer le profil SoC, afin de ne pas avoir un nombre important de profils dont les possibilités offertes sont redondantes. Il est toutefois très difficile de tout pouvoir exprimer à partir d'un seul modèle. Dans le cas d'applications plus spécifiques à un domaine, il peut y avoir des exigences et des configurations particulières qui nécessitent le besoin d'enrichir la sémantique existante. C'est le cas du domaine de la Radio Logicielle, où la communauté est nombreuse, et dont le profil qui a été développé est en fin de processus de standardisation. D'autres en revanche comme le profil TUT [3] ont été spécialement développés pour répondre au besoin de réalisation d'un flot de conception complet, de la spécification à haut-niveau jusqu'à l'implantation.

Les différentes méthodes présentées utilisent le langage UML et ses diagrammes pour modéliser et spécifier les systèmes à haut-niveau d'abstraction au début du flot de conception. La plupart de ces méthodes utilisent d'ailleurs une version 2.x du langage UML. Le projet A3S et le travail présenté dans ce mémoire reposent sur la version précédente (1.4) mais le profil a été adapté en conséquence. Il introduit notamment la notion de port présente dans la version UML 2.0. Contrairement aux méthodologies [68] [87] qui utilisent le langage UML et ses propriétés pour décrire les éléments du système et leur comportement dans un autre langage (SystemC pour les deux cités), nous utilisons ce langage indépendamment de la sémantique d'un autre et sans utilisation d'un autre langage. L'intérêt n'est pas d'élever le niveau d'abstraction d'un autre langage, mais de pouvoir obtenir une évaluation de la faisabilité du système au plus haut-niveau d'abstraction possible (niveau système), donc dès la modélisation, comme [76]. Nos ambitions dans le cadre de ces travaux, n'étaient pas d'aller jusqu'à l'implantation, mais de déterminer au plus tôt dans le cycle de conception que les choix d'implantation retenus sont pertinents. Les travaux de l'Université de Tampere, et ceux autour de MOCCA [77] vont jusqu'à la synthèse matérielle. Le premier se limite dans les architectures matérielles qu'il peut concevoir (tout comme NEC [74] qui bride ses plates-formes suivant une architecture semi-répétitive) et utilise un profil spécifique. Le second utilise un compilateur spécifique pour effectuer la synthèse matérielle. Une des particularités de la méthodologie A3S est de ne pas considérer la fonctionnalité même des traitements qui composent l'application. En effet, le niveau d'abstraction retenu dans notre méthode suppose que le concepteur dispose de l'ensemble des IP utiles à sa conception. La garantie de la fonctionnalité ainsi que la connaissance des performances individuelles de chaque traitement sont donc connues et vérifiées avant la conception du système. Metropolis

[78] en a besoin et ajoute des fragments de code décrivant la fonction de chaque «*process*». La méthode utilisée par Fujitsu [72] s'en affranchie également, mais elle ne prend en compte par la suite que la charge des communications et le nombre d'exécution des fonctions pour ces vérifications de mapping. Dans notre cas un calcul d'ordonnançabilité, et un ensemble de vérifications ont également été intégrés afin de garantir la validité des choix d'implantation. Le tableau I.1 dresse une comparaison des différentes méthodes présentées dans ce chapitre.

L'aspect temps réel apporté par les systèmes d'exploitation n'est pas toujours pris en compte dans les méthodologies utilisant UML ([68],[76]), où alors il l'est sous une autre forme : ordonnanceur ([74][72]), mécanisme de communication ([78]), machine virtuelle([4]), ou ajout d'overheads [92]). L'Université de Nantes a proposé un modèle générique de système d'exploitation temps réel (RTOS) utilisé pour la simulation du SystemC [93]. Les caractéristiques du système d'exploitation sont portées en tant qu'attributs du processeur, les services en tant que méthodes, et les priorités des tâches sont des attributs de celles-ci. Le modèle ainsi représenté ne permet cependant pas de spécifier toutes les réalisations de RTOS possibles. Dans les méthodologies sans UML, comme celle de l'environnement δ de Mooney III et Douglas [94], générant des systèmes d'exploitation logiciel/matériel dédiés pour les SoC, les OS sont représentés uniquement par des services, réalisés en matériels (gestion mémoire (SoCDMMU), unité de vérouillage du cache (SoCDDU), unité de détection d'interblocage (SoCLC)) et les mêmes services réalisés en logiciel (sémaphores, événements, mailboxes...). Ces services sont disponibles en bibliothèque et l'outil à partir des spécifications de l'utilisateur génère les OS nécessaires. Dans notre approche de conception au niveau système, nous voulons pouvoir représenter et spécifier l'ensemble des éléments et services représentatifs des OS actuels. C'est pourquoi nous considérons les OS au même titre que des composants logiciels ou matériels, ce sont des composants propres (éléments stéréotypés différemment) représentant les systèmes d'exploitation et leur services associés. Leur sémantique et les relations de ces entités dans le profil UML A3S permettent de modéliser l'ensemble des possibilités de réalisation d'un RTOS.

I.3 Conclusion

Ce premier chapitre a permis de nous immerger dans le monde des SoC, de savoir ce qu'était ces composants émergeants qui demain seront implantés dans les appareils électroniques de notre environnement. Mais avant de se retrouver dans nos téléphones, nos voitures ou nos baladeurs, il faut réfléchir à leur conception (leurs fonctionnalités, la manière de les implanter). Les outils de conception existent en effet pour faciliter leur développement, vérifier que le concepteur ne se trompe pas, que ses choix s'avèrent exacts, enfin lui assurer les performances du produit final. Nous avons vu que la conception des systèmes électroniques s'est appuyée sur plusieurs méthodologies successives s'adaptant aux nouvelles technologies des composants matériels, ces méthodologies disposant d'un large choix de langages pour réaliser les conceptions. Nous nous sommes focalisés essentiellement sur un de ces langages, UML, car il est pré-disposé à répondre aux besoins des méthodologies de conception actuelles concernant les SoC.

Un certain nombre de ces méthodologies ont par ailleurs été présentées, mettant en avant les différentes façons d'utiliser ce langage pour les besoins de modélisation et de spécification exigés par les outils de conception à haut-niveau. La complexité des systèmes entraîne en effet l'élévation du niveau d'abstraction de la représentation de ceux-ci afin de pouvoir traiter indépendamment et en parallèle ses différentes parties (matérielles et logicielles). Nous nous sommes aperçu que même si de nombreux travaux permettent de concevoir un système à haut

niveau en utilisant le langage UML et les mécanismes d'apport de sémantiques particulières à un domaine (profil, stéréotype, "*tagged value*"), il était encore possible d'obtenir des vérifications de faisabilité d'un système à un niveau d'abstraction supérieur (niveau système sans préciser la fonctionnalité des traitements utilisés). Le prochain chapitre se focalise à présenter la manière dont nous avons utilisé ce langage et développé le profil A3S pour permettre cela.

Méthodologie	Objectif	Métamodèle/ profil	Modèle d'exécution	Partitionnement	Spécification diagramme	OS
Univ. Singapour [68]	Générer du SystemC à partir du langage UML pour la simulation	UML 2.0 stéréotypes SystemC	états / transitions	aucun	DCl, DE	non
Sloop Fujitsu [72]	Fournir un processus de conception des SoC orienté objet et appuyé sur UML	UML 2.0	réseau de processus de Kahn	manuel	DCl, DCU, DS, DStr	type d'ordonnancement ?
ACES NEC + Univ Lugano [74]	Utiliser UML pour la conception à haut-niveau des systèmes électroniques	UML 2.0	états / transitions	auto	DA, DCl, DCU, DS	simple ordonnanceur
UFRGS [76]	Fournir un outil d'exploration de l'espace de conception pour le logiciel embarqué, basé sur UML	UML 2.0 + profil SPT profil QoS	?	?	DCl, DCU, DD, DO, DS	à venir
MOCCA Univ. Mittweida [77]	Automatiser la synthèse matérielle à partir de modèles UML	UML 2.0+ stéréotypes + langage d'action MOCCA	machine à états finis avec chemin de données	dédié matériel	DC, DCU, DD	?
Metropolis [78]	Environnement de conception pour les systèmes hétérogènes	UML 1.x + profil PBD	multi-modèle	manuel	DA, DC, DCU, DE, DS	media (reseau)
Univ. Manchester [4]	Méthode de conception incrémentale	UML 2.0 + profil-RT +stereotype	flot de données synchrone	auto	DCU, DE, DS	oui ?
Univ. Tampere [83]	Prototypage à partir du langage UML	UML 2.0 + profil TUT	machine à états finis	manuel et auto.	DCl, DE, DStrCom	dédié ecos
A3S	Valider un système au niveau système à partir d'une modélisation UML	UML 1.4 + profil SPT profil QoS profil SDR	Flot de données	manuel	DA, DCl, DD	oui

DA :Diagramme d'Activité
DCl :Diagramme de Classes
DC :Diagramme de Composant
DCU :Diagramme de Cas d'Utilisation
DD :Diagramme de Déploiement
DE :Diagramme d'Etats
DO :Diagramme d'Objets
DS :Diagramme de Séquence
DStr :Diagramme de Structure
DStrCom :Diagramme de Structure Composite

TAB. I.1 – Comparaison des méthodologies de conception de SoC utilisant UML

Chapitre II

Concevoir avec UML

Beaucoup de méthodologies de conception existent et reposent sur un nombre d'outils important où chacun est spécifique à une seule tâche (spécification, vérification, modélisation, etc.). Dans cette thèse, la méthodologie de conception proposée repose sur le langage UML, et notamment sur l'utilisation des mécanismes d'extension qu'il propose (stéréotypes) pour spécifier la sémantique d'un domaine particulier (regroupée sous la forme d'un profil). Nous exploitons ses propriétés pour réaliser le début du flot de conception (les modélisations, les spécifications et les vérifications) en respectant une approche dirigée par les modèles (MDA). Cette partie présente donc notre manière d'utiliser UML pour concevoir des applications Radio Logicielle. Elle se focalise sur la création des modèles (identification des composants et leur transposition en éléments), en particulier sur l'étape d'identification de la sémantique à employer pour la modélisation des architectures matérielles, des applications logicielles et des systèmes d'exploitation des systèmes Radio Logicielle. Comme un flot de conception ne se limite pas aux étapes de modélisation et de spécification, un outillage nécessaire à l'interfaçage à d'autres outils d'analyse, en aval du flot de conception, a été mis en œuvre et est adressé à la fin du chapitre.

II.1 Approche MDA pour la conception de systèmes temps réel embarqués

II.1.1 Le flot de conception A3S

Le concepteur de systèmes électroniques d'aujourd'hui dispose d'un éventail de solutions matérielles pour réaliser un système conforme à son cahier des charges, en terme de consommation d'énergie, de performances temporelles, d'occupation d'espace, d'opportunité de reconfiguration ou d'évolutivité. Il peut utiliser des cartes déjà existantes si elles correspondent à ses besoins, ou créer une architecture matérielle dédiée à son application avec tous les choix architecturaux que cela implique. Il doit cependant trouver l'architecture matérielle la mieux adaptée à ses besoins, avec la diversité de composants matériels existants. Les ASIC sont des composants dédiés très performants mais limités à n'exécuter que ce pour quoi ils ont été conçus. Les processeurs généralistes ou spécifiques à certains traitements (DSP), sont programmables et disposent donc d'un peu de flexibilité (on peut modifier le programme mais pas l'architecture du composant). Les FPGA sont très flexibles de part la reconfiguration matérielle qu'ils offrent, et possèdent désormais des ressources de calcul importantes avec les processeurs qu'ils intègrent

(ou synthétisables). Dans le cas des SoC, toutes ces ressources utiles aux concepteurs sont dans le même composant, et l'espace de solutions architecturales se "limite" aux possibilités de combinaison des ressources potentiellement contenues. Autant dire que le nombre de solution est important. Les critères pris en compte dans ces décisions sont fonction des multiples contraintes de natures différentes provenant à la fois de l'application à réaliser et de l'architecture retenue. Avec les contraintes de temps de mise sur le marché, il est donc primordial de proposer au concepteur un outil lui permettant d'estimer l'adéquation entre des architectures et des applications, au plus tôt dans le flot de conception (niveau système), afin de valider des solutions d'implantation conformes à son cahier des charges.

La méthodologie développée et présentée dans les paragraphes suivants, est basée sur le langage UML, et suit une approche MDA. Dans la méthode proposée, il est pressenti que le concepteur de SoC réutilise l'existant avec des IP matérielles et logicielles pour réaliser ses systèmes. Dès lors, la fonctionnalité même d'un composant IP n'est plus à démontrer, car elle est assurée par le vendeur de l'IP. De même, le concepteur dispose de toutes les caractéristiques de performances, de surface et de consommation propre à chacune des IP qu'il détient. Cette thèse tend à prouver que ces informations peuvent être utilisées pour la spécification d'une partie du système à haut-niveau. Le flot de conception A3S, représenté sur la figure II.1, a été basé sur ces considérations et propose une modélisation et une spécification au niveau système en 4 étapes :

- *Modélisation de l'architecture* : le concepteur peut indifféremment débuter par la modélisation de son (ses) architecture(s) matérielle(s) ou de son (ses) application(s). Une bibliothèque de composants dit "matériels" lui propose un ensemble de composants matériels (précisés ci-après) qu'il peut utiliser pour créer son architecture. Ces différents composants sont paramétrables en fonction des configurations d'utilisation requises par l'architecte. Par exemple, le concepteur peut paramétrer la fréquence de fonctionnement de certains composants (DSP, FPGA) ou dimensionner la taille des mémoires en fonction de ses contraintes. Il peut donc modéliser sa (ses) carte(s) ou son SoC par l'assemblage de différents composants spécifiés.
- *Modélisation de l'application* : sachant que le concepteur peut réutiliser des codes logiciels de traitements déjà développés dans d'autres applications, sachant également que les applications peuvent être décomposées en tâches spécifiques, une seconde bibliothèque, de composants dit "logiciels", lui propose un ensemble de composants logiciels (précisés par la suite) représentant des fonctionnalités (Par exemple : un filtrage). Il peut donc modéliser sa (ses) application(s) de manière indépendante de toute cible architecturale par l'assemblage de composants logiciels "génériques" sous la forme d'un graphe de tâches. La modélisation réalisée est indépendante de la plate-forme matérielle (PIM). L'avantage réside dans la simplicité des modifications à apporter à l'application le cas échéant, où tout remplacement, insertion, suppression d'une fonction, ou réutilisation d'une partie de l'application peut se faire sans impact majeur sur l'architecture matérielle.
- *Déploiement* : Une fois l'application logicielle et l'architecture matérielle modélisées, le concepteur peut alors décider des choix d'implantation à mettre en œuvre et à tester pour son système. Il décide alors manuellement de la réalisation en matériel ou en logiciel de chacune des fonctions du graphe de tâches. La modélisation devient alors dépendante de l'architecture matérielle (PSM). Il se doit alors, de spécifier les contraintes imposées par ses choix architecturaux pour chaque composant logiciel instancié (des précisions sont apportées dans la suite du document).
- *Vérifications et Analyse* : A chacune des étapes précédentes, des vérifications sont ef-

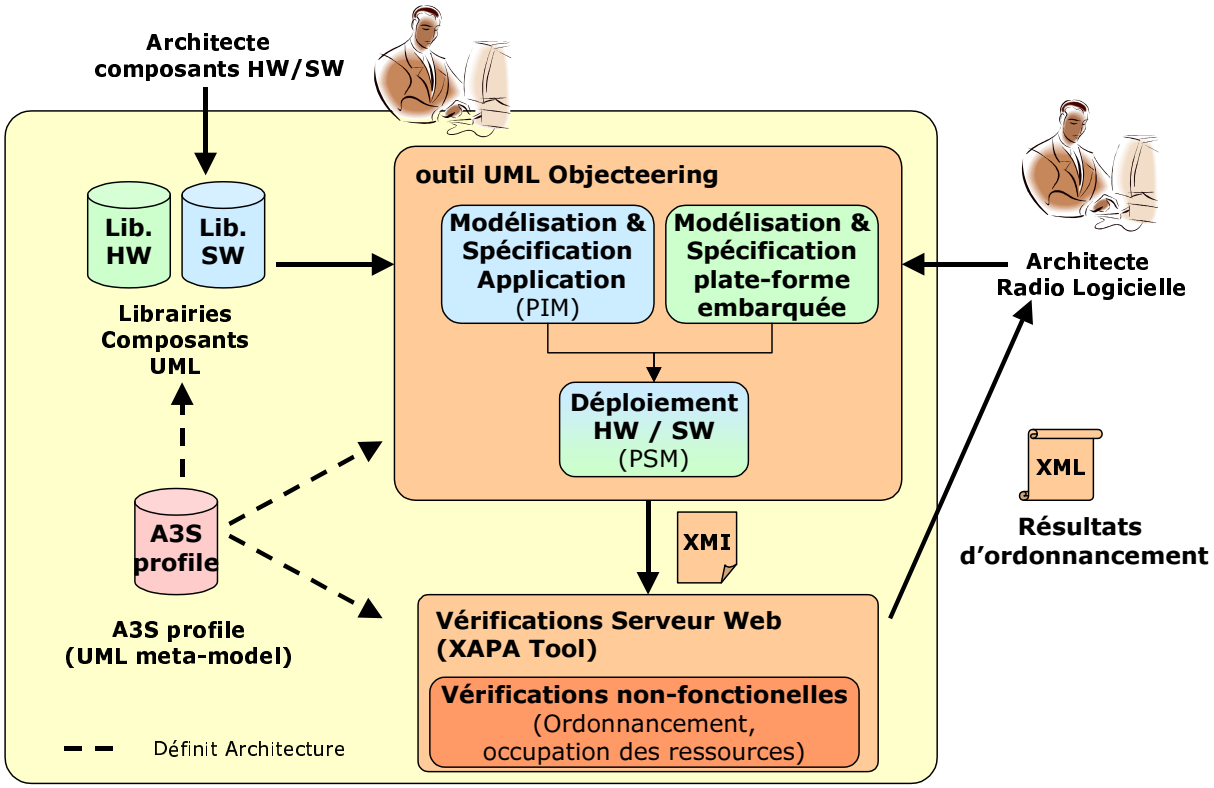


FIG. II.1 – Flot de conception A3S

fectuées pour valider les modèles réalisés. Elles avertissent le concepteur des éventuelles erreurs de modélisation ou de spécifications commises. Elles permettent de vérifier le respect des contraintes d'utilisation des composants matériels et d'approuver la cohérence des choix d'implantation retenus. Ces vérifications accomplies, les analyses de faisabilité et d'estimation de performances peuvent avoir lieu. Elles déterminent la faisabilité de l'implantation et renvoient les résultats d'ordonnançabilité et de performance.

La méthodologie de conception mise en œuvre au travers du flot présenté, peut être assimilée à du prototypage virtuel. En effet, le résultat de l'analyse de l'implantation d'une application sur une architecture est issu des paramètres spécifiés par le concepteur. Ces paramètres correspondent aux caractéristiques réelles des composants matériels ainsi qu'aux caractéristiques des IP liées aux composants logiciels, issues d'implantations réelles et fournies avec l'IP.

Ce prototypage repose sur des modélisations UML effectuées dans l'atelier de développement Objecteering, à partir de métamodèles UML définis et présentés ci-après.

II.1.2 Identification des éléments du modèle

Un système sur puce, comme il a été défini au I.1, se compose de ressources matérielles et logicielles. Notre objectif est de pouvoir le modéliser au niveau système, c.-à-d., une application exécutée sur son support d'exécution (architecture matérielle), afin d'évaluer sa faisabilité et ses performances temporelles. La faisabilité correspond, à ce niveau d'abstraction, à déterminer si le système est ordonnançable, mais aussi à vérifier que les choix d'implantation sont possibles (ressources suffisantes pour l'application). Dans toute modélisation, il est essentiel d'identifier les différentes entités utiles pour la meilleure traduction conceptuelle de la réalité,

car le modèle est établi à partir de ces entités. Leurs choix et celui de leurs paramètres associés sont notamment conditionnés par le niveau d'abstraction retenu pour représenter un système. En effet, suivant que nous voulons représenter un système électronique au niveau physique, logique ou RTL par exemple, les représentations, et donc les entités utilisées, seront différentes, leurs propriétés également. L'utilité de paramétrer ces entités, pour le domaine de la conception de SoC, est de disposer d'informations utiles et pertinentes pour les vérifications de cohérence des modèles, les calculs de performances et de faisabilité des modèles considérés, ainsi que leur raffinement.

Notre analyse d'un système SoC, sépare la partie matérielle de la partie logicielle. Le matériel se divise en divers composants identifiables par leur type (ex : mémoire) et leur implantation (ex : RAM,ROM). Le logiciel, lui, se résume à l'application qui exécute un ensemble de tâches/fonctions/traitements réalisés grâce au support matériel. Nous considérons un traitement comme le terme conceptuel pour définir une activité à mener (ex : filtrage). La fonction est l'algorithme particulier qui réalise le traitement à l'aide d'un code source compilé ou synthétisé (ex : filtre de Kalman). La tâche est une ou un ensemble de fonction(s) réalisant le traitement. Ce terme est principalement utilisé dès qu'une application s'exécute avec un système d'exploitation. Nous avons également des composants à cheval entre le logiciel et le matériel même s'ils se rapprochent d'avantage du logiciel, ce sont les systèmes d'exploitation et les intergiciels. Nous citons ces composants supplémentaires car nous considérons qu'ils font partie intégrante d'un SoC et qu'à ce titre ils doivent pouvoir être modélisés et pris en compte dans la conception. Même si les ambitions du départ étaient d'incorporer des composants intergiciels [25], les moyens engagés et les contraintes temporelles du projet pré-compétitif nous ont contraint à les écarter durant l'étude. Cependant des travaux complémentaires à A3S concernant la prise en compte des systèmes d'exploitation ont été menés pour incorporer ces composants au profil développé. Nous évoquerons ces travaux dans une autre section du document (II.2) car ils démontrent l'extensibilité du langage, et donc du profil. Nous avons donc un ensemble de **composants matériels** et un ensemble de **composants logiciels**.

Les résultats de faisabilité et de performances que nous souhaitons obtenir sont issus de techniques d'analyses quantitatives prenant en compte des propriétés non-fonctionnelles (analyse d'ordonnancement, analyse de performances) et des propriétés de qualité de services du système (vérifications, faisabilité). En effet, à ce niveau d'abstraction, nous considérons l'application comme un ensemble de traitements pré-conçus (IP), combinés entre eux pour réaliser une application, s'exécutant sur une plate-forme matérielles composée de ressources matérielles indépendantes interfaçées entres-elles. Il convient donc de vérifier que les choix d'implantation du concepteur garantissent l'ordonnançabilité du système et d'évaluer les performances prévisibles pour vérifier le respect des contraintes imposées. Il faut de plus s'assurer que les modèles soient corrects pour garantir les résultats obtenus. Une application ne peut s'exécuter sur une architecture ne disposant pas des ressources dont elle a besoin (modèle après mapping (PSM)).

C'est dans le but de répondre à ces besoins que des paramètres non-fonctionnels et des paramètres de qualité de services propres aux différents composants (matériels et logiciels) ont été identifiés. A partir de cette analyse, il convient de trouver la meilleure représentation UML, de chaque élément et de ses propriétés, à apparaître dans les modèles. Cette sémantique particulière est obtenue par la création d'un profil UML, le profil A3S. Il définit ainsi la spécification UML 1.4 utilisée dans le cadre du domaine de conception de SoC appliqué aux systèmes Radio Logicielle. Il stéréotype les éléments du métamodèle UML ("*Class*", "*Component*", etc.), utilisés pour représenter les composants matériels, et leur adjoint des "*tagged value*" correspondant à leurs propriétés.

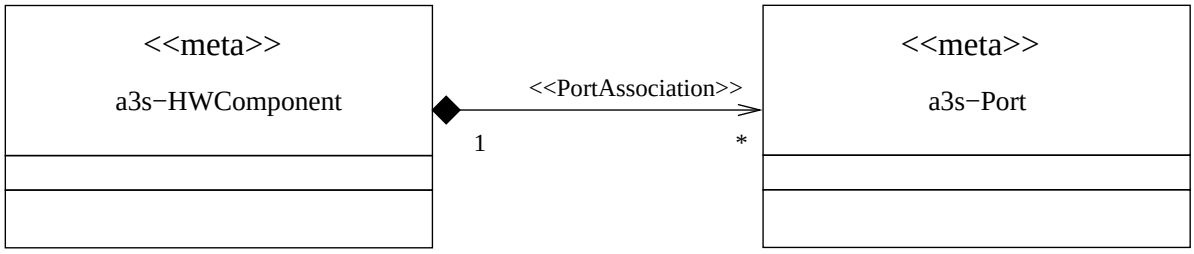


FIG. II.2 – Modèle d'un composant matériel

Le composant matériel

La représentation choisie pour une architecture matérielle est partie du constat qu'elle était constituée d'un ensemble de composants matériels hétérogènes interconnectés. Ce sont ces composants que le concepteur sélectionne et interface les uns aux autres, afin d'obtenir des ressources de calcul, de mémorisation et des vitesses d'exécution suffisantes pour le fonctionnement correct de ses applications. La diversité des composants (mémoire, interconnexion, processeur, convertisseur etc.) nous amène à extraire des similarités entre eux, afin de dégager une représentation uniforme. Le résultat de cette recherche est la décomposition en deux parties des composants matériels (excepté les composants d'interconnexion, tels les bus) : une vue externe (interface), et sa fonctionnalité.

Chaque composant matériel présente en effet, d'un point de vue externe, une interface à son environnement, lui permettant d'échanger avec d'autres composants. Cette première partie du composant correspond à son ou ses ports d'entrées/sorties («*a3s-Port*»). La seconde partie correspond à sa fonctionnalité (mémoire, processeur)(«*a3s-HWComponent*»). Sa représentation, présentée sur la figure II.2 s'attache donc à avoir un bloc fonctionnel lié à une ou plusieurs interfaces.

Afin de pouvoir effectuer des vérifications d'adéquation matérielle/logicielle, des caractéristiques de qualité de service et de performance sont attribuées aux composants matériels. Ces caractéristiques sont réparties sur chacune des deux parties des composants sous forme de paramètres. Le rôle des ports d'entrées/sorties est le transfert d'informations (de contrôles ou de données). Les caractéristiques de services sont donc, la directivité des ports, leur bande passante, leurs débits si ceux-ci sont numériques, le niveau d'impédance, le niveau d'entrée, si ceux-ci sont analogiques. Pour la partie fonctionnelle les caractéristiques sont celles que l'on trouve dans les data sheets des composants, à savoir la fréquence de fonctionnement, la capacité de mémorisation, le nombre de cellules dont ils disposent etc. Ces derniers paramètres diffèrent d'un composant à l'autre suivant sa nature. Un FPGA n'a pas les mêmes caractéristiques qu'un convertisseur analogique/numérique.

L'élément du métamodèle UML 1.4 retenu pour représenter un composant matériel a donc été naturellement le "*component*". Ce choix s'explique car le terme *component* est tout d'abord la traduction anglaise d'un composant, mais surtout sa signification UML correspond tout à fait à la représentation d'un composant matériel. Il est utilisé pour représenter une partie d'un système modulable, déployable et remplaçable. De plus, il peut, encapsuler une implantation. Ce qui convient tout à fait pour modéliser un composant matériel tel qu'un FPGA qui implante des fonctions. De plus la représentation d'une plate-forme matérielle comporte un ensemble de composants matériels interfacés les uns aux autres. Le langage UML ne propose qu'un diagramme adéquat pour cette représentation. Il s'agit du diagramme de déploiement. Celui-

ci permet principalement le déploiement de "*Component*" interconnectés par des "*Link*". La ressemblance facilite la modélisation.

Un composant matériel se modélise donc par un ensemble de deux "*Component*", comme illustré sur le métamodèle d'association des ports des composants matériels figure II.3. Pour dissocier les "*Component*" représentant l'interface du composant matériel et le "*Component*" représentant sa fonctionnalité, le profil UML A3S procure donc des stéréotypes différents. Les stéréotypes concernant la fonctionnalité tout comme l'interface du composant matériel sont dépendants de la nature du composant. Par exemple un DSP sera modélisé par un "*Component*" stéréotypé «*a3s-DSP*» et ses ports associés seront des "*Component*" stéréotypés «*a3s-DSPPort*». Ces distinctions ont lieu car les caractéristiques intrinsèques des composants matériels et de leurs ports, sont différentes d'un composant à l'autre. Le métamodèle des composants matériels, présenté figure II.4 formalise les différences en présentant les composants par famille. Il est possible alors de déterminer les caractéristiques communes des composants, de les exprimer sous la forme de stéréotypes et d'en hériter pour préciser les différences à l'aide d'autre stéréotypes. C'est par exemple le cas des DSP («*a3s-DSP*») qui se classent dans la catégorie des processeurs. Ce sont des processeurs particuliers à distinguer des cœurs de processeurs synthétisables sur FPGA. C'est pourquoi ils appartiennent aux processeurs "logiciels" («*a3s-SoftwareProcessor*»). Mais ils se distinguent des autres processeurs logiciels car ils sont dédiés à des types de traitements particuliers.

Ce sont donc ces composants matériels, proposés au concepteur dans une bibliothèque de composants matériels, qui vont lui servir pour les modélisations de ses plates-formes matérielles.

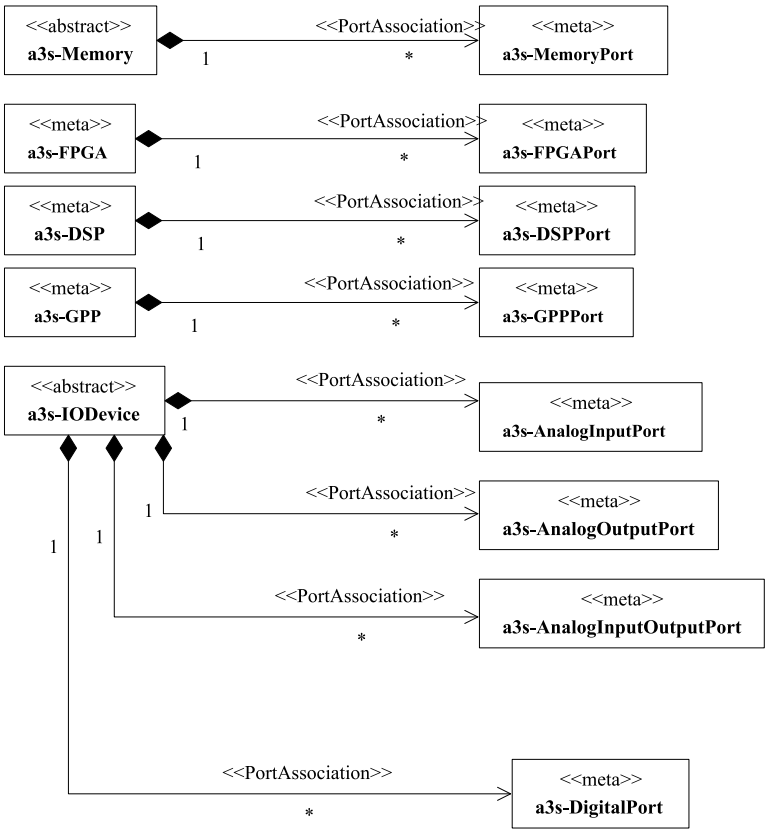


FIG. II.3 – Métamodèle des ports associés aux composants matériels

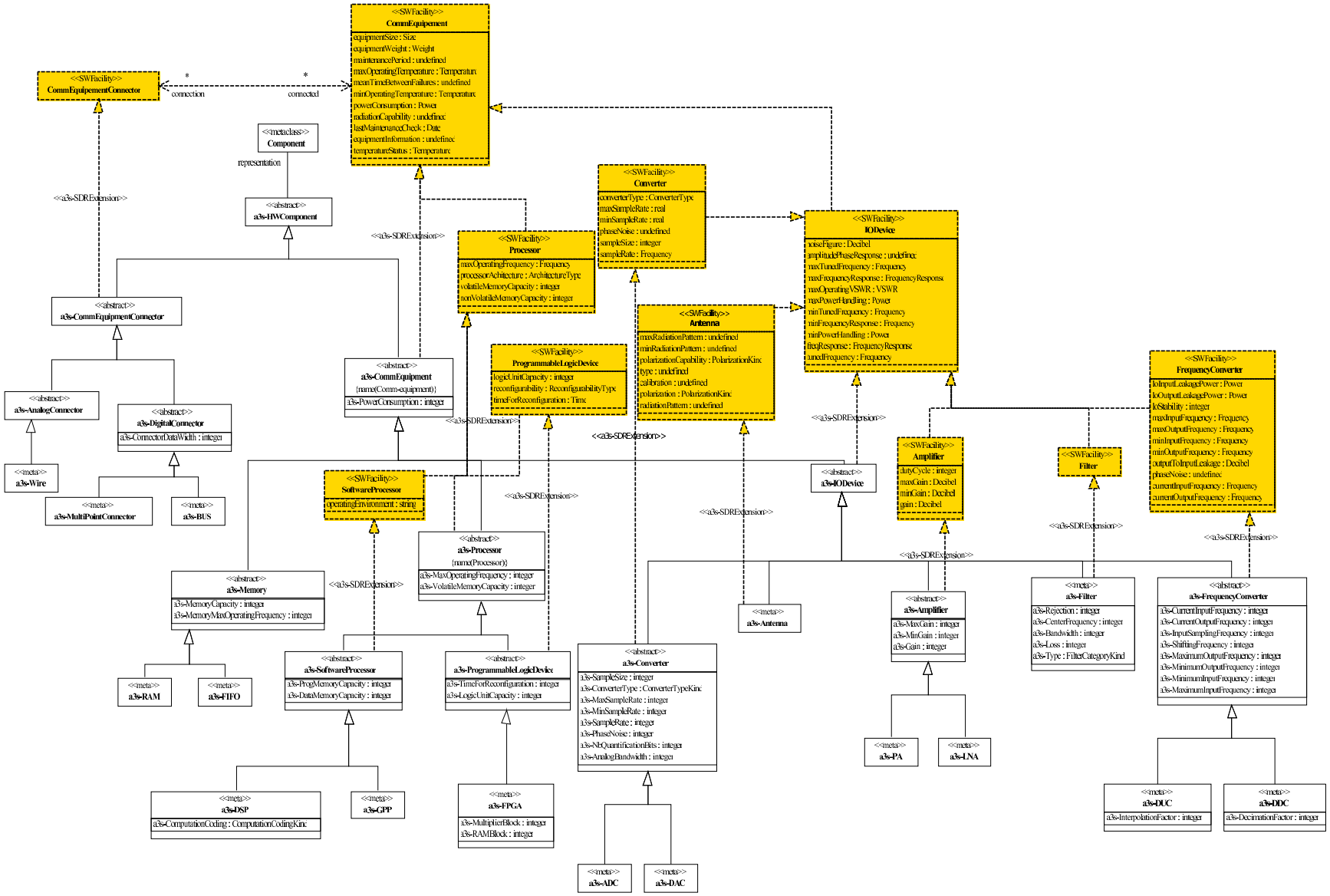


FIG. II.4 – Métamodèle des composants matériels
(en grisé (ou jaune) apparaissent les éléments du profil SDR)

Le composant logiciel

Un composant logiciel est le terme général utilisé pour parler d'un traitement logiciel, dans les différents niveaux de modélisation du système. Nous avons trois représentations du composant logiciel pour trois configurations d'utilisation différentes. Nous considérons le composant logiciel comme :

(1) une IP logicielle d'un traitement particulier. Le concepteur d'un système dispose d'un ensemble d'IP logicielles fonctionnelles déjà réalisées, et dont les caractéristiques de performances et de comportements sont connues.

(2) un traitement général dans le cas d'une représentation d'une application indépendante de tout choix d'implantation (PIM). Le composant logiciel peut être par exemple un traitement de filtrage ou de code correcteur d'erreur, sans que l'on sache quel algorithme particulier va être mis en œuvre pour réaliser le traitement.

(3) une implantation particulière du traitement utilisé dans la représentation PIM pour devenir PSM. Cette implantation fait le lien entre le (1) et le (2), où l'on identifie l'algorithme particulier utilisé et sa mise en œuvre (IP).

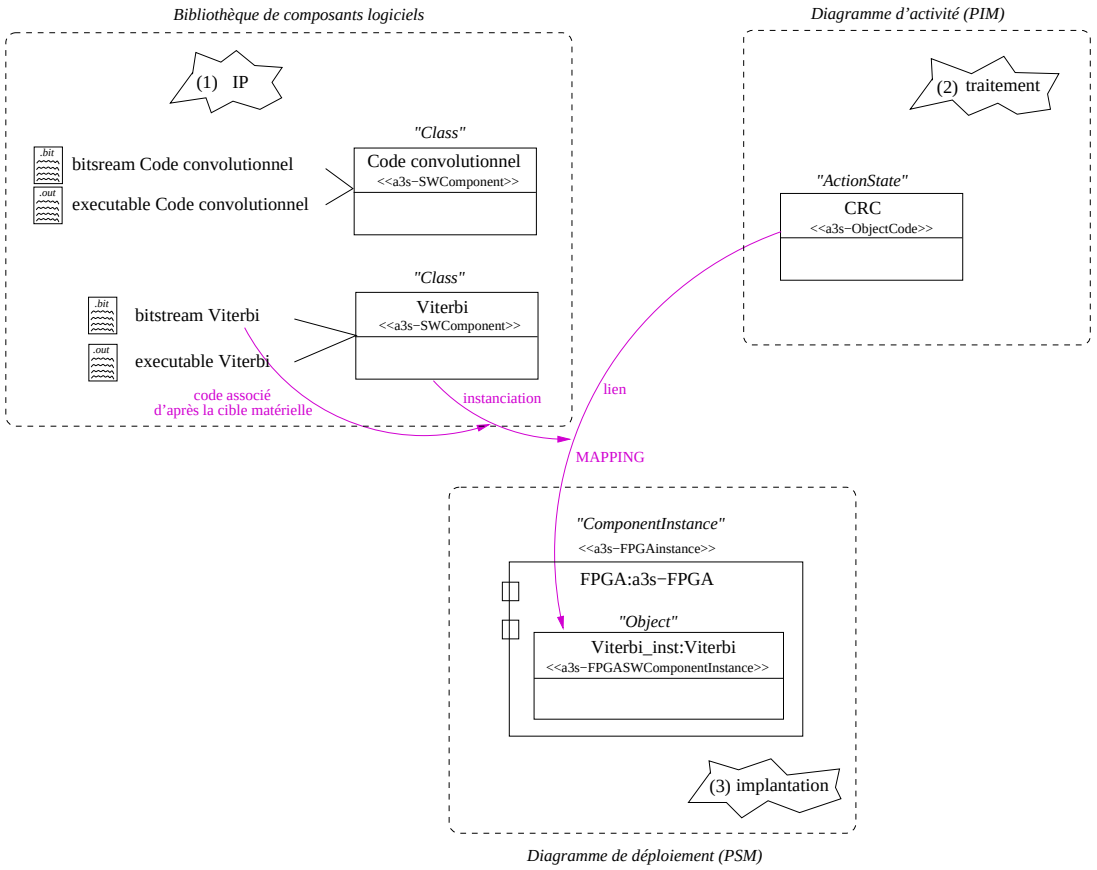


FIG. II.5 – Schéma du processus d'implantation d'un composant logiciel

Le schéma du processus d'implémentation d'un composant logiciel de la figure II.5 illustre les 3 représentations qui viennent d'être définies. Chacune de ces représentations demande à être formalisée en UML 1.4. La décomposition en deux vues d'un même composant matériel, s'applique également dans la représentation (1) du composant logiciel. En effet, les IP, reçoivent,

émettent et traitent des informations. La représentation UML du composant logiciel, retenue à ce niveau pour l'interface et le cœur du traitement (fonctionnalité), se fait au travers l'élément "*Class*" stéréotypé. Les stéréotypes retenus pour distinguer les deux sont l'«*a3s-SWIOPort*» pour les ports et l'«*a3s-SWComponent*» pour la fonctionnalité. Chaque composant, contenu dans la bibliothèque de composants logiciels, est relié à sa/ses IP, qui lui seront définitivement attribuées une fois les choix d'implantation effectués.

La représentation (2), indépendante de l'architecture (PIM), est traduite en UML à travers des "*ActionState*" stéréotypés. Les composants logiciels représentent uniquement les traitements qu'ils peuvent effectuer (ex : filtrage, code correcteur d'erreur etc.). Le concepteur d'une application connaît les différents traitements qu'il est amené à mettre en œuvre et à enchaîner pour obtenir son application. Il connaît également les contraintes à imposer aux systèmes pour le bon fonctionnement de l'ensemble de l'application. A partir de ces postulats, des paramètres non-fonctionnels sont associés à chaque composant, indépendamment de tout choix architectural, à travers le stéréotype «*a3s-ObjectCode*». Le composant logiciel est vu comme un traitement, caractérisé par des paramètres communs à tous les traitements, comme par exemple son caractère périodique, sa contrainte de temps d'exécution, son nombre d'itérations le cas échéant etc. C'est avec cette représentation que les composants logiciels s'interconnectent pour former une application.

La représentation (3), dépendant de l'architecture (PSM), s'obtient après les décisions de mapping du composant logiciel sur un composant matériel. Le composant logiciel se matérialise donc et acquière par la même occasion des contraintes supplémentaires dues à son implantation. Ses contraintes, qui se traduisent par des paramètres, sont issues des caractéristiques de l'IP liées au composant et se rajoutent aux caractéristiques initiales. Le passage au modèle PSM s'effectue à partir des composants logiciels du modèle PIM (2). Un lien s'effectue pour spécifier quel est l'algorithme utilisé et comment le concepteur veut le voir implanté (1) sur la plate-forme matérielle. Dès lors les éléments "*Class*" sont instanciés sur les "*Component*" en tant qu'"*Object*". Ces "*Object*" sont stéréotypés «*a3s-"implantation matérielle"ComponentInstance*», en fonction de la nature du composant matériel sur lequel ils ont été instanciés, ce qui leurs confèrent les paramètres supplémentaires traduisant les contraintes rajoutées.

La figure II.5 donne l'exemple d'une implantation d'un traitement de code correcteur d'erreur. En pratique ce traitement peut être réalisé de différentes manières et possède donc des performances différentes. Dans l'exemple illustré, le concepteur dispose en bibliothèque de 2 réalisations possibles (code convolutionnel et algorithme de Viterbi), à la fois pour une cible logicielle particulière (exécutable .out) et pour une cible matérielle particulière (bitstream .bit). Il dispose donc de 4 IP. Ce qui importe au concepteur c'est de pouvoir dans un premier temps modéliser son application sans se soucier de l'implantation. C'est pourquoi il représente son application à travers un diagramme d'activité présentant les traitements sous forme d'"*ActionState*". A partir de cette représentation, et du diagramme de déploiement, le concepteur peut prendre les décisions d'implantation qu'il souhaite en fonction des IP dont il dispose et des contraintes imposées. Les composants logiciels de la bibliothèque, retenus comme solution d'implémentation des traitements du diagramme d'activité, sont alors instanciés et identifient les traitements correspondant du diagramme d'activité. C'est le cas de l'algorithme de Viterbi, retenu comme solution d'implémentation du code de redondance cyclique (CRC). Comme celui-ci est implanté sur un FPGA, c'est donc le bitstream de cet algorithme, correspondant à la cible désignée, qui est jointe au composant.

II.1.3 Profil UML A3S

Le profil A3S développé, définit les métamodèles utilisés pour réaliser les modélisations et spécifier les SoC dédiés aux applications Radio Logicielle. Il structure et formalise les modèles, grâce aux éléments employés et aux relations inter-éléments spécifiées. Il définit ainsi les règles d'assemblage des éléments de chaque modélisation et permet d'établir les règles de vérifications de cohérences associées.

Ce profil fait donc apparaître l'ensemble des éléments définis, utilisables dans les étapes de modélisation et de spécification d'un flot de conception. L'avantage d'un profil UML est qu'il n'est pas figé et qu'il est aisé de réaliser des ajouts, suppressions ou modifications, afin de prendre en compte des évolutions. Ce qui, dans le cas des systèmes temps réel embarqués, est un atout pour l'adaptation à de nouvelles technologies.

Le profil A3S, dont la structure est présentée sur la figure II.6, rassemble plusieurs métamodèles qui fixent les règles et les éléments de modélisation. Ce sont des représentations graphiques issues de nos réflexions menées autour des règles d'assemblage et des propriétés de composants matériels et logiciels. Il repose sur des réflexions précédemment menées (II.1.2) et traduites par le profil Software Radio qu'il complète par une définition plus bas niveau du modèle. Il hérite, comme illustré sur la figure II.4, de certaines méta-classes définies dans ce profil.

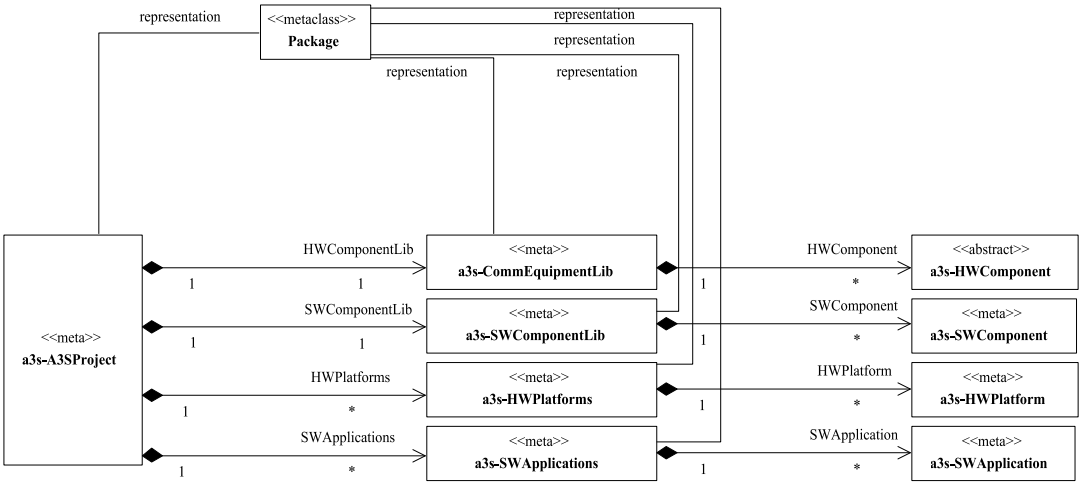


FIG. II.6 – Métamodèle du projet A3S

Les différents métamodèles sont tous liés les uns aux autres. Il est nécessaire d'avoir des métamodèles présentant les éléments qui sont utilisés dans des modélisations, et des métamodèles qui définissent les règles de modélisation. Ainsi ce ne sont pas moins de 14 métamodèles qui composent le profil A3S. Ces métamodèles reposent tous sur des diagrammes de classes traduisant les relations inter-éléments. Ces métamodèles peuvent donc être regroupés en 4 sous-ensembles :

1. le premier représente à lui tout seul ce que comprend le projet A3S, c.-à-d. l'ensemble des éléments généraux qui le constitue.
   - *métamodèle du projet A3S* : Pour toute nouvelle conception, si le concepteur désire utiliser le profil UML A3S pour effectuer ses modélisations il devra créer un projet. Ce métamodèle, présenté figure II.6 définit les implications induites par la création de ce

projet au travers un diagramme de classes. Chaque nouveau projet A3S entraîne la création de bibliothèques de composants matériels et logiciels dont les éléments respectifs vont permettre la création d'applications logicielles et d'architectures matérielles avec toutes les exigences que cela impliquent.

2. le second représente ce qui est lié à la modélisation du matériel et de ses dépendances, avec les stéréotypes associés.
   - *métamodèle des composants matériels* : présenté sur la figure II.4, il spécifie les relations d'héritage entre le métamodèle "PIM and PSM for Software Radio" et les éléments rajoutés, nécessaires à la modélisation de plate-forme SoC dédiées aux applications Radio Logicielle. Il fait également apparaître les attributs de chaque élément liés aux caractéristiques intrinsèques des composants matériels. L'association de ce métamodèle avec les deux métamodèles relatifs aux ports des composants matériels formalise ce que nous avons dénommé composant matériel.
   - *métamodèle des ports associés aux composants matériels* : présenté sur la figure II.3, il précise que chaque composant matériel possède, en fonction de son stéréotype (déterminé par sa nature), un ou plusieurs ports dont le stéréotype diffère selon la nature du composant auquel il appartient. Il est en effet utile d'identifier la nature d'un port et son propriétaire lorsque que ceux-ci sont tous des "*component*".
   - *métamodèle des ports des composants matériels* : ce second métamodèle concernant les ports, se distingue du précédent car il définit les différents stéréotypes des ports et leur hiérarchie au sens d'héritage. L'analogie avec la réalité est qu'un composant matériel possède un certain nombre de ports d'entrées/sorties. Ceci est transcrit par le métamodèle précédent. Ces ports sont de natures différentes en fonction des composants matériels auxquels ils appartiennent. En effet, un composant analogique a des ports analogiques, tandis qu'un composant numérique dispose en général de ports numériques. Certains composants disposent même des deux types de ports. C'est ce que ce métamodèle, présenté sur la figure II.7, formalise. Il est donc nécessaire de pouvoir disposer de cette information dans une modélisation, car les caractéristiques de chaque port (présentées dans la suite du document) seront différentes suivant leur nature (stéréotype).
   - *métamodèle des instances des composants matériels* : il traduit l'utilisation d'un composant matériel. La bibliothèque dispose de composants matériels que le concepteur peut ou non utiliser lorsqu'il construit son architecture matérielle. C'est cette utilisation qui est traduite par l'instanciation des composants matériels. Les différentes instanciations possibles suivant la nature du composant sont formalisées sur le métamodèle présenté figure II.8. Par exemple, lorsque le concepteur désire construire une plate-forme matérielle possédant un DSP, il va instancier un composant DSP issu de la librairie. Il obtiendra alors une instance de component stérotypée «*a3s-DSPInstance*» susceptible d'implanter des composants logiciels qui seront stéréotypés «*a3s-DSPSWComponentInstance*». Le DSP appartient à la famille des processeurs, il hérite donc des propriétés des processeurs («*a3s-ProcessorInstance*»), qui héritent eux même des CommEquipement.
   - *métamodèle des instances de ports des composants matériels* : l'utilité de ce métamodèle est la même que le métamodèle précédent mais appliquée aux ports des composants matériels.
   - *métamodèle des connexions des instances de ports des composants matériels* : il formalise l'interconnectivité des ports et est présenté sur la figure II.9. Il spécifie qu'un port

modélisé doit être connecté à un et un seul élément de connexion.

- *métamodèle de plate-forme* : présenté sur la figure II.10, il formalise les règles de construction d'une plate-forme matérielle. Il formalise les différents éléments utilisés dans une plate-forme matérielle et leurs règles d'association. Nous y retrouvons évidemment la possibilité d'avoir une ou plusieurs instances de composants matériels appartenant à une carte représentée par un bloc fonctionnel. L'ensemble d'une représentation forme une plate-forme matérielle («*a3s-HWPlatform*»).

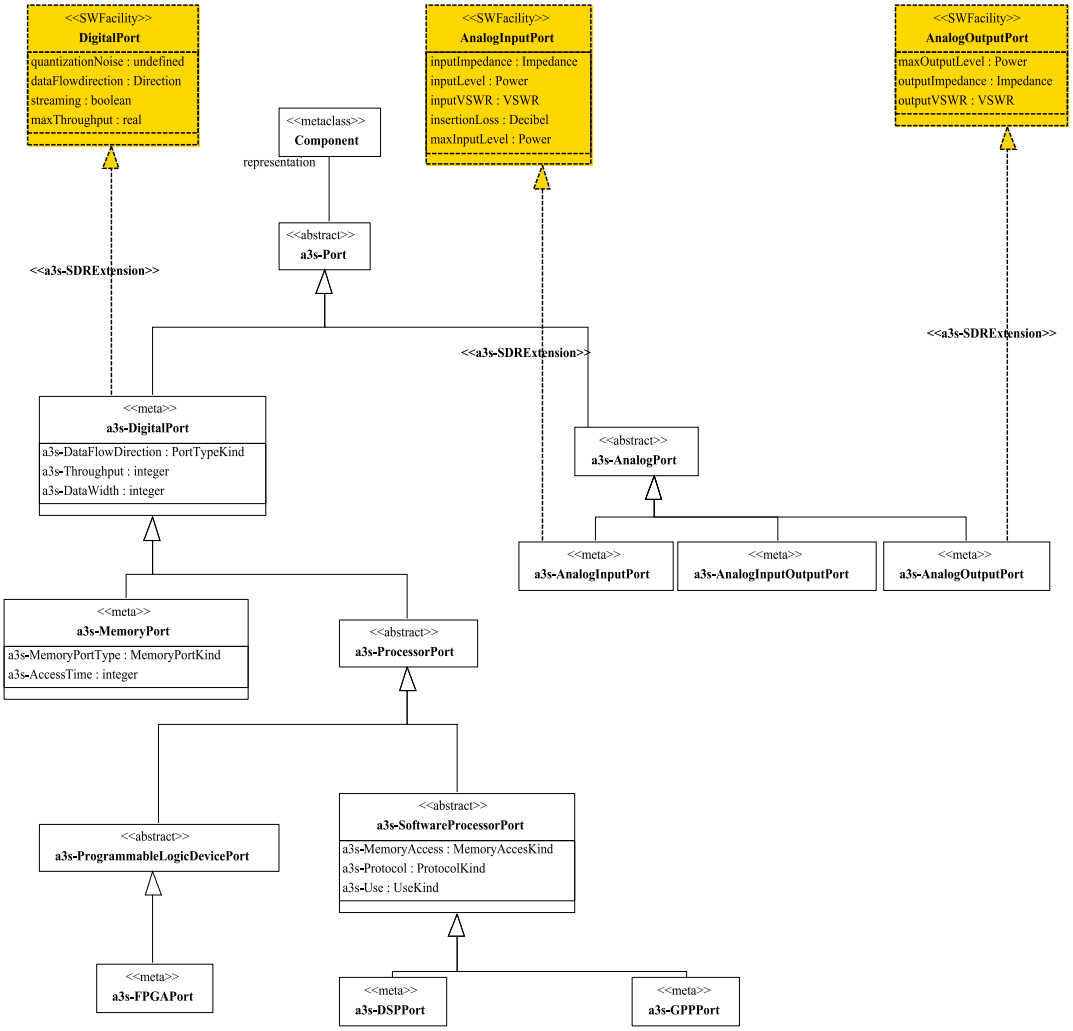


FIG. II.7 – Métamodèle des ports des composants matériels

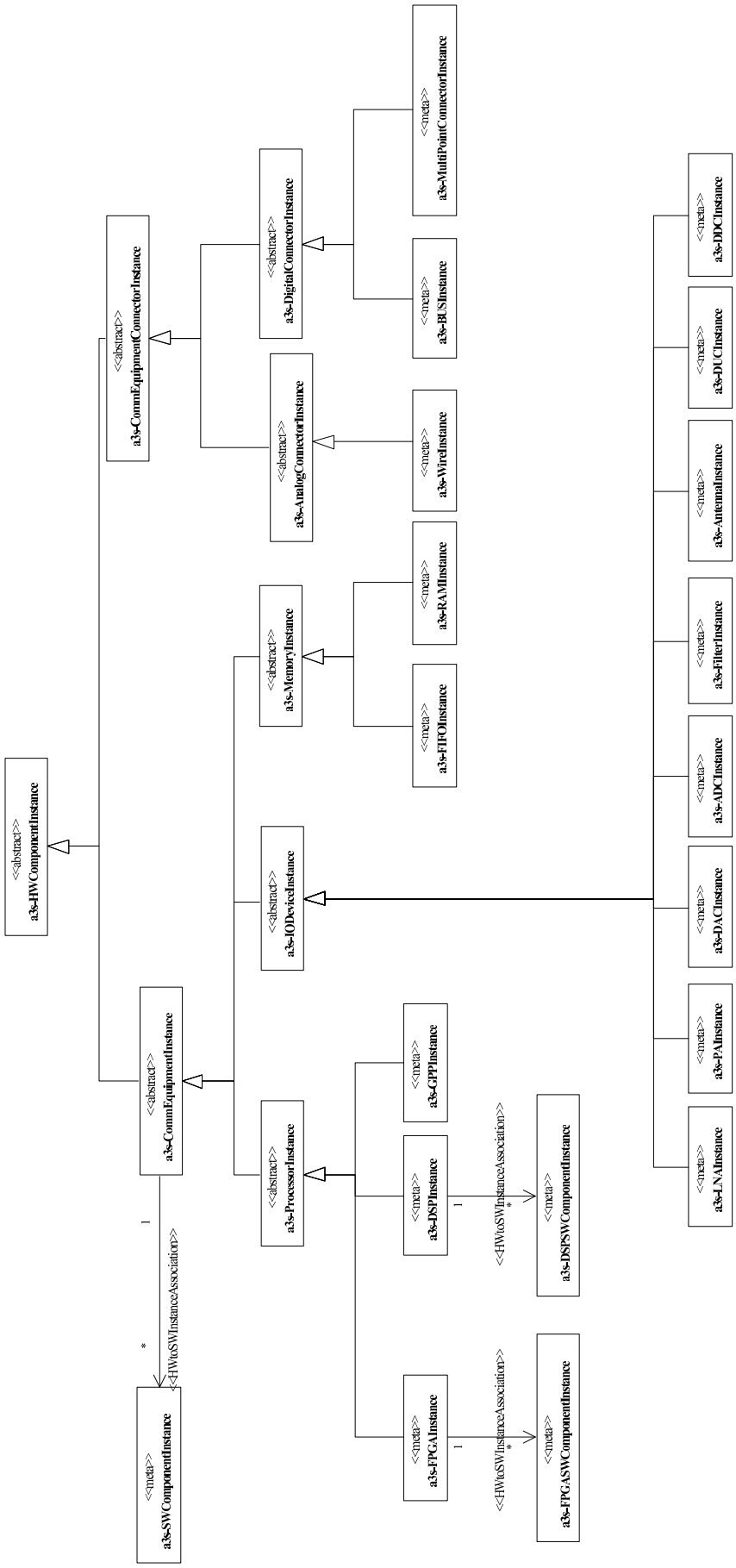


FIG. II.8 – Métamodèle des instances de composants matériels

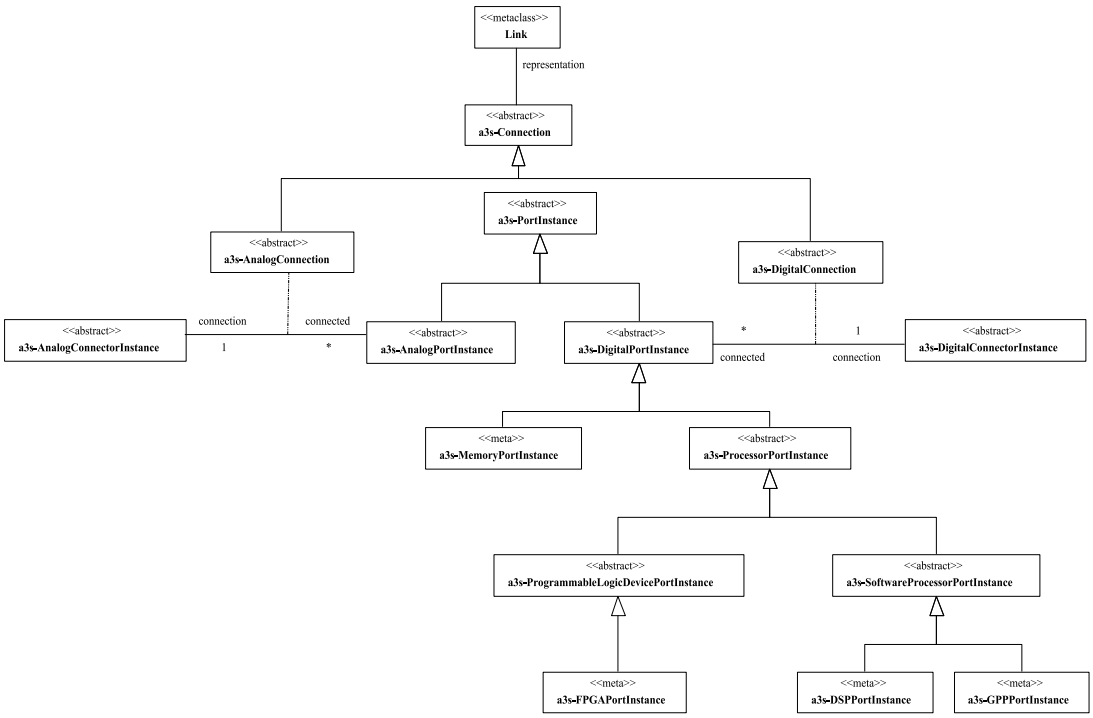


FIG. II.9 – Métamodèle des instances de ports matériels

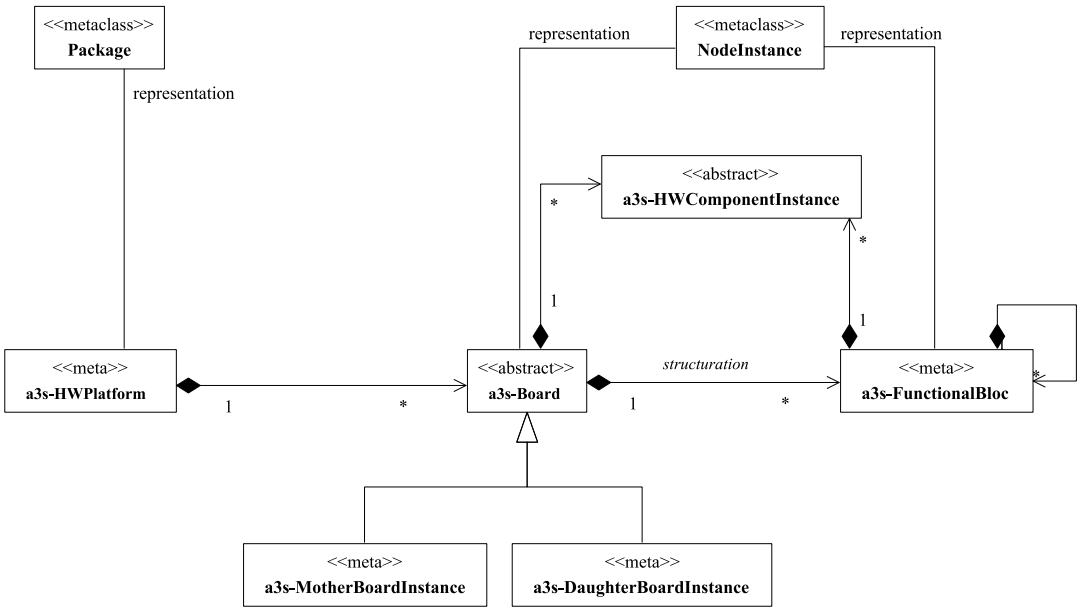


FIG. II.10 – Métamodèle de plate-forme

3. le troisième représente ce qui est lié à la modélisation du logiciel, à ses dépendances et aux dépendances relatives au déploiement des éléments logiciels sur les éléments matériels, ainsi que les stéréotypes associés.
   - *métamodèle des composants logiciels* : il précise, sur la figure II.11, que chaque composant logiciel possède un ou plusieurs ports.
   - *métamodèle des instances des composants logiciels* : il formalise les nouveaux attributs alloués aux composants logiciels lors du passage PIM vers PSM. La figure II.12,

précise l'évolution du composant logiciel lors du mapping. En effet, lorsque le concepteur effectue ses choix d'implémentation, il détermine la réalisation du traitement par un algorithme et l'implantation de cet algorithme (IP). Il est alors possible, à ce niveau de représentation, de spécifier certaines propriétés de l'IP implantée. Certaines de ces propriétés (détaillées par la suite) dépendent de la cible et d'autres non. C'est pourquoi l'instance de composant logiciel «*a3s-SWComponentInstance*» dispose d'attributs qui sont enrichis par d'autres en fonction de la nature de l'instanciation («*a3s-FPGASWComponentInstance*»)

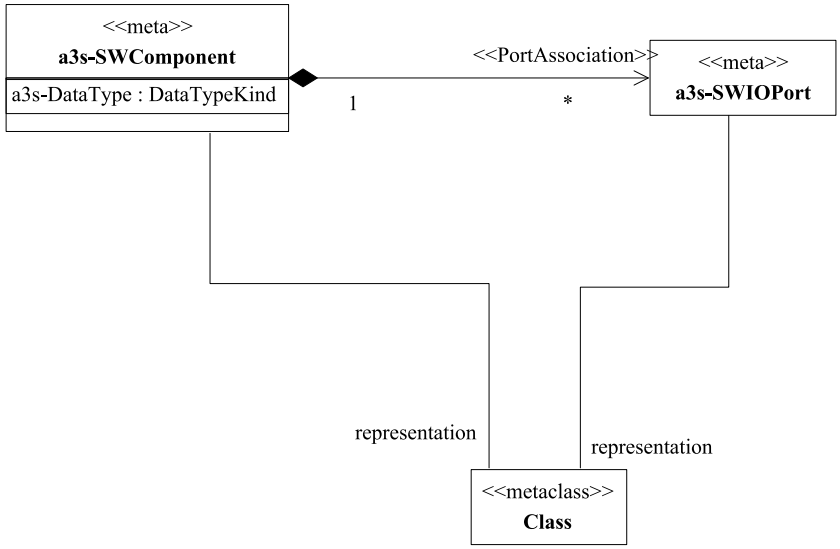


FIG. II.11 – Métamodèle des composants logiciels

- *métamodèle de l'application* : il formalise les éléments utilisés pour modéliser les applications logicielles, comme le montre la figure II.13. Il précise les différents liens et relations qui permettent de considérer le même composant logiciel dans le modèle PIM (traitement représenté par un "*ObjectCode*") et dans le modèle PSM ("*Object*" stéréotypé «*a3s-SWComponentInstance*»).
- *métamodèle statique de déploiement* : il formalise le système global une fois les décisions manuelles de partitionnement logiciel/matériel effectuées. Il est représenté sur la figure II.14. Il associe et met en relation les métamodèles de plate-forme, de système d'exploitation, d'instance de système d'exploitation et d'instance de composants logiciels. Il définit ainsi les règles de déploiement des composants logiciels sur les composants matériels, de déploiement d'un système d'exploitation sur un composant matériel et d'utilisation de ces services instanciés par des composants logiciels.

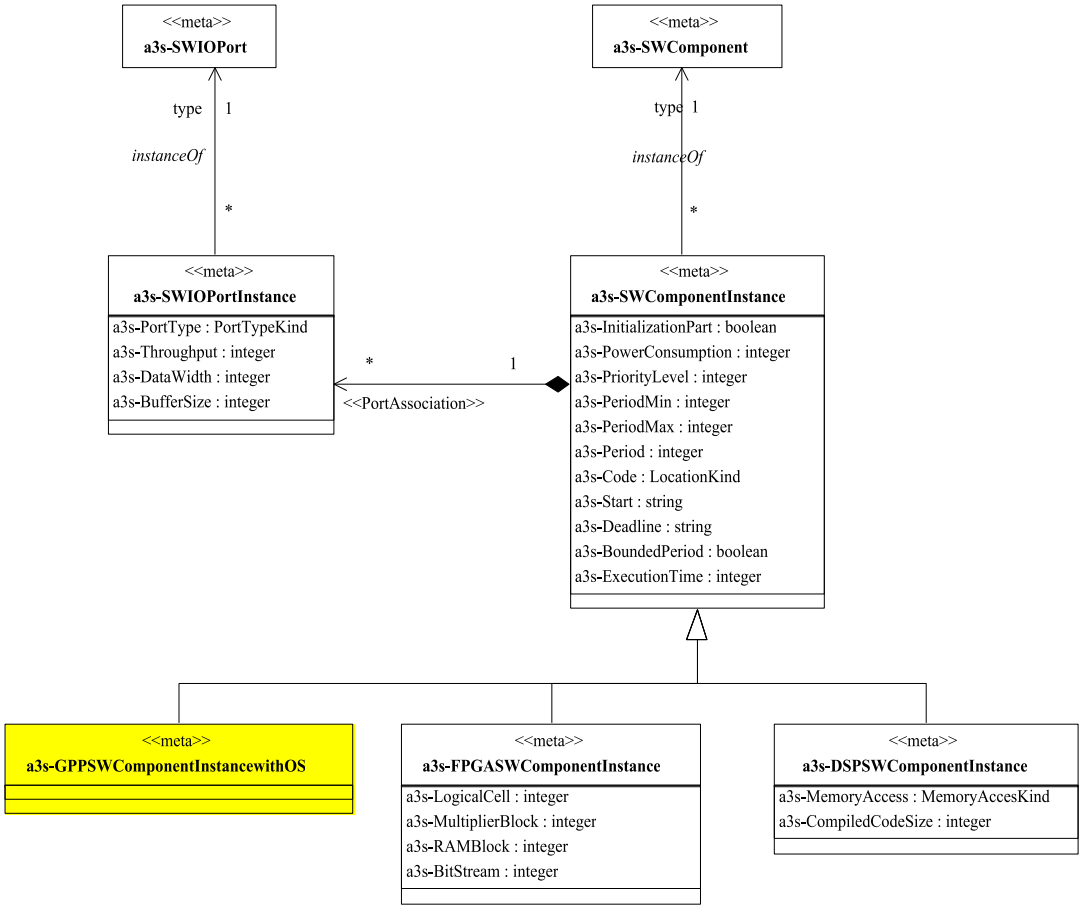


FIG. II.12 – Métamodèle des instances de composants logiciels

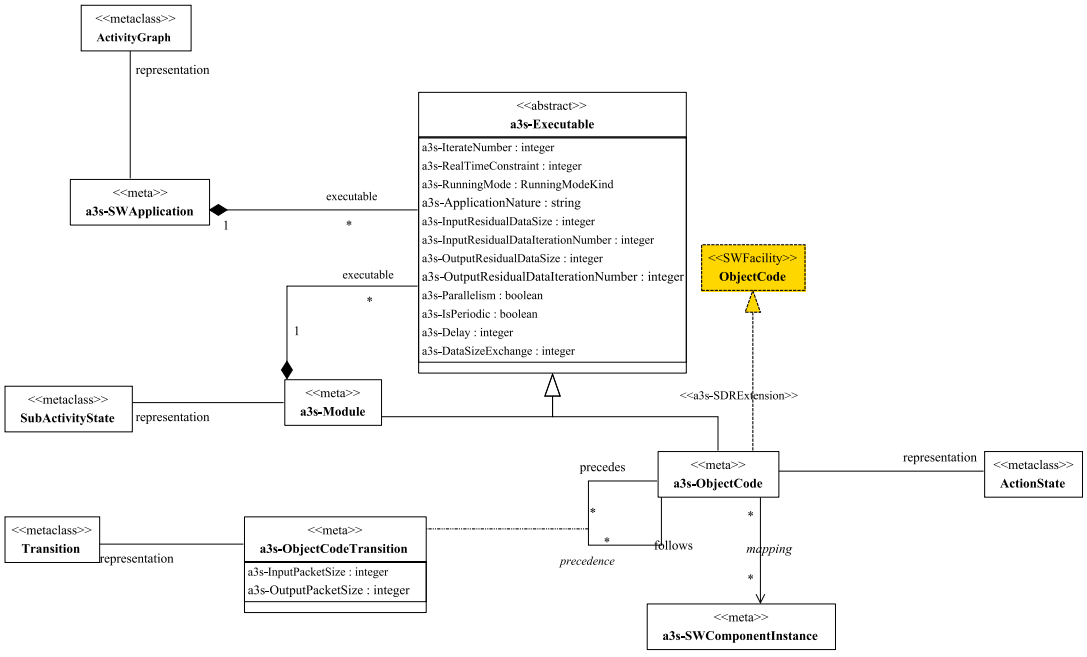


FIG. II.13 – Métamodèle de l’application

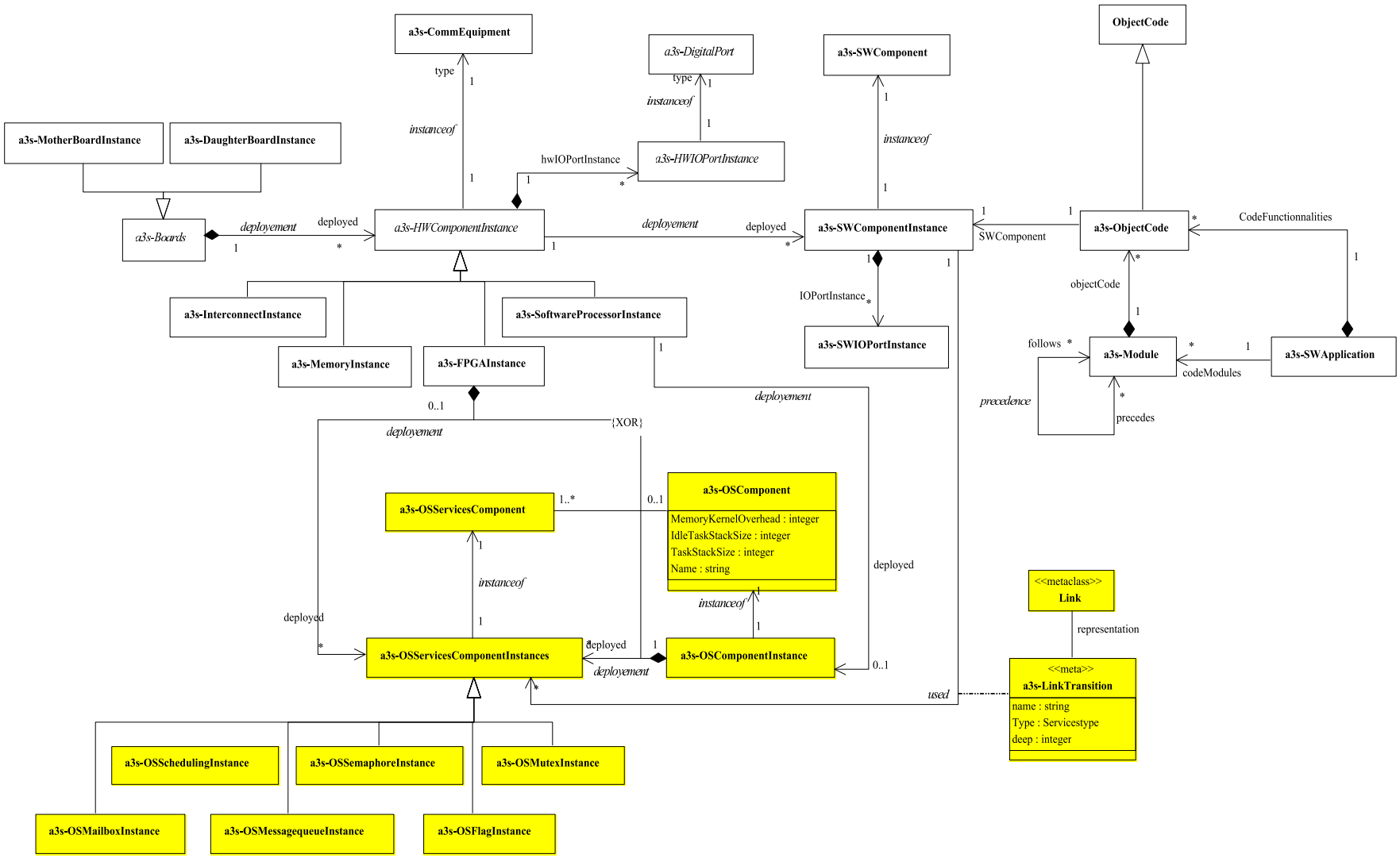


FIG. II.14 – Métamodèle statique de déploiement A3S avec OS
(en grisé (ou jaune) l'ajout des éléments liés aux systèmes d'exploitation)

4. le quatrième représente ce qui est lié à la modélisation des systèmes d'exploitation et ses dépendances, avec les stéréotypes associés.
   - *métamodèle du système d'exploitation* : il formalise les éléments représentant un système d'exploitation et ses services avec leurs attributs spécifiques. Il est présenté plus en détail dans la section II.2.2 et notamment sur la figure II.19.
   - *métamodèle des instances du système d'exploitation* : il formalise l'utilisation d'un système d'exploitation et de ses services. Il définit notamment les différents attributs attachés aux instances des services. Il est représenté sur la figure II.15.

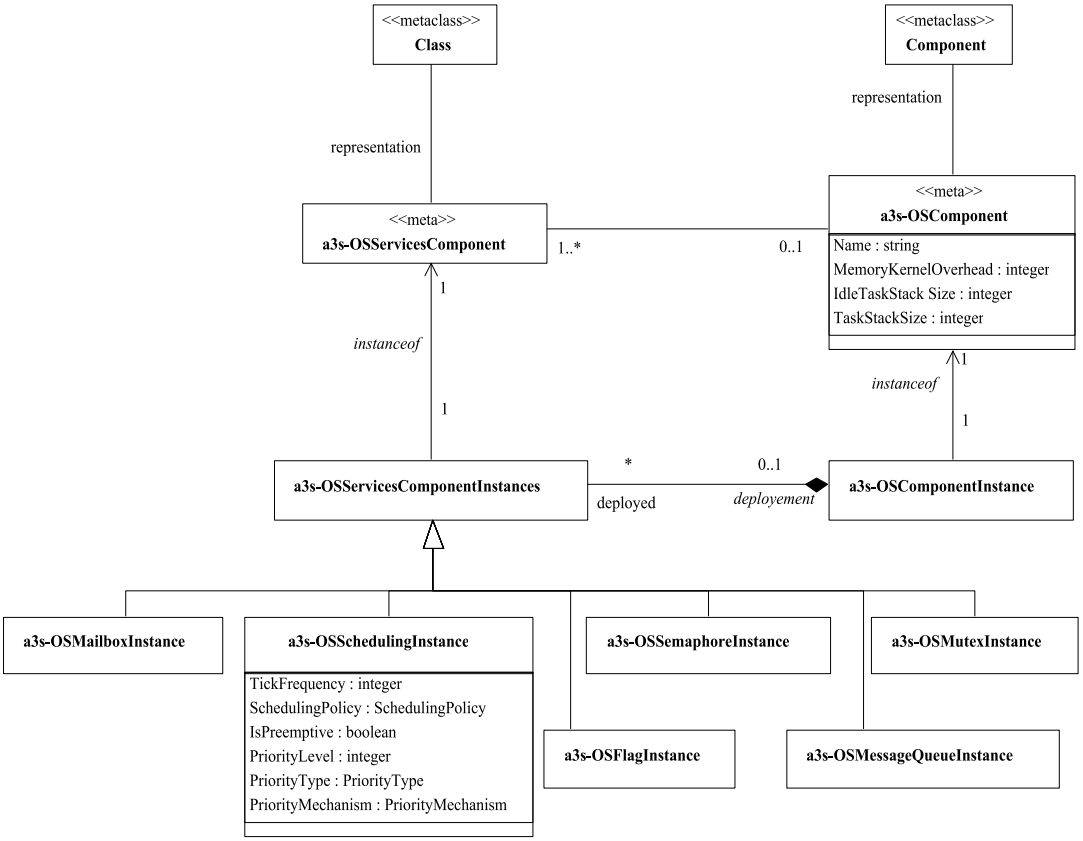


FIG. II.15 – Métamodèle de l'instance des services du système d'exploitation

Les relations entre les différents métamodèles qui composent le profil, sont données sur la figure II.16. Le profil A3S hérite des trois métamodèles cités dans la section I.2.2. Les différents métamodèles sont dissociés les uns des autres par soucis de lecture, mais ils sont tous complémentaires car ils précisent et complètent les éléments, leurs sémantiques et leurs inter/intra-relations. La présentation des métamodèles est complexe du fait de la sémantique à introduire au langage UML et des interactions entre les éléments définis. Cependant elle est nécessaire pour la compréhension de l'effort de formalisation fourni, sous-jacent à l'outil. Maintenant que les éléments UML ont été présentés, les différentes modélisations peuvent être présentées.

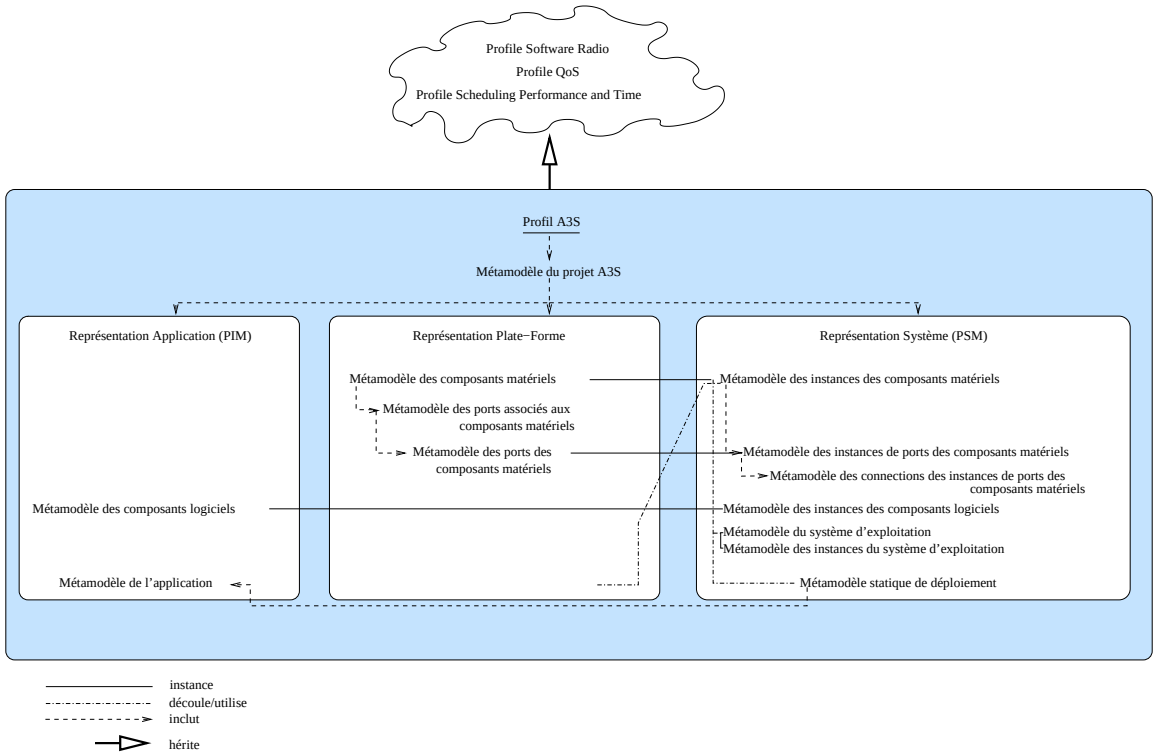


FIG. II.16 – Vue d'ensemble des métamodèles qui compose le profil

II.1.4 Modélisation des applications

Le modèle PIM

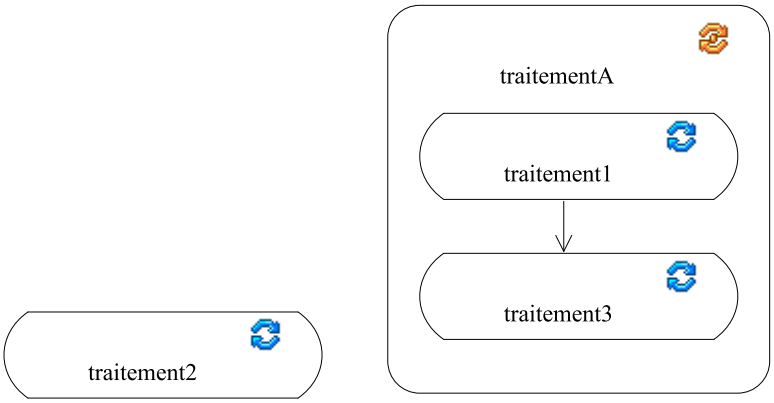
Au niveau système, nous considérons une application logicielle comme une succession d'exécution de traitements (fonctions), respectant des dépendances fixées par le travail qu'elle doit assurer. Elle peut être décomposée en un ensemble, ou plusieurs sous-ensembles de fonctions interactives qui s'échangent des informations de contrôle ou de données. La modélisation UML que nous considérons la mieux adaptée pour représenter des applications dans un modèle d'abstraction PIM, est le diagramme d'activité. Ce diagramme permet en effet, de retrouver une structure similaire et transposable à un graphe de tâches hiérarchiques. Ce que nous représentons à ce niveau, ce sont les différents traitements qui composent l'application et la manière dont ils s'exécutent et échangent leurs signaux. Nous rappelons qu'au niveau PIM, aucun choix d'implantation n'a encore été effectué, et que, par conséquent, toutes les solutions d'implantation offertes par une plate-forme sont envisageables. Ce qui est connu du système se résume donc :

- aux contraintes applicatives de celui-ci (les performances attendues),
- aux différents traitements utilisés dans la réalisation du système,
- aux interfaçages des traitements.

Nous parlons d'interfaçage pour désigner les relations d'interconnexions engendrées par l'échange d'informations entre deux ou plusieurs traitements. Nous avons précédemment présenté le composant logiciel sous trois déclinaisons, à ce niveau il est modélisé en tant que traitement.

Chaque traitement est donc représenté schématiquement par un bloc correspondant à un élément UML utilisé par les diagrammes d'activités qui est l'élément UML "*ActionState*", sté-

réotypé. La hiérarchisation des traitements s'exprime par un élément de granularité supérieure ("*SubactivityState*"). Celui-ci regroupe un ensemble d'"*ActionState*" dans un "*SubactivityState*". Notre modélisation de l'application, au niveau système, traduit l'enchaînement (séquentiel ou parallèle) des différents traitements suivant un modèle d'exécution de type *Synchronous Data Flow* (SDF) [95]. Chaque traitement ne s'exécute que si les traitements précédents, dont il dépend, lui ont envoyé le nombre de données attendues. Les échanges de signaux de contrôle (considéré comme des signaux de données) et de données sont représentés par des connecteurs (flèches), éléments "*Transition*", stéréotypés, qui stipulent le nombre d'informations à envoyer et à recevoir pour chaque traitement dépendant. Ils permettent également d'orienter le sens des communications. En effet pour chaque connexion, sont spécifiés le traitement émetteur et le traitement récepteur. Les points d'entrées et de sorties de l'application sont définis par un des paramètres des composants logiciels. L'exemple de modélisation de la figure II.17 illustre un composant logiciel générique, traitement2 (II.17(a)), et un bloc hiérarchique traitementA composé de deux traitements dépendants, traitement1 et traitement3 (II.17(b)).



(a) ActionState (b) SubactivityState

FIG. II.17 – Les éléments d'une application logicielle

Caractérisation d'une application

Le profil A3S développé permet de formaliser le langage UML pour lui fournir la sémantique particulière au domaine de conception des SoC pour les systèmes Radio Logicielle. Dans le cas de la conception des applications de ces systèmes, comme dans le cas général des applications des systèmes temps réel embarqués, il est nécessaire de les caractériser (spécifier les contraintes non-fonctionnelles) afin de prendre en compte ses caractéristiques dans les mécanismes de vérification des modèles et des calculs de performances et de faisabilité. Cette caractérisation se traduit par la définition de stéréotypes particuliers aux éléments de représentation pour leur attribuer des paramètres (sous la forme de "*tagged value*").

Ces paramètres sont liés à nos besoins de vérification et d'analyse. La modélisation de l'application PIM se doit d'être vérifiée. Elle permet l'identification et la spécification des traitements et des différents échanges d'informations intervenant dans l'application. Nous devons donc vérifier la cohérence de l'application pour identifier les erreurs ou les lacunes de la modélisation et de la spécification. La modélisation est-elle conforme au métamodèle du profil A3S ? Est-ce que les traitements s'interfacent bien entre eux ? Ne nécessitent-ils pas d'interfaçage

particulier pour la mise en forme des informations échangées ? De plus, l'étape du passage à la modélisation PSM a besoin d'informations provenant de cette modélisation (type de traitement) dans le cadre du mapping. L'analyse de faisabilité repose également sur le graphe de tâches créé à partir de cette modélisation. Un certain nombre de paramètres judicieusement identifiés sont donc nécessaires pour effectuer ces vérifications et les analyses suivantes. Ils sont liés aux connaissances que peut avoir le concepteur sur les traitements à ce niveau de modélisation. Nous allons donc avoir des paramètres généraux identiques pour tous les types de traitement, donc propres aux "*ActionState*" et aux "*SubactivityState*", et d'autres identiques aux éléments "*Transition*".

En tenant compte des considérations pré-citées, au niveau PIM, seules les contraintes non-fonctionnelles non liées à l'architecture, sont exprimées. Ces contraintes sont transcrites sous forme de paramètres ("*tagged value*") attribués à chaque "*ActionState*" et "*SubactivityState*" d'après leur stéréotype. Dans le profil A3S, nous avons formalisé deux stéréotypes différents pour ces deux éléments, mais dont les "*tagged value*" sont identiques. En effet, ces deux éléments représentent tous deux des traitements, dont seule la granularité est différente. Dans le cas des éléments "*ActionState*" et "*SubactivityState*" les stéréotypes créés sont respectivement l'«*a3s-ObjectCode*» et l'«*a3s-Module*». L'étude ci-présente, a permis d'identifier les caractéristiques non-fonctionnelles de chaque composant logiciel afin de dégager les "*tagged value*" de chaque stéréotype. Ainsi chaque élément UML qui possède le même stéréotype peut être caractérisé par des valeurs d'attributs différentes, prises en compte pour effectuer les vérifications et analyses pré-citées. Ces caractéristiques non-fonctionnelles connues du concepteur à ce niveau de modélisation vont donner les renseignements sur l'exécution des traitements, utiles à la construction du graphe de tâches. Tout d'abord il est essentiel de savoir l'ordre d'exécution des traitements, lequel débute l'application, lequel termine l'application. Il faut également savoir les quantités de données produites et consommées par chacun des traitements, afin de vérifier la cohérence des échanges d'informations. De même, cette vérification doit prendre en compte le caractère périodique ou non du traitement, ainsi que le nombre de fois que le traitement doit être répété, s'il doit s'adapter en terme de volume de données produites à transmettre. L'application sera par la suite soumise à des choix de déploiement des traitements sur des composants matériels. Dans cette étape de mapping, le modèle PSM obtenu nécessitera des vérifications de cohérence pour s'assurer de la véracité des choix d'implantation. Dans cette optique, certaines informations nécessitent d'être connues et spécifiées au niveau PIM pour valider le modèle PSM. C'est le cas des données résiduelles qui doivent être gardées en mémoire pour certains traitements. Par nature, et indépendamment de leur réalisation, les traitements nécessitent la sauvegarde d'informations entre deux itérations, afin d'être utilisées à l'itération suivante. L'information de présence et de quantité d'informations à stocker, spécifiées au niveau PIM, sera une contrainte d'implémentation à respecter au moment du mapping.

Cette réflexion nous a amené à identifier une liste de paramètres communs à tous les traitements dans le modèle PIM. Il s'agit des paramètres :

- *a3s-IterateNumber* : qui spécifie le nombre d'itérations du traitement sous la forme d'une valeur entière. Il correspond au nombre de fois que la fonction doit s'exécuter avant de retourner un résultat qui permettra aux fonctions suivantes de s'exécuter.
- *a3s-IsPeriodic* : qui spécifie si le traitement s'effectue de manière périodique ou non.
- *a3s-RealTimeConstraint* : qui spécifie la contrainte de temps d'exécution imposée au traitement (imposée par un standard).
- *a3s-RunningMode* : qui spécifie si le traitement est un des traitements considérés comme point d'entrée, point de sortie, traitement d'initialisation ou traitement intermédiaire de

l'application.

- *a3s-Delay* : qui spécifie le temps d'attente en μs avant de débuter l'exécution du traitement lorsque celui-ci est élu.
- *a3s-InputResidualDataSize, a3s-InputResidualDataIterationNumber, a3s-OutputResidualDataSize, a3s-OutputResidualDataIterationNumber* : qui spécifient les volumes de données d'entrée et de sortie à conserver en mémoire pour être utilisés dans les itérations suivantes, et qui spécifient également le nombre d'itérations pendant lesquelles il faut conserver ces quantités en mémoire.
- *a3s-DataSizeExchange* : qui spécifie la taille des informations échangées, exprimée en bits.

La liste des paramètres, propres aux composants logiciels génériques, sera enrichie de paramètres liés à leur implantation au moment du mapping de ces composants sur les composants matériels utilisés dans la modélisation d'une plate-forme matérielle.

Les différents composants logiciels constituant une application s'échangent des informations de contrôle et de données. Il est primordial de pouvoir s'assurer du bon interfaçage des différents éléments devant communiquer. C'est pourquoi les contraintes de communication inter composants logiciels sont exprimées sur l'élément d'interconnexion : la "*Transition*". Un même composant logiciel générique peut interagir avec plusieurs autres composants logiciels génériques et échanger de manière différente avec chacun d'eux. C'est la raison pour laquelle les contraintes sont spécifiques à chaque connexion. Cette particularité se traduit par l'adjonction du stéréotype «*a3s-ObjectCodeTransition*» aux "*Transition*" qui confèrent les paramètres suivants :

- *a3s-InputPacketSize* : qui spécifie le nombre d'informations à envoyer par la connexion à destination d'un seul autre composant.
- *a3s-OutputPacketSize* : qui spécifie le nombre d'informations à recevoir par la connexion de la part d'un seul destinataire.

Le choix de ces attributs plutôt que d'autres a donc été conditionné par le niveau d'abstraction système auquel nous nous sommes situés, qui nous a fait considéré que l'aspect non-fonctionnel des traitements de l'application. Les unités choisies pour chacun des paramètres ont été retenues pour spécifier les composants au plus près des conditions d'utilisation actuelles. De plus, le choix du modèle considéré (PIM) entraîne une restriction des paramètres utilisables pour caractériser les traitements. Mais surtout, ce sont les besoins des mécanismes (calcul, comparaison) mis en place pour vérifier et analyser l'application qui ont conditionné ces paramètres.

II.1.5 Modélisation des plates-formes matérielles

Représentation UML

Une plate-forme matérielle d'un système est issue de l'interconnexion de composants matériels. Le niveau d'abstraction retenu pour la modélisation de l'architecture matérielle, nous amène à représenter également une plate-forme comme un ensemble de composants interconnectés entre eux. La complexité des applications Radio Logicielle, leurs besoins en terme de ressources de calcul et de flexibilité, réclame des architectures matérielles riches en terme de composants matériels, support d'exécution de ces applications. C'est pourquoi, la liste des composants identifiés et retenus possède un certain éventail de composants matériels numériques et analogiques généralement utilisés pour ces systèmes. Nous avons identifiés, des ressources de traitements, (processeurs généralistes (GPP), processeurs spécialisés (DSP)), des éléments

reconfigurables (FPGA), des éléments de mémorisation (FIFO, RAM), des éléments d'interconnexion (bus, fil, liaison multipoint), des éléments numériques/analogiques (convertisseur A/N, N/A), des éléments analogiques (filtre), des antennes). Ces composants sont représentés en UML par des "*Component*" stéréotypés en fonction de la nature du composant. Ils possèdent donc des paramètres différents en fonction de leurs caractéristiques. La modélisation de l'architecture s'effectue par l'intermédiaire d'un diagramme de déploiement qui manipule des instances de "*Component*". Cela signifie que les composants matériels, définis et accessibles en tant que "*Component*" sont instanciés. Dès lors, les ports qui lui sont associés, sont eux aussi instanciés. L'interconnexion des différents ports des composants matériels se fait par des éléments "*Link*" neutres, puisque les composants d'interconnexion sont des "*Component*". La spécification des HWComponent effectuée dans les bibliothèques, se retrouve "figée" dans le composant instancié. Le diagramme de déploiement suit les règles de construction hiérarchiques imposées par le métamodèle (figure II.10), où un composant est instancié sur une instance de carte qui peut, elle-même être instanciée sur une instance d'une carte etc.

Caractérisation d'une architecture

Le cadre applicatif de l'étude concernant les applications radio logicielles, justifie la nature des composants matériels modélisés. Encore une fois, le profil UML A3S, définit le formalisme adapté à la conception de système Radio Logicielle qui concerne également la modélisation et la spécification des composants matériels des architectures. Le choix des paramètres caractérisant ces composants est également conditionné par le niveau d'abstraction des composants considérés, et les besoins des mécanismes de vérification et d'analyse des modélisations et du système. Une fois modélisée, il faut toujours pouvoir assurer au concepteur que la modélisation effectuée soit correcte. Dans le cas précis de l'architecture, le diagramme de déploiement est utilisé pour deux étapes du flot. La modélisation de l'architecture matérielle, et l'implantation des composants logiciels sur les composants matériels. Pour être en mesure de valider les choix d'implantation décidés par le concepteur, d'obtenir des résultats de performances, et d'assurer la véracité de la modélisation, ces composants matériels doivent disposer d'attributs particuliers renseignés par le concepteur. Le choix de ces attributs, pour chacun des composants, a été déterminé en fonction des objectifs qui viennent d'être cités. Les différents points de vérifications effectués pour chacune des modélisations, explicitent l'utilisation de ces paramètres retenus. Ces vérifications sont présentées au paragraphe II.1.7. Il est notamment important, par exemple, de connaître la fréquence de fonctionnement des composants pour déterminer les performances temporelles, de connaître la directivité des ports pour vérifier le bon interfaçage des composants, de connaître les débits supportés pour les confronter aux contraintes de l'application etc. Ainsi, les attributs identifiés et retenus se retrouvent dans les métamodèles des composants matériels (figure II.4) et de leurs ports (figure II.7) qui spécifient les différentes relations amenant à la modélisation des composants et de leurs attributs. Une liste exhaustive est également présente dans le délivrable "Typologie des ressources et exigences non fonctionnelles à vérifier" du projet A3S [96]. Voici les attributs les plus importants rassemblés par type de composant :

Pour le DSP ,

- les attributs identifiés relatifs à son interface (stéréotypé «*a3s-DSPport*») sont :
  - *a3s-DataFlowDirection* : spécifie la directivité des ports d'interconnexion. Les types énumérés potentiels pour les types de ports sont, input/output, input, output.
  - *a3s-DataWidth* : spécifie la largeur des données échangées en bits, supportée par le port.

  - *a3s-Throughput* : spécifie le débit supporté par le port en kbits/s.
  - *a3s-MemoryAccess* : spécifie la nature du mécanisme d'adressage mémoire supporté par le composant. Les types énumérés identifiés sont DMA ou CPU.
  - *a3s-Protocol* : spécifie le type de port potentiellement utilisable. Les types énumérés identifiés sont Parallel, Ethernet, Serial, PCI, other
  - *a3s-Use* : spécifie le type d'usage du processeur, co-processor, processor, general.
- Les attributs relatifs à son fonctionnement (stéréotypé «*a3s-DSP*») sont :
  - *a3s-PowerConsumption* : spécifie la consommation statique du composant en milliWatt.
  - *a3s-MaxOperatingFrequency* : spécifie la fréquence maximale de fonctionnement en méga hertz (Mhz).
  - *a3s-VolatileMemoryCapacity* : spécifie la taille de mémoire totale disponible par le composant en kbits.
  - *a3s-ProgMemoryCapacity* : spécifie la taille de mémoire allouée au programme en kbits.
  - *a3s-DataMemoryCapacity* : spécifie la taille de mémoire allouée aux données en kbits.
  - *a3s-ComputationCoding* : spécifie le type de calcul supporté par le composant. Les types énumérés identifiés sont *fixed* (pour le calcul en virgule fixe) et *float* (pour le calcul en virgule flottante).

Pour le FPGA,

- les attributs identifiés relatifs à son interface sont :
  - *a3s-DataFlowDirection* : spécifie la directivité des ports d'interconnexion. Les types énumérés potentiels pour les types de ports sont, input/output, input, output.
  - *a3s-DataWidth* : spécifie la largeur des données échangées en bits, supportée par le port.
  - *a3s-Throughput* : spécifie le débit supporté par le port en kbits/s.
- Les attributs relatifs à son fonctionnement sont :
  - *a3s-PowerConsumption* : spécifie la consommation statique du composant en milliWatt.
  - *a3s-MaxOperatingFrequency* : spécifie la fréquence maximale de fonctionnement en méga hertz (Mhz).
  - *a3s-VolatileMemoryCapacity* : spécifie la taille de mémoire totale disponible par le composant en kbits.
  - *a3s-TimeForReconfiguration* : spécifie le temps nécessaire à la reconfiguration total du composant en μs.
  - *a3s-LogicUnitCapacity* : spécifie le nombre d'unités logiques de base disponibles sur le composant.
  - *a3s-MultiplierBlock* : spécifie le nombre de blocs multiplieurs disponibles sur le composant.
  - *a3s-RAMBlock* : spécifie le nombre de blocs mémoires disponibles sur le composant.

Pour les éléments de mémorisation,

- les attributs identifiés relatifs à son interface sont :
  - *a3s-DataFlowDirection* : spécifie la directivité des ports d'interconnexion. Les types énumérés potentiels pour les types de ports sont, input/output, input, output.
  - *a3s-DataWidth* : spécifie la largeur des données échangées en bits, supportée par le port.
  - *a3s-Throughput* : spécifie le débit supporté par le port en kbits/s.
  - *a3s-MemoryPortType* : spécifie le type de port pour l'adressage mémoire. Les types énumérés définis sont dual et simple.

  - *a3s-AccessTime* : spécifie le temps d'accès mémoire exprimé en millisecondes.
- Les attributs relatifs à son fonctionnement sont :
  - *a3s-PowerConsumption* : spécifie la consommation statique du composant en milliWatt.
  - *a3s-MaxOperatingFrequency* : spécifie la fréquence maximale de fonctionnement en méga hertz (Mhz).
  - *a3s-MemoryCapacity* : spécifie la taille de mémoire disponible par le composant en kbits.

Pour les éléments d'interconnexion,

- Ils ne possèdent que des attributs relatifs à leurs interfaçages car ils remplissent la fonction de jonction entre deux composants (interconnexion). Les bus et les connecteurs multi-points sont les seuls éléments à posséder un attribut de part leurs stéréotypes. Il s'agit de la largeur des données qu'ils supportent, *Data width*, exprimée en bits. Le fil dont l'élément est stéréotypé «*a3s-wire*» n'a aucun paramètre.

II.1.6 Choix d'implantations

Lorsque le concepteur dispose de la modélisation de son application logicielle et de la modélisation de sa plate-forme matérielle, il peut commencer à prendre des décisions d'implantation des composants logiciels génériques qui composent l'application. Toutes les implantations sont envisageables mais pas forcément réalisables ou conformes aux contraintes imposées.

Le schéma du processus d'implantation d'un composant logiciel de la figure II.5 a été présenté pour identifier les différentes modélisations du composant logiciel en fonction de son niveau d'abstraction. Les 3 points de vue du composant logiciel sont représentés par trois éléments UML différents, résumés dans le tableau II.1.

Composant logiciel	Représentation UML	Stéréotype associé
IP (1)	"*Class*"	«*a3s-SWComponent*»
traitement (2)	"*ActionState*"	«*a3s-ObjectCode*»
	"*SubactivityState*"	«*a3s-Module*»
implantation (3)	"*Object*"	«*a3s-(HW)SWComponentInstance*»

TAB. II.1 – Représentations du composant logiciel

L'implantation d'un composant logiciel sur un composant matériel découle du choix manuel du concepteur. A partir, du diagramme d'activité (modélisation de l'application) le concepteur décide donc de l'algorithme (1) correspondant au traitement et choisit son support d'exécution (2) sur la plate-forme matérielle modélisée de son choix. Ces 2 actions combinées se traduisent donc en UML par :

(1) la création d'un lien de correspondance entre l'"*ActionState*" ou "*SubactivityState*" et la "*Class*" (IP) correspondante,

(2) l'instanciation de la "*Class*", qui devient un "*Object*" stéréotypés «*a3s-"implantation matérielle(HW)"ComponentInstance*», sur l'instance du composant matériel désignée. Le "implantation matérielle(HW)" correspond dans la réalité au type de composant matériel (FPGA, DSP, GPP) sur lequel la "*Class*" est instanciée.

Dans le modèle indépendant de la plate-forme, le composant est en effet générique, il ne représente que le type de traitement (ex : un filtrage) et non pas sa réalisation (ex : filtre de Kalman). Dans le modèle PSM, son identification se traduit par la génération d'un lien qui rattache

les deux composants (traitement/algorithme). Les choix d'implantation, induits par le déploiement, ajoutent des contraintes supplémentaires qui sont rapportées sur l'instance du composant logiciel générique. Les stéréotypes des "*Object*" sont en effet dépendants de la cible, car certaines contraintes rajoutées sont spécifiques au composant sur lequel est implanté un traitement. Comme le formalise le métamodèle des instances de composants logiciels sur la figure II.12, les objets ont tous un nombre de paramètres identiques liés à l'implantation, mais en fonction des implantations particulières (sur DSP, FPGA) ils possèdent des contraintes supplémentaires.

Ces paramètres supplémentaires identifiés, issus des contraintes liées à la cible d'implantation, servent dans les processus de vérification des modélisations, et sont utiles aux processus d'analyse de faisabilité du système. Certains autres ont été ajoutés (tel a3s-PowerConsumption) dans une éventuelle extension des possibilités de l'outil (calcul de la consommation), ou dans la perspective de déjà disposer de ce paramètre qui serait nécessaire à un autre outil s'interfaçant avec le notre. Il faut pouvoir renseigner le concepteur sur d'éventuelles erreurs de choix d'implémentation. Par exemple, si le concepteur implémente deux algorithmes qui s'échangent des informations (d'après le diagramme d'activité) sur des composants matériels qui ne sont pas connectés entre eux, l'échange ne peut avoir lieu. Il faut alors pouvoir le spécifier au concepteur. De même, il se peut que les ressources matérielles nécessaires au traitement soient insuffisantes pour l'implantation choisie. Le calcul de faisabilité inclut un calcul d'ordonnancement qui nécessite de connaître les temps d'exécution de chaque traitement mis en œuvre. Ce temps d'exécution étant dépendant de l'implantation, c'est donc un paramètre que les "*Object*" doivent posséder. Ce temps ne peut être connu au niveau PIM (seul une contrainte peut être avancée). Voici la liste des attributs identifiés, classés suivant qu'ils sont généraux ou dépendants de leur implantation, concernant les "*Object*".

Les "*Object*" stéréotypés «*a3s-SWComponentInstance*», possèdent les attributs généraux :

- *a3s-InitializationPart* : qui spécifie par un booléen si le composant logiciel possèdent ou non une partie d'initialisation.
- *a3s-PowerConsumption* : qui spécifie la consommation en milliWatt de l'exécution du traitement.
- *a3s-PriorityLevel* : qui spécifie le niveau de priorité donné au traitement.
- *a3s-PeriodMin* : qui spécifie la période minimale du traitement en μs.
- *a3s-PeriodMax* : qui spécifie la période maximale du traitement en μs
- *a3s-Period* : qui spécifie la durée d'une période du traitement en μs.
- *a3s-Code* : qui spécifie la localisation du code d'exécution du traitement. Les types énumérés identifiés sont "internal" et "external".
- *a3s-Start* : qui spécifie le temps à partir duquel le traitement peut débuter. Ce temps est référencé par rapport au début de l'exécution de l'application.
- *a3s-Deadline* : qui spécifie le temps à partir duquel le traitement doit être terminé. Ce temps est référencé par rapport au début de l'exécution de l'application.
- *a3s-BoundedPeriod* : qui spécifie si la période est bornée, auquel cas les valeurs min et max de la période sont à prendre en compte.
- *a3s-ExecutionTime* : qui spécifie le temps d'exécution du traitement par rapport à l'implantation désignée incluant le transfert des informations. Ce temps est spécifié en μs.

En fonction de leur implantation sur DSP ou FPGA, les stéréotypes respectifs «*a3s-DSPSWComponentInstance*» et «*a3s-FPGASWComponentInstance*» ajoutent des contraintes supplémentaires.

Ce sont pour les «*a3s-DSPSWComponentInstance*» :

- *a3s-MemoryAccess* : qui spécifie le type de mécanisme d'adressage mémoire utilisé par

le traitement. Les types énumérés identifiés sont les mêmes que pour l'attribut du même nom du «*a3s-DSP*».

- *a3s-CompiledCodeSize* : qui spécifie la taille du programme compilé correspondant au traitement. Cette taille est exprimée en kbits.

et pour les «*a3s-FPGASWComponentInstance*» :

- *a3s-LogicalCell* : qui spécifie le nombre de cellules logiques nécessaires au traitement.
- *a3s-MultiplierBlock* : qui spécifie le nombre de blocs multiplieurs nécessaires au traitement.
- *a3s-RAMBlock* : qui spécifie le nombre de blocs RAM nécessaires au traitement.
- *a3s-Bitstream* : qui spécifie la taille du bitstream en kbits.

II.1.7 Vérifications et règles associées

Modéliser une application et une architecture à partir de composants, et effectuer la spécification en renseignant les paramètres dans des champs, semblent très facile. Néanmoins, le concepteur peut faire des erreurs rapidement par inadvertance, que ce soit sur les valeurs qu'il renseigne ou sur les modélisations qu'il crée. C'est pourquoi la vérification des modélisations, afin de garantir une réalisation valide répondant aux contraintes du concepteur, a toute son importance. Chacune des modélisations se doit d'être vérifiée pour éviter les erreurs ou les oublis. Les premières vérifications doivent donc assurer au concepteur que ses modélisations sont correctes. C'est à partir de modélisations saines qu'il est possible de vérifier la faisabilité du système, puis de procéder aux analyses assurant la validité et le respect des contraintes spécifiées par le concepteur.

Deux types de règles ont été établies pour effectuer ces vérifications. Nous avons les règles de vérification concernant la structure des modélisations. Et nous avons les règles concernant les cohérences des valeurs des attributs des composants renseignées par le concepteur. Ces deux types de règles correspondent aux conditions de construction que nous avons identifiées, en rapport avec la réalisation réelle des cartes électroniques et la cohérence des applications.

Les règles et vérifications associées s'intéressent donc :

- Au niveau de la modélisation de la plate-forme : il faut assurer au concepteur que la plate-forme modélisée soit correcte. Les vérifications effectuées portent donc principalement sur les connexions entre les différents composants utilisés dans la modélisation (diagramme de déploiement). Chaque connexion doit être vérifiée pour signaler toute incohérence due à une incompatibilité matérielle. C'est à dire que l'outil vérifie que les débits, le dimensionnement des données, l'orientation des connexions, pour chacun des composants matériels connectés deux à deux sont corrects. Les attributs de chacun des composants matériels vont donc être comparés pour valider les modélisations de chacune des plates-formes. Tous les attributs doivent être renseignés, pour pouvoir être pris en compte. Si des erreurs sont détectées, l'outil génère un rapport d'erreur qui les identifie. De même il est vérifié que les ports des composants matériels sont connectés à des ressources de communication, c.-à-d. des éléments dont les stéréotypes héritent des «*a3s-CommEquipementConnectorInstance*». Dans la réalité, il y a en effet, des ressources de communications entre les ports de deux composants connectés. Au niveau des spécifications fournies par le concepteur, les valeurs renseignées pour chaque paramètre doivent être réalistes. Ces premières vérifications valident la modélisation de la plate-forme matérielle.
- Au niveau de la modélisation de l'application (PIM) : il faut vérifier que le diagramme

d'activité soit correct. Les vérifications portent principalement sur la cohérence du modèle : tous les éléments présents sont-ils tous connectés ? , tous les paramètres ont-ils été spécifiés ? Il faut s'assurer, tout comme pour la modélisation de la plate-forme, que les paramètres renseignés ne soient pas farfelus. La cohérence impose également que la quantité d'information entre deux traitements soit égales (données produites = données consommées).

- Après l'implantation (PSM) : la faisabilité se vérifie en comparant les caractéristiques des traitements (modèle PIM), les valeurs des contraintes d'implantation ajoutées (modèle PSM) et les ressources disponibles pour chacun des composants (modèle de plate-forme). La première vérification concerne l'identification des éléments implantés. Il faut s'assurer que le concepteur a implanté tous les traitements du diagramme d'activité. Il faut ensuite vérifier que le choix d'implémentation est cohérent. Si le traitement nécessite l'utilisation de mémoire, il faut que le composant sur lequel il est instancié en dispose ou ait la possibilité physique d'en adresser une. Il faut également s'assurer que les composants d'interconnexion supportent les débits et les largeurs de données spécifiées dans le modèle PIM, en fonction des choix d'implantation. Suivant l'implantation, notamment sur FPGA, les algorithmes nécessitent l'utilisation de cellules logiques, de blocs dédiés, dont les quantités sont spécifiées. Il faut s'assurer que l'implantation du nombre d'algorithmes déterminés par le concepteur soient réalisables (que le FPGA soit proportionné en conséquence). Lorsque toutes les modélisations sont correctes, alors une ultime vérification concernant la faisabilité est effectuée. Il s'agit de vérifier l'ordonnançabilité du système tel qu'il a été modélisé. Pour cela, les valeurs de période, de temps d'exécution, de deadline, de priorité, ainsi que l'analyse de l'implantation, sont utilisées pour le calcul.

A titre d'exemple, pour une fonction implantée sur un DSP ou GPP, l'outil va vérifier que la quantité de mémoire disponible est suffisante pour stocker le fichier de programmation du DSP. Pour une fonction implantée sur un FPGA, l'outil va vérifier que les quantités de ressources encore disponibles (nombre de cellules logiques, blocs rams, blocs multiplieurs) par rapport à celles nécessitées par le(s) composant(s) logiciel(s), conviennent à son (leurs) implantation(s).

Voici la liste des vérifications qui correspondent à l'ensemble des règles. Nous avons :

⋆ des vérifications basiques concernant
  – le stéréotypage des éléments :
    – chaque élément doit posséder un stéréotype issu du profil A3S.
    – les stéréotypes des instances des éléments doivent être corrects, ils doivent correspondre aux bons éléments.(ex : «*a3s-FPGA*» -> «*a3s-FPGAInstance*»).
    – les stéréotypes des ports attribués aux composants (logiciels/matériels) doivent correspondre au type de port que le composant peut supporter (ex : «*a3s-DSPport*» pour les ports d'un «*a3s-DSP*»).
  – les valeurs des paramètres ("*tagged value*") de tous les composants doivent être renseignées.
  – les valeurs de certains paramètres de composants doivent être cohérentes, à la fois dans les valeurs fournies mais aussi comparativement aux valeurs des mêmes attributs des autres composants auxquels ils sont connectés (ex : largeur des données produites, largeur du bus).
  – les éléments instanciés :
    – Il faut que tous les traitements du modèle PIM soient mappés sur la plate-forme matérielle. Ce qui signifie qu'il faut que l'on retrouve les instances correspondant à

chacun des traitements ("*ActionSate*","*Class*","*Object*", stéréotypés).
  - Il faut que tous les ports des composants (logiciels et matériels) définis dans les bibliothèques **et** utilisés dans les modélisations soient instanciés avec l'instance de leur composant.

⋆ des vérifications structurelles
  - tous les ports de composants matériels doivent être connectés à un composant matériel de communication de type «*a3s-CommEquipmentConnector*». Et que celui-ci soit conforme au type de port (ex : un port analogique (resp. numérique) ne peut être connecté à un élément de communication analogique (resp. numérique)).
  - les ports soient intelligiblement connectés. A savoir qu'il n'y ait pas que des ports d'entrées connectés ensembles sans avoir au moins 1 port de sortie.

⋆ vérifications d'implantation
  - les valeurs de certains paramètres de ressources des composants matériels doivent être en adéquation avec la somme des ressources nécessaires aux différents composants logiciels qui y sont instanciés (ex : nombre de cellules logiques disponibles sur un FPGA et la somme du nombre de cellules logiques utilisées par les IP qui y sont instanciées).

Les contraintes vérifiées peuvent donc être de simples vérifications numériques (valeur d'un attribut différente de zéro), ou le résultat de calculs plus "complexes" (vérification du nombre de données consommées par rapport au nombre d'exécutions de la fonction (production / période consommateur = consommation / période producteur)). Dans tous les cas les vérifications effectuées dans Objecteering sont des vérifications basiques. D'autres règles sont vérifiées au sein de l'outil XAPAT, lorsque celui-ci analyse le fichier XMI généré, qui transcrit toutes les modélisations en langage XML. Il s'agit des règles appliquées une fois le déploiement effectué. Elles vérifient notamment que **tous** les traitements du diagramme d'activité ont été implantés sur les (bons) composants d'une plate-forme matérielle (composants autorisés par les métamodèles du profil). Ces vérifications sont du même type que celles données dans l'exemple de la figure II.18. Ce sont des recherches d'éléments, des sauvegardes dans des listes, des comparaisons de stéréotypes.

```
∀ élément stéréotypé «a3s-ObjectCode» et «a3s-Module»
   Rechercher son lien avec «a3s-SWComponentInstance»
      Si lien_existe = OK
         vérifier nature implantation
            Si nature implantation ≠ a3s-CommEquipment
               ajouter liste_erreur : erreur = mauvaise implantation de a3s-ObjectCode
            Sinon vérification = correcte
            FinSi
      Sinon
         ajouter liste_erreur : erreur = absence d'implantation de a3s-ObjectCode
      FinSi
   FinRechercher

   Si vérification = correcte
      retourner succès vérification
   Sinon retourner liste-erreur
  FinSi
```

FIG. II.18 – Exemple d'algorithme de vérification

D'autres vérifications sont également implantées, elles concernent l'analyse de faisabilité et

notamment l'ordonnançabilité du système. Un algorithme d'ordonnancement statique à priorité fixe est utilisé (provenant des travaux de thèse d'Hedi Tmar [97]) pour déterminer si l'ordonnancement est réalisable, et s'il l'est, effectue l'ordonnancement. Cet algorithme utilise pour son exécution les différentes valeurs des paramètres des composants (période, temps d'exécution, priorité...) ainsi que les informations de mapping issues de la modélisation PSM (quelle tâche est implanté sur quelle ressource).

II.2 Intégration des systèmes d'exploitation

Le projet A3S a permis de définir un ensemble de modèles permettant de modéliser et de spécifier à la fois les applications logicielles, les architectures matérielles, et de définir les règles de mapping concernant le déploiement des composants logiciels sur les composants matériels. Les applications complexes des systèmes électroniques temps réel utilisent également les systèmes d'exploitation des processeurs. Or dans le modèle A3S du projet, cet aspect n'a pas été abordé. Au vu du modèle déja produit et dans la continuité du projet, il est utile de pouvoir également modéliser le(s) système(s) d'exploitation afin de prendre en compte les *overhead* ajoutés. La méthodologie utilisée correspond à l'approche MDA où l'on dissocie la modélisation de l'application logicielle de celle de l'architecture matérielle. La même approche est conservée pour la modélisation des systèmes d'exploitation. En effet un concepteur de systèmes qui connaît le système qu'il veut réaliser (l'application, la plate-forme matérielle), connaît également le(s) système(s) d'exploitation présent(s) sur le(s) processeur(s). Il peut donc spécifier l'OS de la même manière qu'il spécifie ses composants matériels et logiciels. Le rajout de la modélisation des OS dans le profil A3S montre toute la flexibilité du langage UML et l'extensibilité de notre modélisation.

II.2.1 Identification des éléments

Au vu de la littérature traitant des systèmes d'exploitation [92][98][99] ceux-ci peuvent être décomposables en 2 parties.

- Un OS, c'est tout d'abord un ensemble de **services** plus ou moins riches en fonction des systèmes d'exploitation. En effet, tous les OS ne possèdent pas tous les mêmes services, il y en a des plus légers que d'autres. Cependant on peut distinguer des services principaux présents pour tous, car ils caractérisent les OS (ex : service d'ordonnancement, les services mettant en œuvre des communications d'inter-processus (IPC)).
- Un **OS**, c'est aussi du code logiciel permettant de gérer l'exécution d'un ensemble de tâches tout en tenant compte de l'environnement dans lequel l'application s'exécute. Une multitude d'OS existent, notamment dans les applications temps réel embarquées [100]. Cette multitude traduit également la variété des codes utilisés pour réaliser les différents OS, ainsi que la variété des supports matériels utilisés pour leurs exécutions. En effet, les systèmes d'exploitation sont présents sur les processeurs généralistes (GPP), mais également, avec des services limités, sur DSP, et leurs services peuvent également être implantés en matériel.

La modélisation des systèmes d'exploitation en deux parties s'explique. Une modélisation concernant les systèmes d'exploitation doit posséder la sémantique nécessaire pour pouvoir représenter tous les OS existants. Lorsque nous faisons un tour d'horizon sur l'implantation de ces OS, nous remarquons qu'ils sont majoritairement logiciels, mais il existe également des

mécanismes propres aux OS réalisés en matériel. De plus, beaucoup de systèmes d'exploitation sont portables et peuvent être implantés sur des cibles matérielles différentes. C'est le cas de ThreadX [101], MicroC/OS II [102], etc., ceci se traduit par des portions de code différentes nécessaires au portage du système d'exploitation sur une cible particulière. Donc en fonction de la cible, pour un même OS, le portage diffère. Les systèmes d'exploitation se différencient également les uns des autres, par les divers mécanismes de synchronisation et processus d'inter-communication qu'ils utilisent, et sont codés différemment. Ils ne possèdent donc pas tous les mêmes services, et ces services peuvent être développés de manières différentes. Sans système d'exploitation présent, nous considérons les éventuels mécanismes similaires aux services d'un OS comme des traitements spécifiques et non des services d'OS. Ainsi, dans une perspective de génération de code, il est possible d'allouer différents codes spécifiques au portage, identifiés à partir de la cible matérielle sur laquelle l'OS doit s'exécuter (idem pour les différents services supportés par l'OS). L'ajout d'un système d'exploitation dans une application vient impacter l'analyse d'ordonnançabilité et vient occuper des ressources. C'est pourquoi il est nécessaire de prendre en compte l'effet des OS dans cette analyse et vérifier l'état des ressources disponibles en fonction de l'ensemble des choix d'implantation effectués. Cet effet dépend des propriétés spécifiques à chaque service, propres à un OS, qui peuvent être exprimés sous forme de paramètres. Une étude de cet impact, présentée plus loin dans ce document (IV.1), nous a permis d'identifier un certain nombre de paramètres à faire apparaître dans le modèle pour être intégrés au calcul d'ordonnançabilité. Ces paramètres apparaissent en tant que *tagged value* associés aux stéréotypes dans le modèle.

Au regard de toutes ces considérations, la représentation des systèmes d'exploitation s'effectue au moyen de différents composants identifiés. Il faut des composants représentatifs des OS, et des composants représentatifs des différents services qu'un OS propose. Les composants OS sont représentés en UML par des "*Component*". Les composants représentant les services sont représentés par des éléments "*Class*", comme défini par le métamodèle des systèmes d'exploitation figure II.19. Ceci se justifie, par le fait que les services sont relatifs à un OS. Ils sont donc à instancier sur un OS, ce qui se traduit en UML par des "*Class*" instanciées sur des instances de "*Component*" comme explicité un peu plus loin dans ce document.

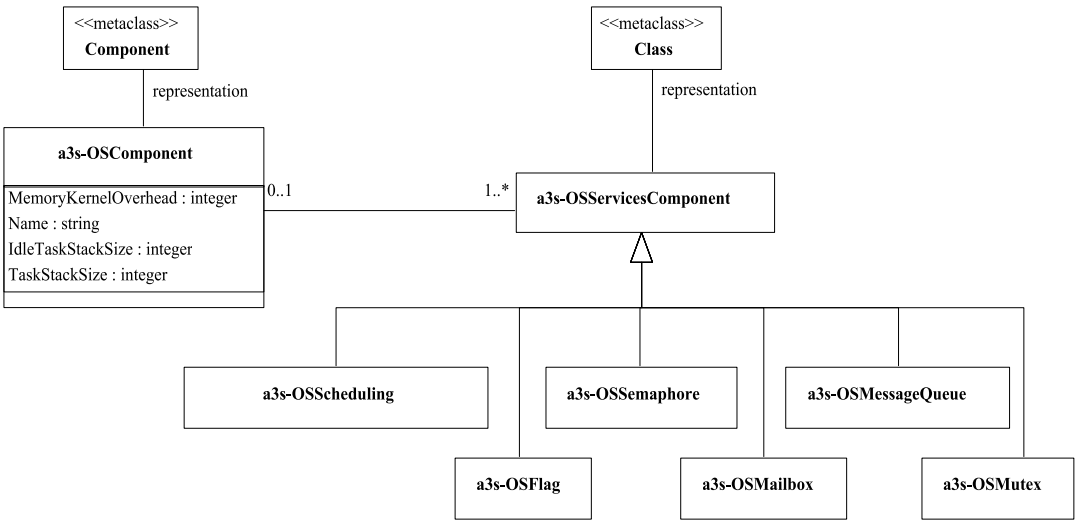


FIG. II.19 – Métamodèle du système d'exploitation

Ces composants sont donc regroupés au sein de deux librairies. Une librairie regroupe les composants ayant trait aux systèmes d'exploitation, et une autre ceux relatifs aux services possibles.
La librairie de composants de système d'exploitation est constituée uniquement d'éléments stéréotypés «*a3s-OSComponent*». Ils permettent de représenter les différents systèmes d'exploitation. Ces éléments ont des attributs spécifiques communs à la plupart des systèmes d'exploitation.

- *Name* : nom du système d'exploitation
- *Idletaskstacksize* : taille de la profondeur de la pile allouée à la tâche de fond, exprimée en bits.
- *Taskstacksize* : valeur de la profondeur des piles allouées aux tâches, exprimée en bits.
- *MemoryOverhead* : coût de la taille mémoire nécessaire utilisée par l'ajout de l'OS, exprimée en kbits.

La librairie des composants représentant les services comprend les services stéréotypés en fonction de leur nature à savoir :

- OSScheduling : définit l'ordonnancement de l'application.
  Il possède des attributs permettant de caractériser ce service :
  - *TickFrequency* : entier correspondant à la fréquence des ticks d'horloge.
  - *SchedulingPolicy* : type de politique d'ordonnancement suivi pour l'application (Round Robin, FIFO, RMS, à priorité etc.).
  - *IsPreemptive* : booléen qui détermine si les tâches peuvent être ou non préemtées.
  - *PriorityLevel* : entier qui détermine le nombre de niveau de priorité potentiel pour les tâches.
  - *PriotityType* : type de priorité suivie par les tâches (dépend des mécanismes de priorité des OS, ce peut être des priorités en tant que valeur (un chiffre = une priorité) ou en tant que nature (priorité suivant événement interruption matérielle ((HW, SW comme pour DSPBios) ou néante s'il l'ordonnancement est un round robin).
  - *PriorityMechanism* : type de mécanisme disponible pour éliminer les deadlocks (héritage de priorité, priorité à seuil, changement de priorité, ou néant).
- OSFlag : définit l'utilisation du service de Flag.
  - *ne possède aucun attribut.*
- OSMailbox : définit l'utilisation du service de Mailbox.
  - *ne possède aucun attribut.*
- OSMessagequeue : définit l'utilisation du service de Message Queue.
  - *ne possède aucun attribut.*
- OSMutex : définit l'utilisation du service de Mutex.
  - *ne possède aucun attribut.*
- OSSemaphore : définit l'utilisation du service de Sémaphore.
  - *ne possède aucun attribut.*

II.2.2 Métamodèle

Quatres *package* ont été définis dans le métamodèle du projet A3S. Ils sont insuffisants dès lors que l'on intègre la notion de système d'exploitation. En effet, l'ajout de deux librairies supplémentaires sont nécessaires. L'une stéréotypée «*OSComponentLib*», contenant les composants systèmes d'exploitation, et l'autre stéréotypée «*OSServicesComponentLib*» contenant les différents services pouvant être implantés et utilisés par les différentes tâches de l'application

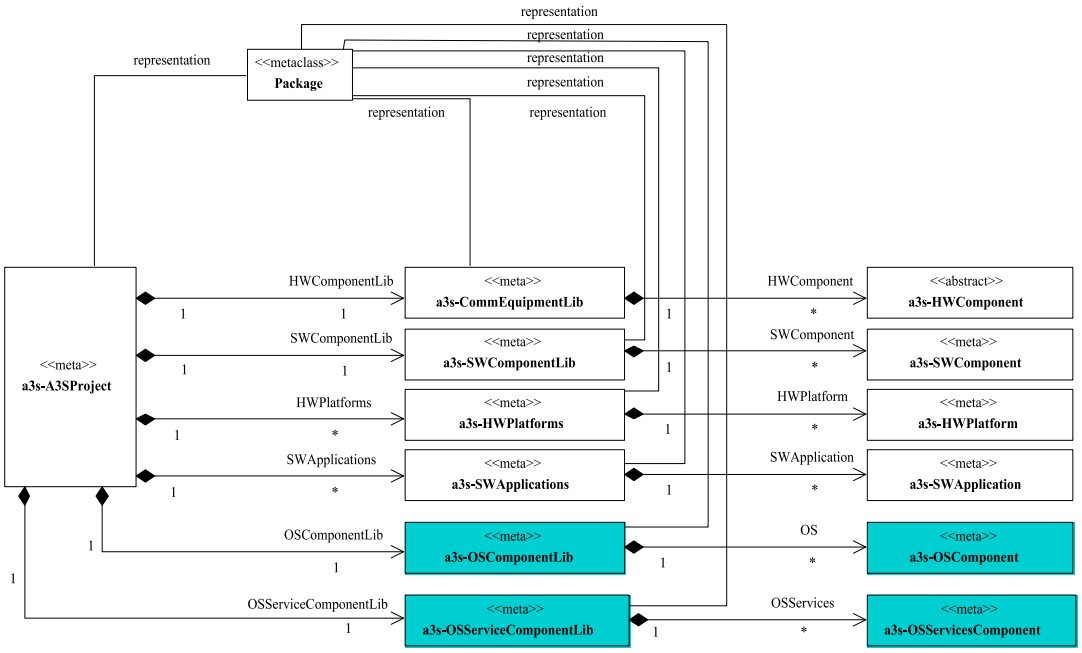


FIG. II.20 – Métamodèle du Projet A3S avec OS

logicielle. Ces deux librairies contiennent les composants que le concepteur peut utiliser pour concevoir un système doté d'un ou plusieurs système(s) d'exploitation. La prise en compte de l'OS se traduit donc par la modification du métamodèle du projet A3S comme le montre la figure II.20. La formalisation des composants et leur utilisation nécessitent respectivement le développement de deux métamodèles supplémentaires et la modification du métamodèle statique de déploiement figure II.14. Les deux premiers métamodèles, illustrés figure II.19 et II.15, formalisent les stéréotypes des "*Class*" et "*Component*" énumérés au paragraphe précédent et précisent l'appartenance d'un ou plusieurs service(s) à un OS. Ils établissent également les règles :

- ▷ qu'un service ne peut pas appartenir à plusieurs systèmes d'exploitation différents.
- ▷ qu'un service peut exister sans OS, précisément dans le cas où il est implanté en matériel.

La modification du métamodèle statique de déploiement formalise les règles d'implantation du système d'exploitation et de ses services. Il traduit les considérations énumérées au paragraphe précédent à savoir qu'un service appartient, et donc, se déploie sur une instance d'un système d'exploitation ou se réalise en matériel par instanciation sur un FPGA.

Ces métamodèles formalisent et offrent donc les moyens de représenter n'importe quel système d'exploitation en fonction de ses caractéristiques et des services qu'il propose. Ils permettent également la représentation des services de systèmes d'exploitation qui seraient implantés en matériel sans OS.

II.2.3 Intégration

Pour modéliser son système, le concepteur commence par définir sa plate-forme matérielle en créant un diagramme de déploiement et en venant instancier les composants matériels dont il a besoin, issus de la bibliothèque de composants matériels. Ensuite il fait de même en ce qui concerne son application en créant un diagramme d'activité à l'aide des composants issus de la

bibliothèque de composants logiciels. Lorsqu'il effectue ses choix d'implantation, le concepteur décide de l'implantation des fonctions (composants logiciels) soit en matériel, soit en logiciel. Il décide de leur implantation sur les différents composants matériels de la plate-forme. A ce stade le concepteur peut alors faire le choix d'utiliser un ou plusieurs système(s) d'exploitation qu'il vient instancier sur le(s) processeur(s) dont il dispose. Il peut alors choisir d'instancier un certain nombre de services sur les systèmes d'exploitation instanciés. Une fois ces services instanciés, il spécifie alors quelles sont les tâches qui utilisent ces services. Pour utiliser les services, les tâches sont reliées à ceux-ci en fonction de leurs besoins. Par exemple si une tâche désire se synchroniser avec une autre à l'aide d'un sémaphore, il faut instancier un sémaphore et connecter celui-ci aux tâches qui l'utilisent.

Un exemple d'intégration est illustré sur la figure II.21. Dans cet exemple, une application composée de deux tâches T1 et T2, utilise le service de sémaphore de μCOS. Le système d'exploitation ainsi que les deux tâches sont implantés sur un Power PC 405, intégré dans un FPGA Virtex II présent sur la carte ML310.

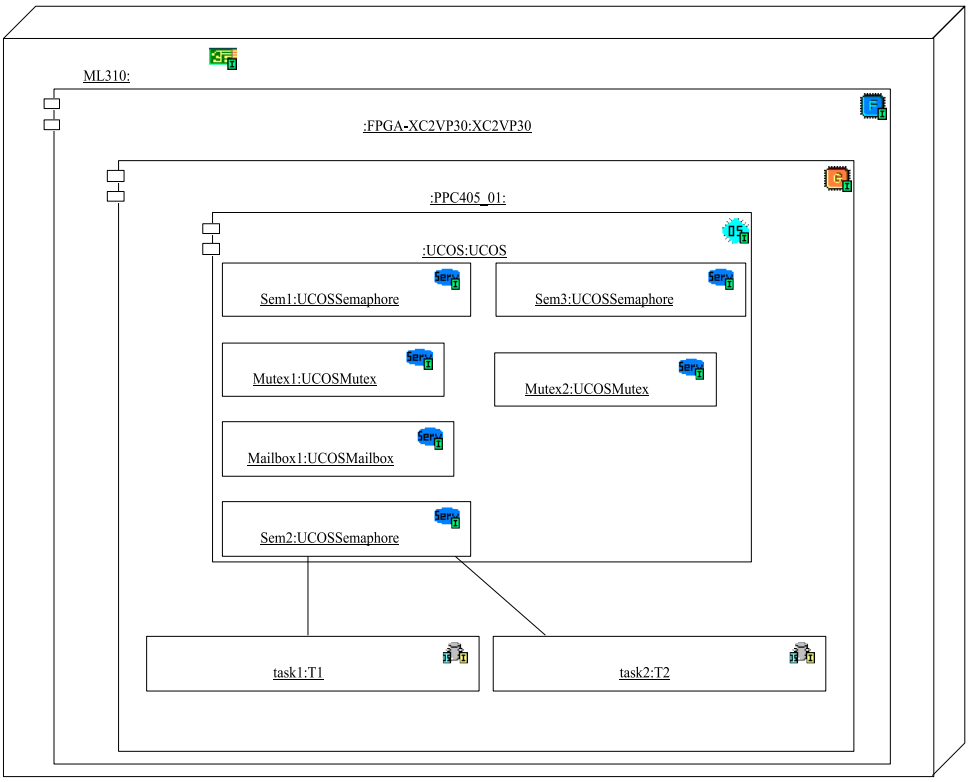


FIG. II.21 – Exemple d'utilisation des services de UCOS

Si tous les paramètres sont correctement renseignés l'*overhead* temporel dû au système d'exploitation peut être déterminé à partir des modèles d'exécution disponibles. En effet, à partir des choix d'implantation précisés sur le diagramme de déploiement, les tâches qui utilisent des services sont connues. Le nombre et la nature des services utilisés par l'OS, sont également connus. L'ensemble de ces informations va permettre le calcul de l'*overhead* dû à l'utilisation du système d'exploitation et de ses services (par application des lois d'estimation déterminées au IV.1.3), et de l'intégrer à l'étape du calcul de l'ordonnançabilité.

II.2.4 Vérifications et règles associées

La modélisation des OS dans notre méthodologie n'apparaît qu'à l'étape du *mapping* de l'application sur la plate-forme matérielle. Comme précédemment pour les autres modélisations UML, il est important de vérifier l'exactitude du déploiement effectué avec l'ajout du métamodèle des systèmes d'exploitation. Les vérifications vont donc porter sur la cohérence du mapping effectué (l'implantation des composants, leurs relations) d'après les règles d'utilisation traduites par le métamodèle statique de déploiement. L'utilisation du système d'exploitation par des tâches, est conditionnée par l'implantation de l'OS sur une cible matérielle de type processeur. Une vérification vérifie donc que toutes les instances "*ComponentInstance*" stéréotypées «*a3s-OSComponentInstance*» sont présentes sur des "*ComponentInstance*" héritant du stéréotype «*a3s-ProcessorInstance*». De plus, il faut également vérifier que les attributs ont été renseignés par des valeurs. Sinon les calculs de faisabilité sont impossibles. Ces vérifications sont effectuées au sein de l'outil Objecteering par des règles similaires à celles présentées au II.1.7 (vérifications des stéréotypes, renseignement des valeurs des paramètres).

En ce qui concerne les différents services des systèmes d'exploitation, le métamodèle concernant leur implantation (II.14) spécifie qu'ils ne peuvent être implantés sur un processeur sans qu'il y ait un OS instancié sur ce processeur. Cependant, ces services peuvent être réalisés de manière matérielle sur FPGA. Les vérifications portent donc sur le respect de ces conditions, et par la vérification que le service implanté en matériel dispose d'un lien de communication avec le service d'ordonnancement du système d'exploitation, implanté lui sur un processeur. Si ce lien est inexistant (absence de lien ou absence de service d'ordonnancement) la modélisation comporte des erreurs. Les vérifications de faisabilité et notamment l'analyse d'ordonnançabilité utilise les paramètres des différents composants suivant les lois d'estimation définies au IV.1.3. Cependant certains paramètres des lois d'estimation ne possèdent pas d'équivalents directs dans les paramètres des composants OS et des services. Ces paramètres sont en effet issus de l'analyse du modèle PSM traduit par le diagramme de déploiement après mapping. Cette analyse s'effectue par l'outil XAPAT, présenté dans la prochaine section. Elle permet d'identifier tous les paramètres nécessaires au calcul des lois d'estimation, notamment : le nombre de chacun des types de services utilisés, le type et le nombre de mécanismes sous-jacent mis en œuvre, le nombre de tâches utilisant un système d'exploitation, les tâches partageant le même sémaphore, etc.. Cette identification s'effectue grâce au stéréotype spécifique à chaque composant, et aux types de relations entre eux. Ces vérifications sont donc effectuées au sein de l'outil XAPAT, pendant le traitement du fichier XMI. Les mécanismes de vérification mis en œuvre reposent sur des séquences de recherche et de tests similaires à celles présentées dans l'exemple de la figure II.18.

II.3 Outillage

L'outillage utilisé pour intégrer le profil A3S en tant que flot de conception des SoC est résumé sur la figure II.22. Elle présente les différents échanges entre les outils (Objecteering, XAPAT, RTDT) et rappelle le rôle mené par chacun d'eux. Des études fonctionnelles ont donné lieu au préalable aux IP qui fournissent les informations dont nous avons besoin à notre niveau de conception. L'outil Objecteering utilise ces informations aux étapes de modélisations des systèmes. Les résultats de ces modélisations et spécifications, une fois les choix de mapping effectués, sont exportés au format XMI. Ce format est exploité par l'outil XAPAT qui effectue les dernières vérifications, puis analyse le système afin de trier les informations et les mettre

en forme (fichier xml) pour l'outil de calcul d'ordonnancement. Il procède à une seconde étape d'analyse, mais cette fois-ci des résultats provenant de RTDT, qui lui permet de proposer un GANTT. Comme précisé au début de ce document I.2.4, l'outil réalisé ne permet pas la réalisation totale du flot de conception, mais il s'y intègre. Comme la figure le montre, il y a déjà eu des études en amont de notre flot, qui ont abouti à la génération et la documentation d'IP, utilisées dans notre flot. De même, nous n'adressons pas la génération de code permettant l'implantation ou la réalisation des systèmes modélisés sur cible. Pour cela, il faudrait, à partir des modèles réalisés et des résultats fournis, effectuer des transformations de modèles pour les adapter aux outils adéquats. Notre outil serait alors le point d'entrée d'autres outils. Les paragraphes suivants précisent les rôles exacts de chaque outil.

II.3.1 UML

L'atelier de modélisation UML utilisé dans le cadre de cette thèse est Objecteering. Il fournit l'infrastructure logicielle pour la modélisation UML qui supporte la démarche MDA. Cet outil a été pour nous l'élément majeur permettant d'exploiter le langage UML pour définir, maintenir et appliquer le processus de conception d'un SoC avec la méthodologie A3S. Le profil UML A3S développé pour les applications Radio Logicielle s'intègre spécifiquement à cet atelier. Néanmoins les métamodèles développés, qui sont le cœur du profil, ne sont que des diagrammes de classes, et de ce fait, peuvent être intégrés dans n'importe quel outil de modélisation UML, car ces diagrammes traduisent la sémantique des concepts à travers les stéréotypes définis. C'est donc au sein d'Objecteering que le concepteur modélise et spécifie les différents modèles de l'application, de l'architecture, et du système d'exploitation. Il modélise son système à partir des différents "*Package*" contenant les différents composants utiles pour les modélisations. Ces différents *package* sont représentés sur la figure II.23.

Il intègre certaines phases de vérifications en interne et permet l'exportation des modélisations dans un format permettant des étapes de vérifications et d'analyses supplémentaires. Les vérifications internes à Objecteering concernent les vérifications de cohérence des modèles spécifiés. Elles sont stockées dans un arbre (un" template" de génération du point de vue d'Objecteering). Cet arbre permet de définir l'ordre dans lequel les vérifications de contraintes seront effectuées. Les méthodes contenant le code des vérifications de contraintes sont "accrochées" aux branches, et aux feuilles de l'arbre. Une représentation d'une partie de l'arbre de vérification est illustrée sur la figure II.24

Le pseudo-code utilisé pour ces règles de vérifications de cohérence est le langage J [103]. C'est un langage proche du Java qui exploite le modèle UML par le biais de requêtes, de règles de validation, de génération de code et de transformation de modèles. L'exemple de code donné sur la figure II.25 concerne la vérification de l'implantation d'IP sur un FPGA. Il permet de vérifier que les ressources nécessaires à l'implantation des IP sur le même composant FPGA, ne dépassent pas les capacités d'implantation proposées par le FPGA, en terme de nombre de cellules logiques, blocs mémoires, et blocs multiplieurs disponibles sur le composant. Etant donné que le langage UML ne permet pas de faire des calculs de performances, il faut extraire les modélisations et les spécifications sous un format qui le permet.

II.3.2 Passerelle XML

L'outil Objecteering avec le profil A3S permet de réaliser l'ensemble des modélisations et spécifications au niveau système correspondant aux premières étapes du flot de conception d'un

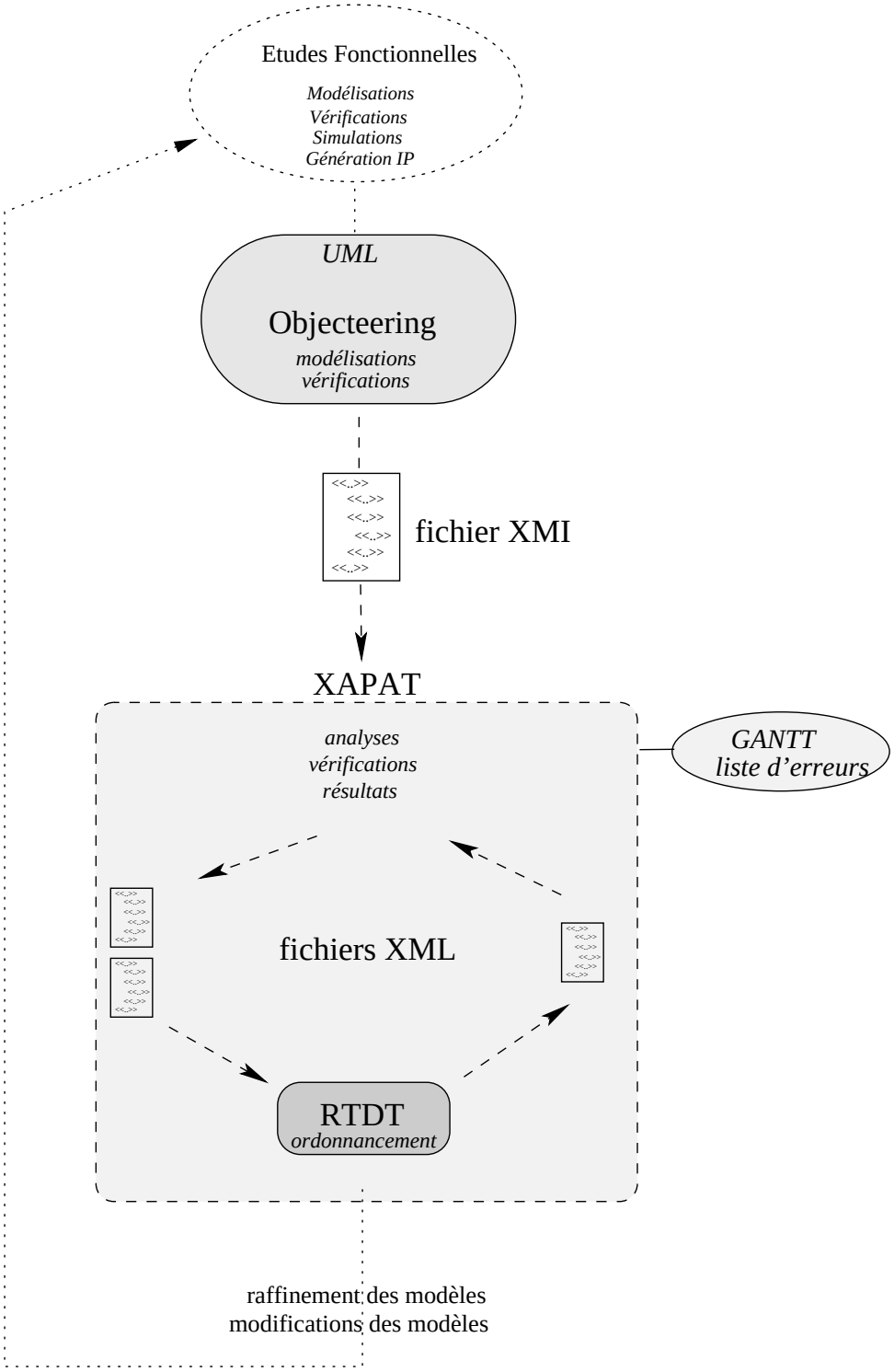
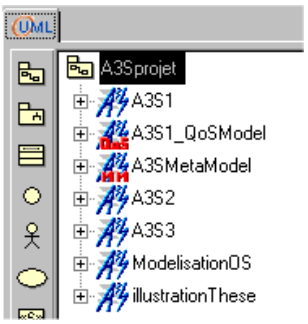
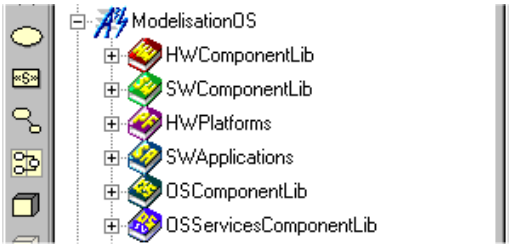


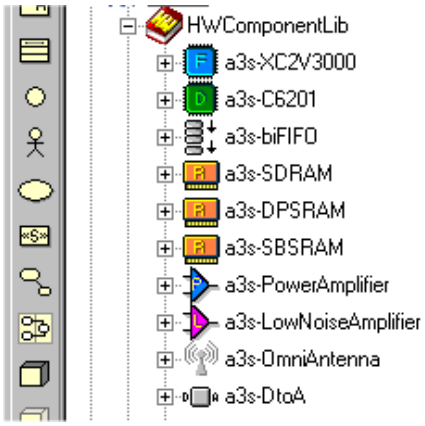
FIG. II.22 – Implication des outils dans le flot A3S



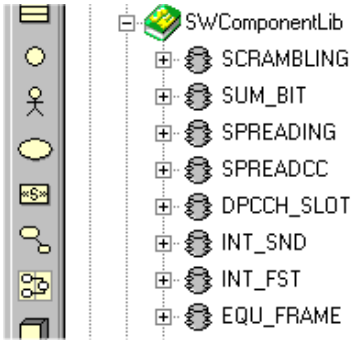
(a) Les projets



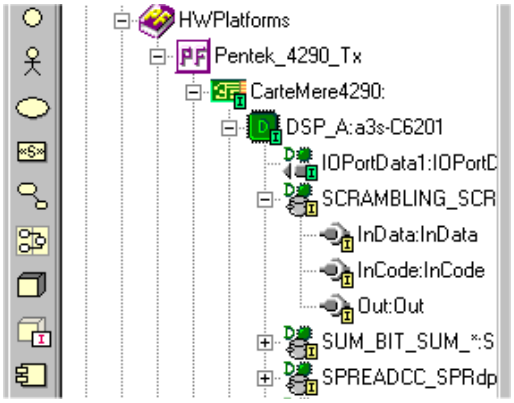
(b) un projet



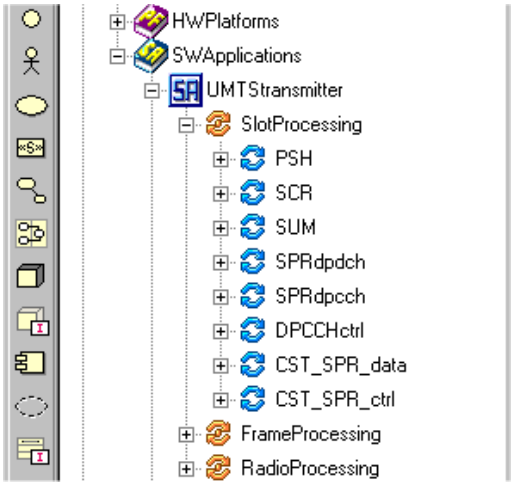
(c) Bibliothèque matérielle



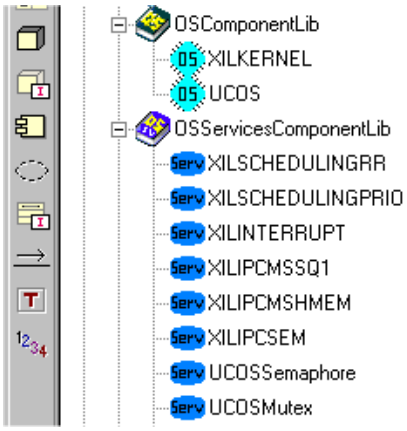
(d) Bibliothèque logicielle



(e) Modèle de plate-forme



(f) Modèle d'application



(g) Bibliothèque OS

FIG. II.23 – Les différents *package* du profil A3S intégrés à Objecteering

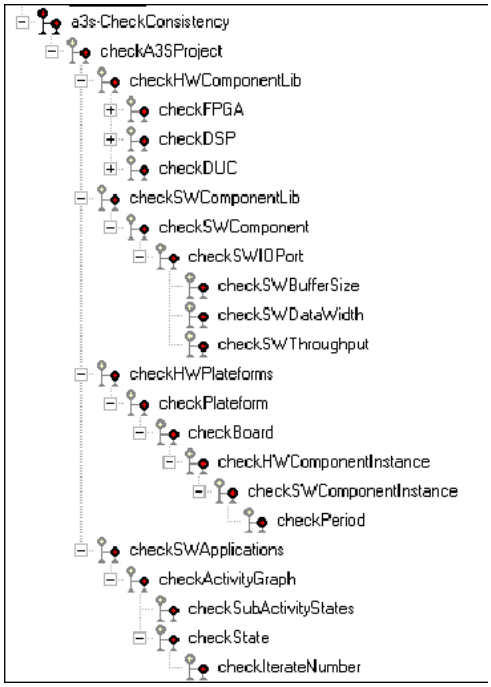


FIG. II.24 – Arbre des vérifications réalisées dans Objecteering

```
int ramBlock = BaseComponent.a3s_getTaggedValueContentInteger("a3s-RAMBlock");
int multBlock = BaseComponent.a3s_getTaggedValueContentInteger("a3s-MultiplierBlock");
int logicCell = BaseComponent.a3s_getTaggedValueContentInteger("a3s-LogicUnitCapacity");

if (ramBlock < RAM_BLOCK) {
currentJTreeItemCommEquipmentInstance.#Report#addReport("error", "RAM blocks of
FPGASWComponentInstances is greater than RAM blocks of " + Name, this);
}

if (multBlock < MULT_BLOCK) {
currentJTreeItemCommEquipmentInstance.#Report#addReport("error", "Multiplier blocks of
FPGASWComponentInstances is greater than multiplier blocks of " + Name, this);
}

if (logicCell < LOGIC_CELL) {
currentJTreeItemCommEquipmentInstance.#Report#addReport("error", "Logical cells of
FPGASWComponentInstances is greater than logical cells of " + Name, this);
}
```

FIG. II.25 – Exemple de vérifications codées en langage J

SoC. L'analyse de ce qui a été réalisé doit fournir au concepteur l'assurance que ses modélisations sont valides et respectent les contraintes qu'il impose. Le premier résultat est obtenu par un certain nombre de vérifications de cohérence pour chacune des modélisations et des choix d'implantation effectués. Le second, provient d'une analyse plus poussée, à partir d'un modèle d'exécution, mettant en œuvre des calculs d'ordonnançabilité. Un système peut être en effet réalisable, mais il peut néanmoins ne pas respecter des contraintes fixées par un cahier des charges (ex : contraintes temporelles, consommation). Pour parvenir à réaliser ces calculs et effectuer des vérifications plus approfondies, il faut passer par une transformation de modèles UML en modèles qui puissent être exploités. Une particularité d'Objecteering est qu'il peut exporter l'intégralité des modèles d'un projet au format XML (eXtensible Markup Language).

XML est une norme de langage de structuration de données, dérivée de SGML (Standard Generalized Markup Language) qui repose sur des balises (comme le HTML). La norme définit des éléments utilisés pour la structuration au sein de balises (*Tag*). XML, contrairement au HTML, ne dispose pas de balises définies. Chacun définit ses propres *Tag* suivant les informations qu'il doit structurer. De ce fait, pour personnaliser ses balises, il doit utiliser un espace de nommage (*namespace*) qui lui est propre. Objecteering formalise le fichier XML généré au moyen du *namespace* "UML :" présent dans toutes les balises, mais aussi en adoptant l'extension ".xmi" plutôt que ".xml" pour les fichiers générés.

Un fichier XMI est principalement composé de Tags contenant des éléments, des attributs, des namespaces, avec des valeurs (valeurs pour les attributs, valeurs pour les éléments). Il est hiérarchique (imbrication des balises) et donc les éléments peuvent contenir d'autres éléments. Chaque élément du métamodèle UML est retranscrit en XMI. Il existe donc une correspondance entre chaque élément stéréotypé du modèle, et sa traduction en XML dans un ensemble de *Tag*. Par identification, les différents répertoires créés dans l'arborescence de l'outil Objecteering se retrouvent en tant qu'éléments nommés '`Package`'. Les différents composants (matériels ou logiciels) manipulés par le concepteur lors de la modélisation de son système sont identifiés de part leurs natures et leurs stéréotypes. Dans l'exemple du fichier XMI du listing II.1, l'élément '`Package`' est contenu dans l'élément '`xmi`'. Le nom du package, "A3S2", est donné par l'élément '`ModelElement.name`'. L'imbrication est illustrée dans l'exemple par l'élément '`ModelElement.name`' imbriqué dans l'élément '`Package`' lui même imbriqué dans l'élément '`XMI`'.

La structure du document XML, reprend la structure des métamodèles. C'est une structure répétitive (ordre d'imbrication d'éléments du même nom). Le nombre d'éléments définis dans les métamodèles et utilisés dans les modélisations implique, dans le cas de la traduction des modélisations UML en XML, que nous obtenons des fichiers XMI très volumineux. En effet, dans le cas de l'application UMTS modélisée, le fichier XMI généré se compose de 16975 lignes de code. Ce chiffre important se justifie au regard de cette structure. A titre d'exemple, les composants matériels présents dans la bibliothèque des composants matériels sont traduits par des éléments '`Component`' dans le fichier XMI. Leurs noms sont exprimés comme valeur de l'élément '`ModelElement.name`' qui est encapsulé dans l'élément '`Component`' puisqu'il lui est propre. L'identification de ces composants (comme tous les autres), se fait par un attribut "xmi.id", dont la valeur est unique pour chaque élément. Leurs stéréotypes (dépendant de la nature du composant matériel FPGA, DSP, DUC etc.) sont exprimés sous forme d'éléments '`Stereotype`' et aussi encapsulés dans l'élément '`Package`' qui les contient. De même, toutes les caractéristiques (attributs) propres au composant matériel sont exprimées dans un élément '`ModelElement.taggedValue`'. Chacune est identifiée par le nom de l'attribut (par un '`TagDefinition.tagType`') et sa valeur dans un

'*UML :TaggedValue.dataValue*'. Les ports de ces composants matériels sont également exprimés en tant qu'élément '*Component*' et sont encapsulés dans le '*Component*'. Les autres éléments sont traduits suivant la même structure. La figure II.1 illustre encore une fois très bien ces imbrications.

L'extraction des informations du fichier XMI se fait par l'intermédiaire d'un parser XML qui extrait les informations essentielles aux vérifications et à l'analyse du système. Ce parser est intégré dans l'outil XAPAT (XMI A3S Parser Analyse Tool) développé en java et interfacé avec le profil A3S via une interface Web Service.

Listing II.1 – Exemple de Fichier XMI

```
<XMI xmi.version="1.1">
 <UML:Package xmi.id="a4263511320-1817">
  <UML:ModelElement.name>A3S2</UML : ModelElement.name>
  <UML:Namespace.ownedElement>
    <UML:Package xmi.id = 'a1112540288-8'>
      <UML:ModelElement.name>HWComponentLib</UML:ModelElement.name>
      <UML:Namespace.ownedElement>
    <UML:Component xmi.id = 'a1116995960-311'>
     <UML:ModelElement.name>a3s-XC2V3000</UML:ModelElement.name>
     <UML:ModelElement.stereotype>
      <UML:Stereotype xmi.idref = 'a4263511320-1165' /> <!-- a3s-FPGA -->
     </UML:ModelElement.stereotype>
     <UML:ModelElement.taggedValue>
      <UML:TaggedValue>
        <UML:TaggedValue.type>
        <UML:TagDefinition>
         <UML:TagDefinition.tagType>a3s-PowerConsumption</
             UML:TagDefinition.tagType>
        </UML:TagDefinition>
        <UML:TaggedValue.dataValue>100</UML:TaggedValue.dataValue>
       </UML:TaggedValue.type>
      </UML:TaggedValue>
</UML:Package></XMI>
```

II.3.3 XAPAT : Xmi A3S Profile Analysis Tool

Les applications modélisées font appel à des IP interconnectées dont la fonctionnalité ne peut être remise en cause. Le calcul des performances et la validité de l'application s'appuie donc sur les performances spécifiées des IP, combinées au graphe de tâches de l'application. Une partie des vérifications sont effectuées directement par Objecteering d'après les modélisations UML. L'autre partie des vérifications fait appel à XAPAT. L'outil XAPAT (Xmi A3S Profile Analysis Tool) a pour rôle de valider le système. Il regroupe et traite l'ensemble des informations contenues dans le fichier XMI. Le traitement de ces informations permet de faire les vérifications non permises par l'outil Objecteering. Il permet également de déterminer les différentes valeurs de périodes des composants logiciels qui n'auraient pas été renseignées par le concepteur. Cette fonctionnalité, mettant en œuvre les méthodes de dérivation de contraintes proposées par Ali Dasdan [104], sont intéressantes. Elle a été intégrée pour le cas où le concepteur veuille rajouter une fonction qu'il n'a pas encore estimée. La méthode calcule ou réutilise la période de chacune des tâches et le temps de séparation de chacune d'elle à partir des contraintes de l'application, de la valeur de la période de la première fonction de l'application et du graphe de tâches. Cependant, cette méthode se voit inhibée par l'introduction, en cours de développement, de valeurs par défaut pour les attributs de chaque élément. En effet, l'absence de valeurs par défaut augmente le taux d'erreur de conception si le concepteur omet de renseigner le paramètre. La validation déterminée par l'outil repose sur le calcul de l'ordonnancement des

différentes fonctions une fois les choix d'implantation déterminés. Il permet donc de détecter les éventuelles incohérences et mauvais choix d'adéquation du système modélisé.

L'outil est écrit en langage java et utilise un parser XML (xerces) pour extraire toutes les informations désirées. Il représente 5800 lignes de code. Il renvoie toutes les performances et erreurs de conception au concepteur. Pour cela, il traite le fichier XMI pour en extraire différents graphes à partir desquels il effectue les calculs (ordonnancement, performances). Tout d'abord il crée un graphe de tâches général pour chaque diagramme d'activité créé. Ce graphe est composé de noeuds et d'arcs. Chaque noeud représente un traitement (c.-à-d. un «*a3s-ObjectCode*» avec certaines valeurs d'attributs). Chaque arc représente le lien de connexion ("*transition*") entre deux fonctions s'échangeant des informations. La représentation de ce graphe en java, revient à créer, pour chaque «*a3s-ObjectCode*» du diagramme d'activité, un objet tâche qui comprend tous les attributs liés à son implantation (période, nombre de données échangées, etc.), ainsi que des attributs contenant la liste des tâches avec lesquelles elle échange des données. Cette étape permet d'obtenir les contraintes de précédence et le parallélisme possible.

L'autre format de graphe reprend la même approche mais il concerne l'architecture matérielle. Un graphe est créé pour chaque diagramme de déploiement. Ce graphe n'est composé que d'éléments matériels. La représentation de celui-ci en java revient à créer, pour chaque «*a3s-HWComponentInstance*», un objet composant. Cet objet a tous les attributs des composants matériels («*a3s-HWComponent*») avec en plus la liste des autres composants avec lesquels il est connecté. Il possède également la liste des tâches qui sont implantées sur lui. Il retranscrit l'ensemble des contraintes issues de la plate-forme matérielle et des choix d'implantation. Avec ce graphe, l'outil effectue les vérifications de cohérence non supportées par Objecteering, notamment celle concernant l'implantation choisie par le concepteur. De plus, la liste des éléments matériels avec leurs attributs permet de générer des fichiers pour un outil intégré à XAPAT nommé RTDT (Real-Time Design Trotter)[97]. Cet outil permet l'ordonnancement d'applications implantées sur des architectures monoprocesseurs avec des DSPs comme co-processeurs, et des FPGAs comme accélérateurs matériels. Cet outil a été adapté pour traiter les architectures multi-DSPs avec des FPGAs comme accélérateurs matériels. Il calcule et renvoie donc les informations relatives à l'ordonnancement et aux performances du système (taux d'utilisation, etc.). Cet outil nécessite deux fichiers d'entrée. Le premier fichier décrit l'architecture matérielle du système. Il s'agit juste d'isoler certaines informations du graphe (liste des composants et leurs caractéristiques) pour les structurer au bon format xml dans un fichier nomplate-formearch.xml Le second fichier comprend les informations sur l'implantation des tâches (localisation de chaque tâche avec les attributs liés à l'implantation). Elles aussi sont présentes dans le graphe et nécessitent d'être structurées au bon format xml dans un fichier nomapplicationcde.xml. Deux exemples extraits des fichiers nomplate-formearch.xml et nomapplicationcde.xml sont donnés sur les figures II.26 et II.27.

La structure du fichier de la figure II.26, composée d'éléments XML dotés d'attributs est hiérarchique. Cette particularité est exploitée par la traduction de la représentation de l'application logicielle du diagramme d'activité. Dans cet exemple, l'application UMTSreceiver est composée de traitements (ici sous l'élément task) qui échangent des informations via des '`connection`' qui leurs sont propres. La nature de leur implantation est spécifiée sous la forme d'un élément '`implementation`'. Tous les paramètres associés, ainsi que leurs valeurs se retrouvent exposés sous formes d'attributs. Il en va de même pour le fichier Pentek4290Rxarch.xml de la figure II.27 où la plate-forme contient des composants logiciels avec des ports qui sont connectés. L'analyse de ces deux fichiers permet de récupérer la valeur des différents paramètres, de traduire les contraintes d'implantation et les contraintes d'exécution

```
<ApplicationSpec>
  <Application name="UMTSreceiver">
    <task name="RAK_MM" id_ref="a1831863648-250" task_type="inter" periodicity="true" remanencecost="0"
      elapsetime="" size_code="0" task_com="false">
      <connection type="in" id-ref="a1831863648-304" nbtokensend="5120.0" nbtokenreceived="5120.0"
       iterate_number="15.0" data_size="32" data_ref="a1831863648-250-a1831863648-304" />
      <connection type="out" id-ref="a1831863648-277" nbtokensend="2560.0" nbtokenreceived="2560.0"
       iterate_number="15.0" data_size="32" data_ref="a1831863648-250-a1831863648-277" />
      <implementation component="DSP_A" period="667.0" areacost="" pwncost="10" executioncycl=""
       executiontime="37" initializationpart="false" />
    </task>
```

FIG. II.26 – Fichier UMTSreceivercde.xml

des traitements, afin d'effectuer les calculs d'ordonnancement.

```
<HardwareSpec>
  <Board name="hwplatform1">
    <component name="fpga1" id="a1831863524-229" type="a3s-FPGA" clock="420" Vdd="1.8" cycleswitch="200"
     poweroff="0.09" pwnIdle="0.2" pwnswitch="0.2">
      <connection name="fgpa1port1" type="input/output" clock="" fifonumber="1" fromto="dsp1" accesstime="5">
        <fifocross fifoid="a1831863524-1698" name="fifo" />
      </connection>
      <connection name="fgpa1port1" type="input/output" clock="" fifonumber="1" fromto="dsp1" accesstime="5">
        <fifocross fifoid="a1831863524-1698" name="fifo" />
      </connection>
```

FIG. II.27 – Fichier Pentek4290Rxarch.xml

II.4 Conclusion

Nous avons présenté dans ce chapitre, les résultats des réflexions qui ont justifié les choix retenus, concernant les éléments utilisés dans notre profil. La première réflexion portait sur l'identification des éléments (matériels et logiciels) utiles à nos représentations de systèmes Radio Logicielle. Cette identification n'a pas été anodine, car elle a été conditionnée par le niveau d'abstraction (niveau système) choisi pour représenter ces systèmes. Ensuite, une seconde réflexion s'est intéressée à définir quels étaient les éléments UML et les diagrammes que ce langage propose, les mieux adaptés pour représenter les éléments identifiés ainsi que nos applications et nos architectures. Dès lors les métamodèles qui composent le profil ont été créés afin de définir la sémantique adaptée à nos modélisations et les différentes relations entre les éléments qui définissent déjà des règles d'utilisation. Une troisième réflexion concernant l'établissement de règles de vérification a été menée. La totalité des règles de vérification intégrées à l'outil Objecteering sont présentes dans le listing A.1 en annexe. Le concepteur dispose d'un support de modélisation qui doit lui assurer que ses modélisations sont correctes, s'il veut que son système fonctionne et s'il veut obtenir des résultats de faisabilité et de performances temporelles. Toutes ces réflexions se sont concrétisées et ont été appliquées au travers des outils. Notre profil UML s'est intégré à l'outil Objecteering, support des modélisations UML. Les vérifications de cohérences ont été réalisées en fonction du potentiel des outils. Une part des vérifications est supportée dans Objecteering et l'autre part l'est dans l'outil XAPAT, spécialement développé pour analyser les modélisations créées et rendre un résultat de faisabilité à travers des calculs

de performances (avec l'aide de l'outil RTDT intégré). Le profil développé et son intégration dans Objecteering, associés à l'outil d'analyse, permet donc aux concepteurs de système Radio Logicielle, de concevoir et tester ceux-ci au niveau système.

Chapitre III

Une application Radio Logicielle : UMTS

Notre méthodologie de conception de SoC basée sur le langage UML, explicitée dans le chapitre précédent propose au concepteur de système Radio Logicielle un ensemble d'éléments pour le concevoir à haut-niveau d'abstraction. Ce chapitre vient valider la méthodologie, en présentant les résultats d'expérimentation de la conception d'une application radio logicielle UMTS avec le profil UML A3S développé. L'application UMTS à concevoir est dans un premier temps présentée avec ses contraintes associées. Puis, les différentes étapes du flot de conception d'un tel système avec notre profil sont explicitées. Les résultats d'analyses obtenus viennent enfin renseigner la pertinence des choix de conception effectués par le concepteur.

III.1 Application UMTS

III.1.1 Emetteur et Récepteur UMTS

L'expérimentation choisie pour tester et valider la méthodologie développée au travers du profil UML A3S, est une application UMTS. Elle est considérée comme Radio Logicielle car elle fait partie des applications de téléphonie mobile de $3^{ème}$ génération et possède une partie front-end dont la fréquence d'échantillonnage du convertisseur analogique /numérique est paramétrable logiciellement. Elle comprend toutes les caractéristiques des applications Radio Logicielle. C'est une application complexe avec des contraintes temps réel fortes imposées par le standard. Il s'agit d'une application type pour confronter la méthode et le profil développé aux besoins d'un concepteur dans le cadre du processus de conception à haut-niveau d'abstraction d'un SoC, appliqué aux systèmes Radio Logicielle.

La différence avec le GSM porte sur le réseau d'accès radio. En effet, le GSM fait passer les données par une cellule (antenne) qui émet et reçoit les informations dans une même bande de fréquence divisée en créneaux temporels (TDMA) pour chaque utilisateur. Contrairement à l'UMTS qui utilise le W-CDMA, ce qui permet d'envoyer simultanément toutes les données, par paquets et dans le désordre (sur n'importe quelle fréquence). C'est ensuite au récepteur de tout remettre dans le bon ordre.

Afin de préciser davantage l'application, les bandes de fréquences allouées pour l'UMTS s'étendent de 1900 à 2025 MHz et de 2110 à 2200 MHz. D'après la norme [105], dans ces deux bandes de fréquence, le mode d'accès radio terrestre (UTRA) offre 2 possibilités d'accès radio. Ces deux possibilités sont basées sur la technique de multiplexage WCDMA et sont le Frequency Division Duplex (FDD), et le Time Division Duplex (TDD), qui utilisent chacun des canaux de 5MHz de bande pour l'émission et la réception. Avec le FDD, les utilisateurs

se partagent des fréquences différentes en même temps sur les liaisons montantes et descendantes, alors qu'avec le TDD, les utilisateurs se partagent du temps sur la même fréquence. L'application expérimentée dans ce mémoire utilise l'accès FDD en liaison montante. C'est une configuration, qui contrairement à l'accès TDD, est adaptée à des tailles de cellules importantes et qui nous place ainsi dans le cas d'une communication orientée d'un mobile vers une station de base (encore appelée *node B*). Les techniques d'accès TDMA et W-CDMA sont illustrées sur la figure III.1, où chaque couleur (ou nuance de gris) correspond à un utilisateur différent.

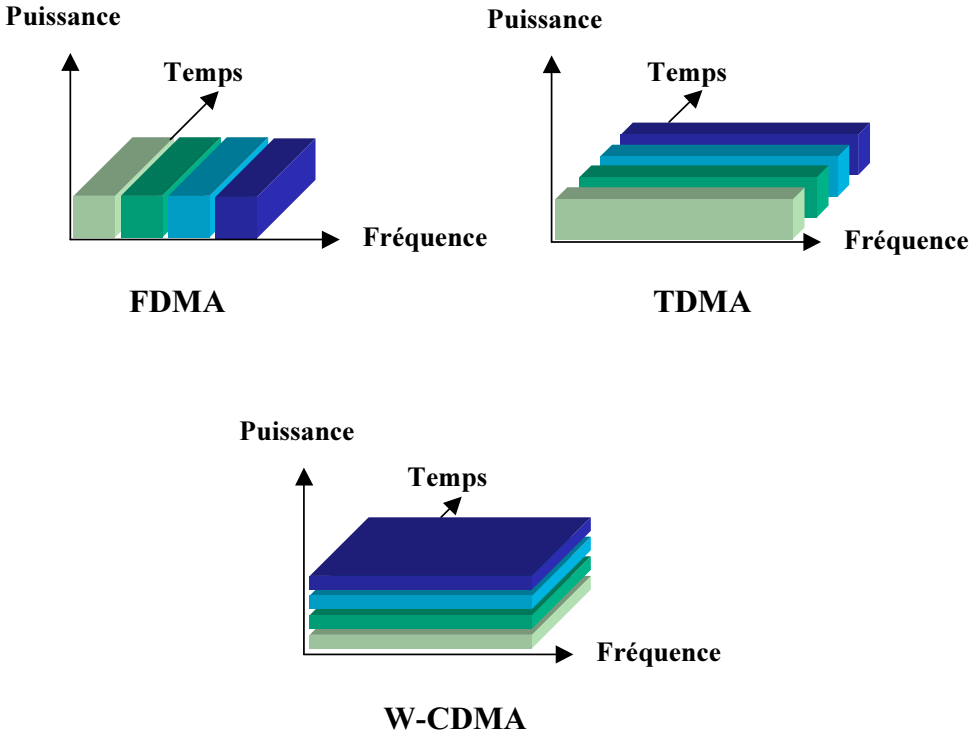


FIG. III.1 – Illustration des techniques d'accès

L'application UMTS utilisée est divisée en deux parties distinctes implantant les couches 1 et 2 du modèle OSI : l'émetteur et le récepteur. C'est une application limitée, puisqu'elle ne représente qu'un sous-ensemble du protocole radio UMTS utilisé entre le terminal mobile (téléphone) et le *node B*. Cette limitation s'explique par la complexité de la norme et nos besoins plus modestes. Elle a donc été dimensionnée pour permettre la confrontation du modèle et de la méthode développée, à un exemple réel de conception d'application Radio Logicielle. Seuls deux canaux physiques utiles au transport des informations de contrôle et de données entre le mobile et le réseau ont été développés (DPDCH, DPCCH) sans signalisation.

Afin de valider et évaluer les performances et les limites de notre modèle et de notre méthode, cette application a été réalisée sur une plate-forme matérielle afin de recueillir ses performances sur une cible déterminée. Ainsi, la démarche expérimentale présentée ci-après revient à modéliser l'application réellement réalisée sur une architecture hétérogène existante, et à s'assurer que le profil développé possède tous les éléments permettant la spécification complète d'un tel système. Cette démarche permet également de vérifier et valider la méthode d'estimation de performance, après avoir effectué des choix d'implantation, en comparant les résultats réels obtenus sur plate-forme. Ainsi nous nous mettons à la place d'un concepteur, qui doit réaliser une application UMTS et qui dispose de cartes de prototypage du commerce et qui se demande si elles sont en adéquation avec le type d'application qu'il souhaite mettre en œuvre.

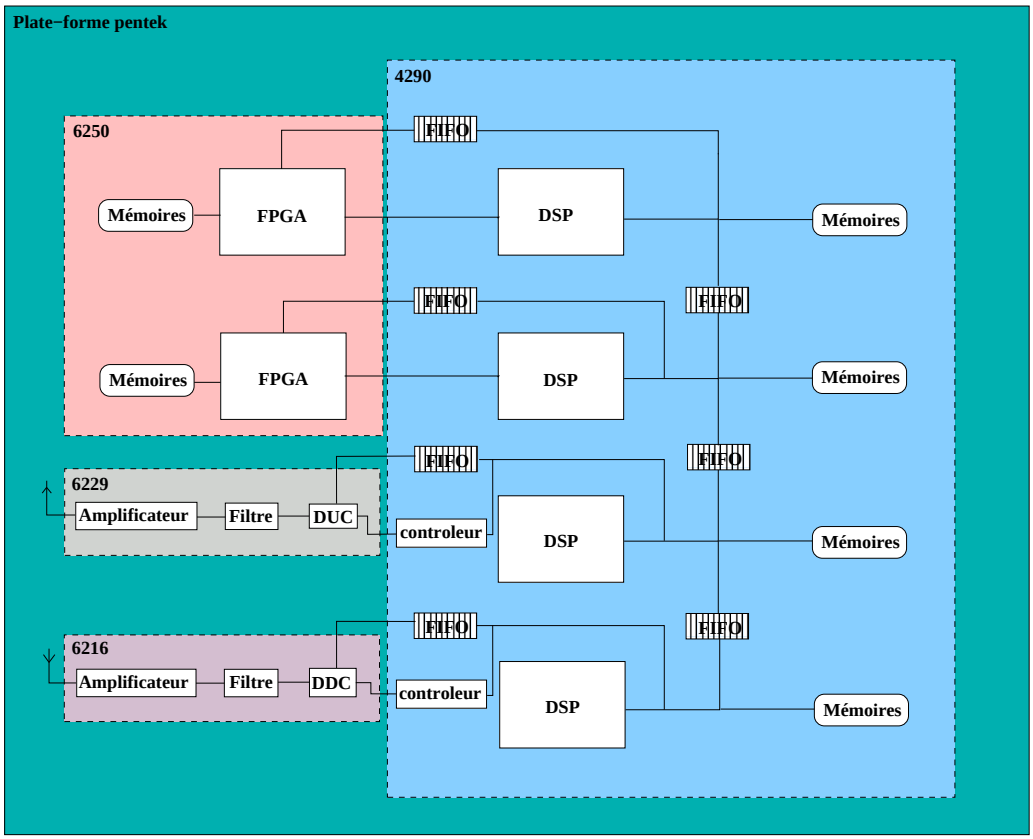


FIG. III.2 – Schéma de la plate-forme Pentek

Plate-forme matérielle

La plate-forme matérielle utilisée pour l'implantation de l'application UMTS est constituée d'un ensemble de cartes Pentek. Il s'agit d'une carte mère, modèle 4290, composée de 4 DSP C6X de chez Texas Instrument disposant de mémoires externes de nature différente (SBSRAM, SDRAM, DPSRAM), sur laquelle est interfacée une carte fille, modèle 6250, composée de 2 FPGAs Virtex II XC2V3000 de chez Xilinx. A cette même carte mère sont ajoutés deux autres modules pentek, d'émission 6229 et de réception 6216. Il s'agit donc d'une plate-forme hétérogène, flexible et reconfigurable, de part les multiples processeurs spécialisés (DSP) présents connectés aux ressources reconfigurables (FPGA). Les possibilités d'utilisation des ressources présentes sont importantes car chaque DSP peut s'adresser aux autres ainsi qu'à n'importe quel FPGA, même si pour se faire il doit passer pour 3 d'entre eux par l'intermédiaire d'un DSP différent à chaque fois. L'architecture de chaque modèle de carte pentek est présentée dans l'annexe B, et la plate-forme complète est schématisée sur la figure III.2.

Emetteur

Le schéma fonctionnel de l'émetteur, donné sur la figure III.3, présente les différentes fonctions mises en œuvre dans l'émetteur ainsi que leurs relations de dépendance. Le schéma représente l'interactivité des différents blocs fonctionnels pour la génération d'une trame de 10 ms. Une exécution de ce schéma permet d'obtenir une trame. Cependant une trame est obtenue par la répétition de l'exécution des parties grisées, c.-à-d. 4 exécutions des blocs fonctionnels du "transport bloc" et 15 itérations des blocs fonctionnels d'un slot dans la configuration d'un

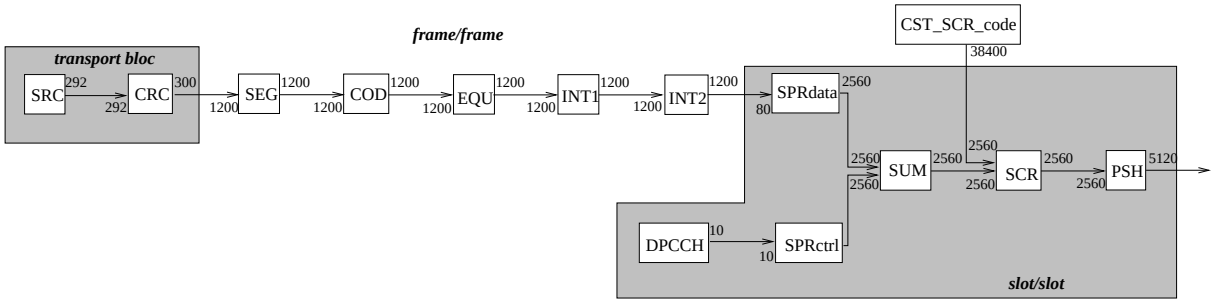


FIG. III.3 – Schéma fonctionnel de l'émetteur UMTS

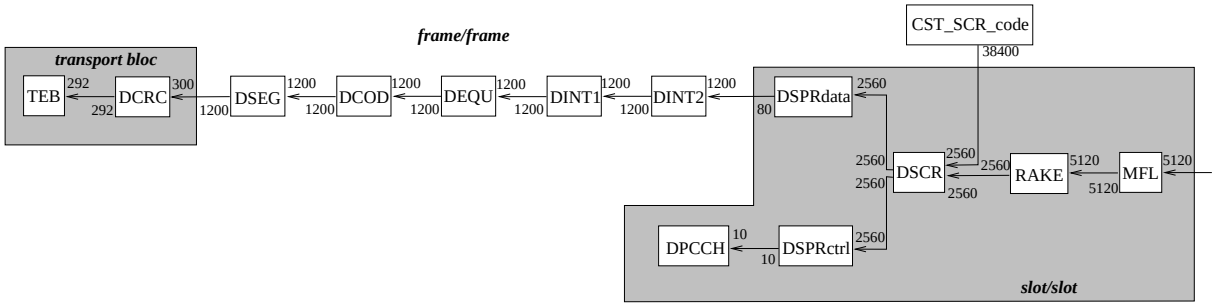


FIG. III.4 – Schéma fonctionnel du récepteur UMTS

débit de 117kbits/s. Les autres blocs fonctionnels ne sont exécutés qu'une seule fois par trame. La construction d'une trame débute donc par des données, indicées (SRC), complétées au sein de chaque transport bloc par un code de redondance cyclique (CRC) qui peut varier de 0 à 24 bits. Les blocs à émettre devant avoir une taille fixe, ils subissent donc une concaténation ou une segmentation (SEG) avant la phase du codage canal (COD) (soit par un code convolutionnel, soit par turbo code). Ils subissent ensuite un ajustement ou une égalisation (EQU) pour être adaptés à la taille des blocs fixée et donc respecter la capacité du canal physique. Le codage canal fait en effet varier le nombre de bits de chaque bloc de transport. Les entrelacements (INT1 et INT2) qui suivent l'adaptation de débit, ont pour effet de répartir aléatoirement les erreurs et donc d'augmenter les performances du CRC. L'étalement de spectre (SPR) qui sépare les différents canaux utilisés par un même utilisateur, peut enfin avoir lieu d'après les informations issues du canal physique (DPCCH). Une fois étalé, un code d'embrouillage (SCR) aléatoire est appliqué pour identifier les différents utilisateurs d'une même cellule. Un dernier filtrage (PSH) est ensuite exécuté avant la modélisation et l'émission de l'onde radio.

Les blocs fonctionnels (traitements) s'échangent des informations. L'échange de ces informations est représenté par les liens de transition. Ceux-ci possèdent les attributs permettant de spécifier le nombre de données échangées. Dans le cas présent ce chiffre correspond au nombre de chips échangés (émis et reçus). Un chips correspond au codage des bits d'information selon la relation : $SF = \frac{DebitChips}{Debitbit}$ soit $Debitbit * SF = 3.84 Mchips/s$, car le débit chips est fixé par la norme UMTS. Il est donc possible de jouer sur le débit bit (débit de l'information) en faisant varier le facteur d'étalement (SF).

Récepteur

De même que pour la partie émission, le schéma fonctionnel présenté figure III.4, identifie les différentes fonctions mises en œuvre dans le récepteur ainsi que leurs relations de dépen-

dance. Le schéma reflète les blocs fonctionnels inverses de ceux de l'émetteur. Une fois la démodulation de l'onde effectuée, un filtrage adapté (MFL) est appliqué. Une synchronisation (Rake) des trajets multiples dus aux rebonds des ondes radio est nécessaire. Elle est suivie par l'étape de désembrouillage (DSCR) (grâce aux codes de générateur d'embrouillage (CST_SRC_code)). Ensuite, le code du canal physique est désétallé (DSPR). Les étapes inverses de l'émission se poursuivent avec les deux désentrelacements, suivis du réajustement de la taille des paquets (DEQU), du décodage canal (DCOD). Le processus se poursuit avec la désegmentation de la séquence de génération des blocs de transport. Le bloc de transport est séparé en 4 blocs (DCRC), afin de récupérer les données transmises (TEB).

L'application est soumise à de fortes contraintes (logicielles et matérielle) imposées par la norme UMTS (une trame dure 10 ms) et la plate-forme (fréquence de fonctionnement maximum des composants) mais aussi par la configuration du fonctionnement qu'il est possible d'adopter (choix de l'implantation). La traduction de ces diverses contraintes s'effectue par la spécification du système via le renseignement des valeurs des attributs des composants (logiciels, matériels) utilisés dans les modélisations. Ces contraintes apparaissent donc en tant que valeurs des attributs des éléments UML et imposent certains choix dans le type d'éléments à utiliser.

III.2 Modélisation

Modélisation de la plate-forme matérielle

La plate-forme sur laquelle nous voulons estimer la faisabilité de l'implantation de l'application UMTS est connue, et nous disposons des data sheets de chaque carte, nous permettant ainsi d'identifier les composants matériels utilisés, leurs caractéristiques ainsi que leurs interconnexions. Avec le profil A3S intégré à Objecteering, la première étape, après avoir créé un nouveau projet intégrant le profil A3S, est de définir les composants qui vont constituer la librairie de composants matériels. Ces composants, une fois identifiés, vont être utilisés pour modéliser la plate-forme matérielle pentek. L'ensemble des cartes est constitué de 4 DSP TMS320C6201, de 4 FGPA XC2V3000 (1 par DSP), de mémoire type FIFO entre chaque DSP et FPGA, de mémoires SDRAM, de bus d'interconnexion, de convertisseurs, d'antennes, de multiplieurs de fréquence, etc.. Ce sont donc ces composants que l'on va retrouver dans la bibliothèque de composants matériels. Chaque composant a ses propres caractéristiques liées à sa fonctionnalité et aux paramètres qu'il est possible de modifier.

La liste des composants matériels ajoutés dans la librairie de composants matériels et utiles à la modélisation de la plate-forme matérielle Pentek compte une vingtaine de composants. Il s'agit :

- d'un DSP TMS320C6201 cadencé à 300 Mhz,
- d'un FPGA XC2V3000 cadencé à 420 Mhz,
- d'éléments mémoires de type bi-FIFO, SDRAM, DPSRAM, SBSRAM,
- d'un amplificateur de puissance,
- d'un amplificateur faible bruit,
- d'une antenne,
- d'un convertisseur numérique analogique,
- d'un convertisseur analogique numérique,
- d'un convertisseur numérique à fréquence ascendante (digital up converter),
- d'un convertisseur numérique à fréquence descendante (digital down converter),

- d'un filtre analogique,
- d'éléments d'interconnexion, bus, connexion multi-points, fil.

Une partie de ces composants sont présents sur la figure II.23(c). Chacun de leurs paramètres est renseigné par une valeur issue des caractéristiques du composant ou des contraintes imposées.

Pour exemple le DSP C6201 a pour valeur d'attributs :

a3s-PowerConsumption	=100 mw
a3s-MaxOperatingFrequency	=300 Mhz
a3s-VolatileMemoryCapacity	=256 kbits
a3s-ProgMemoryCapacity	=384 kbits
a3s-DataMemoryCapacity	=512 kbits
a3s-ComputationCoding	=fixed

Ses ports sont renseignés comme suit :

a3s-DataFlowDirection	=Input/Output
a3s-Throughput	=600000 bits/s
a3s-DataWidth	=32 bits
a3s-MemoryAccess	=CPU

Toutes les valeurs renseignées pour les attributs sont regroupées dans le délivrable A3S [96].

La plate-forme matérielle, modélisée avec le profil A3S, se présente donc sous la forme d'un diagramme de déploiement UML, visible la figure III.5. Cet extrait du diagramme est obtenu après avoir créé une nouvelle plate-forme matérielle et avoir instancié les composants présents dans la bibliothèque de composants matériels. La complexité de la plate-forme, le nombre d'éléments qu'elle comporte (89) et la résolution atteignable, ne permettent pas de voir l'intégralité de la plate-forme dans le détail. Seule la carte mère 4292 avec deux de ces DSP apparaît sur la figure. Les différentes cartes (4292, 6250, 6229, 6216) sont donc représentées par des éléments "*NodeInstance*" stéréotypés «*a3s-Board*». Chaque carte est modélisée le plus respectueusement possible à la réalité. Les composants qui les composent sont représentés par les instances des composants matériels appropriés dont les attributs sont déjà renseignés dans la bibliothèque.

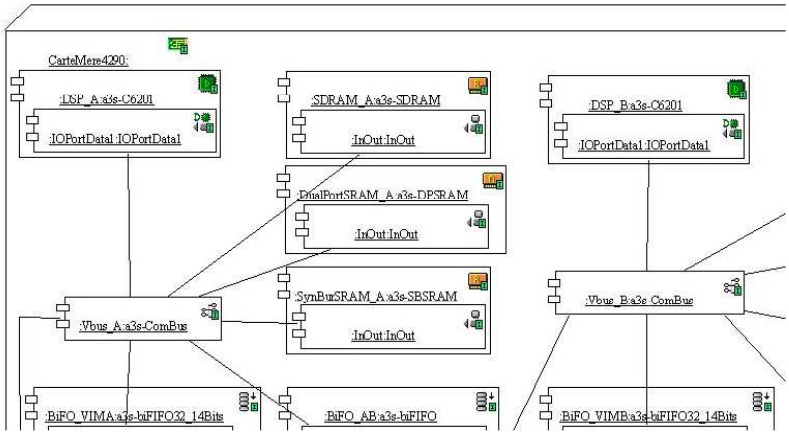


FIG. III.5 – Extrait du diagramme de déploiement de la plate-forme UMTS

Dans l'expérience, deux plates-formes matérielles très similaires ont été modélisées. La première modélise la plate-forme émettrice (Pentek_4290_Tx) et est constituée de la carte mère 4290, d'une carte fille 6250, et de la carte émettrice 6229. La seconde plate-forme concerne la partie réceptrice (Pentek_4290_Rx). Elle est également constituée de la carte mère 4290, d'une carte fille 6250 mais la carte émettrice est remplacée par la carte réceptrice 6216. Ainsi il est possible de vérifier si le profil supporte bien plusieurs modélisations de plates-formes matérielles et qu'il les dissocie parfaitement lors de l'étape de mapping où chaque application

(diagramme d'activité) s'implante sur une des plates-formes modélisées dans le projet créé.

Modélisation de l'application logicielle

L'application logicielle est découpée en deux sous-applications : l'application de l'émetteur UMTS d'une part, et l'application du récepteur UMTS d'autre part. Ces deux applications sont modélisées par deux diagrammes d'activités distincts dénommés respectivement UMTStransmitter et UMTSreceiver. Comme pour le cas des deux plates-formes modélisées, il est ainsi possible de vérifier que le profil supporte bien plusieurs modélisations d'applications différentes. Les deux schémas fonctionnels des deux applications, sont traduits à l'aide d'éléments "*ActionState*" et "*SubactivityState*" respectivement stéréotypés «*a3s-ObjectCode*» et «*a3s-Module*». Toutes les relations inter blocs fonctionnels (échange de chips) sont modélisées par des éléments "*Transition*" stéréotypés «*a3s-ObjectCodeTransition*». L'ensemble des caractéristiques de chaque bloc fonctionnel, indépendantes de leur implantation, est spécifié par l'affectation de valeurs aux attributs qu'ils possèdent. De même, le volume de données échangées sous forme de chips et leur directivité sont spécifiés par l'affectation de valeurs aux attributs de chaque "*Transition*". L'ensemble des attributs et des valeurs spécifiées pour chacun des 20 blocs utilisés par l'émetteur et des 18 blocs utilisés par le récepteur est donné dans le modèle de l'application système [106]. Pour illustrer voici les valeurs des attributs concernant le traitement PSH :

a3s-IterateNumber	=	15
a3s-IsPeriodic	=	true
a3s-RealTimeConstraint	=	667 μs
a3s-RunningMode	=	run
a3s-Delay	=	0

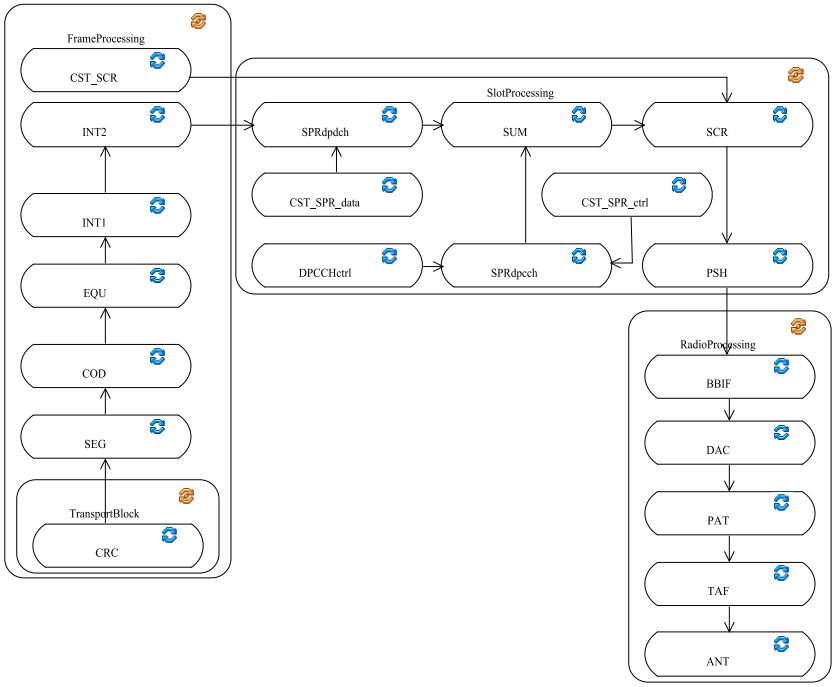


FIG. III.6 – Modélisation de l'application émetteur UMTS

Les choix d'implantation (mapping)

La modélisation des deux sous-applications sous forme de diagramme d'activité ne représente que le modèle PIM de l'application. L'avantage est que l'on peut l'implanter sur n'importe quelle plate-forme matérielle sans avoir à modéliser ni spécifier à nouveau l'application. Le passage au modèle PSM, qui traduit les choix d'implantation de chaque bloc fonctionnel, s'effectue en sélectionnant, dans une liste déroulante, une plate-forme matérielle déjà modélisée. A partir de cette sélection, chaque bloc fonctionnel, doit être associé à un composant logiciel (IP du bloc fonctionnel) issu de la bibliothèque de composants logiciels. Il est dans le même temps instancié sur une instance de composants matériels de la plate-forme sélectionnée, que le concepteur choisit. Ceci est représenté sur le schéma de la figure III.7. Dans l'exemple illustré, le concepteur veut implanter l'"*ObjectCode*" ObjC1 du diagramme d'activité sur le composant matériel DSP1. Ce composant matériel est utilisé dans la plate-forme matérielle modélisée par le diagramme de déploiement avant mapping. Il est modélisé par l'instance du DSP1 de la bibliothèque : DSP1i. L'implantation de ObjC1 sur DSP1i implique des contraintes d'implantation qui déterminent l'IP correspondante à utiliser pour cette implantation et ses caractéristiques. Cela se traduit par un lien entre ObjC1 et l'IP utilisée (SwC1) présente en bibliothèque pour sa mise en œuvre sur la plate-forme (SwC1_ObjC1 :SwC1i). Dans notre application la traduction de ce schéma serait, par exemple, l'implantation du traitement du code correcteur d'erreur (CRC) du diagramme d'activité de l'émetteur, sur une des instances de DSP de la plate-forme Pentek DSP_A (TMS320C6201) du diagramme de déploiement. Cette implantation se traduit par le lien entre, l'"*ObjectCode*" : CRC, et sa réalisation par un code convolutionnel issu de la bibliothèque de composant logiciel : CRC_END ("*Class*"). La réalisation par ce code convolutionnel implique son instanciation sous forme d'un "*Object*" CRC_END_CRC :CRC_END sur le DSP_A dans le diagramme de déploiement après mapping. L'avantage de cette méthode est que si les résultats d'estimation obtenus avec cette implantation ne sont pas satisfaisants, il est aisé de modifier l'algorithme implanté en choisissant par exemple un codeur de viterbi au lieu d'un code convolutionnel, ceci sans modifier le diagramme d'activité. La modification à apporter se situe sur le lien à effectuer entre le CRC et sa réalisation, donc sur le choix de l'IP dans la bibliothèque.

Pour reprendre l'exemple du traitement PSH, celui-ci est implanté sur l'instance d'un FPGA. Il hérite donc d'attributs supplémentaires en lien avec les contraintes de son implantation :

a3s-PriorityLevel	=	1	a3s-BoundedPeriod	=	false
a3s-ExecutionTime	=	441 μs	a3s-InitializationPart	=	false
a3s-PeriodMin	=	10 μs	a3s-PowerConsumption	=	10 mw
a3s-Period	=	667 μs	a3s-LogicalCell	=	14336
a3s-PeriodMax	=	10000 μs	a3s-MultiplierBlock	=	96
a3s-Code	=	interne	a3s-RAMBlock	=	96
a3s-Start	=	0	a3s-BitStream	=	40 kbytes
a3s-Deadline	=	0			

Plusieurs configurations ont été testées pour tester l'efficacité de notre méthode et de notre modèle. Les différentes configurations viennent modifier les contraintes d'implantation (choix du mapping) mais aussi les contraintes applicatives (débit souhaité). Comme nous l'avons dit précédemment, le débit bit est modifiable selon la valeur choisie pour le facteur d'étalement. Ainsi nous avons testé l'application suivant les quatre configurations résumées dans le tableau III.1.

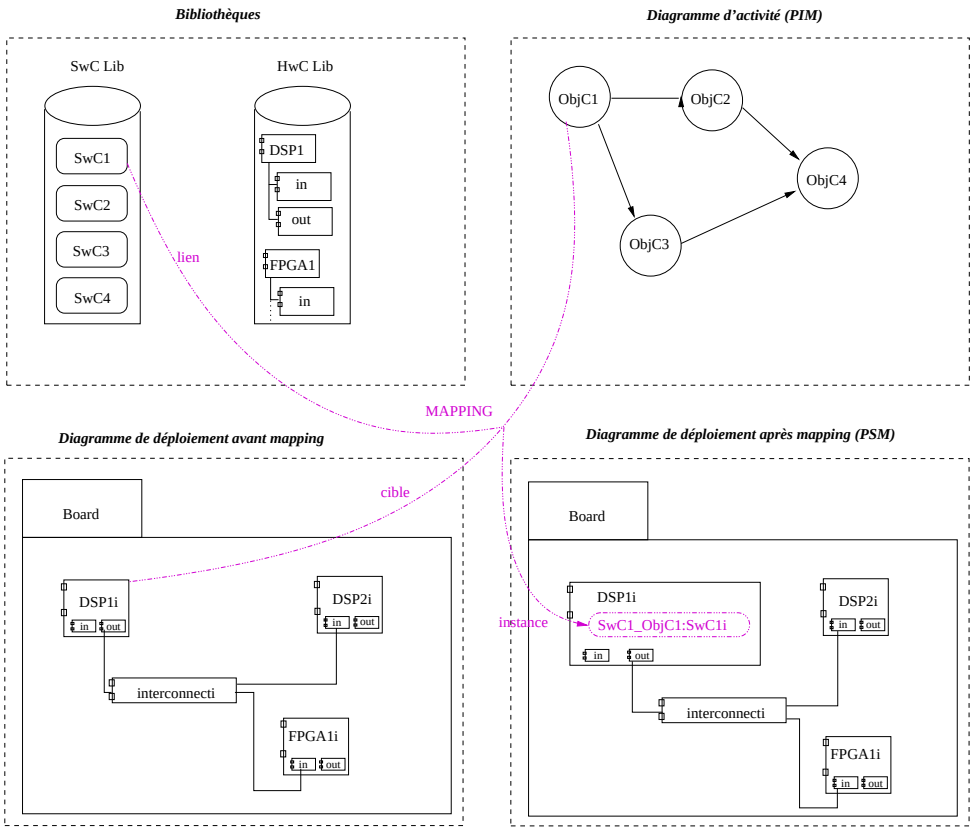


FIG. III.7 – Schéma du processus de mapping pour un ObjectCode

Débit / Application	117 Kbits	950 Kbits
émetteur et récepteur	tout logiciel sur DSP (1)	tout logiciel sur DSP (2)
émetteur et récepteur	logiciel et matériel (DSP+FPGA) (3)	logiciel et matériel (DSP+FPGA) (4)

TAB. III.1 – Les différentes configurations liées aux débits et à l'implantation choisis

(1) La première configuration est l'implantation sur les deux plates-formes, Pentek_4290_Tx et Pentek_4290_Rx, de toutes les fonctions en logiciel. Cette implantation s'effectue uniquement sur les DSPs pour les deux applications UMTStransmitter et UMTSreceiver. Les fonctions des deux applications sont réparties sur deux des DSPs (DSP_A et DSP_C) dans chacun des cas. La contrainte applicative correspond au débit bit à tenir et est fixé à 117 Kbit/s.

(2) La seconde configuration est identique à (1) sauf que la contrainte applicative passe de 117 Kbit/s à 950 Kbit/s. Ceci est réalisé par la modification du nombre d'itérations du transport bloc qui passe de 4 itérations à 10 puisque le facteur d'étalement passe de 32 à 4 pour obtenir le débit bit désiré.

(3) La troisième configuration conserve la contrainte applicative de (1), mais l'implantation matérielle des deux applications devient hétérogène. En effet, les deux fonctions les plus gourmandes en terme de ressources et de temps d'exécution ne sont plus implantées en logiciel mais en matériel (sur FPGA). Ainsi les fonctions PSH et MFL respectives des applications UMTStransmitter et UMTSreceiver, précédemment mises en œuvre sur le DSP_A sont mises en œuvre sur le FPGA_A de la plate-forme Pentek_4290_Tx et sur le FPGA_A de la plate-forme Pentek_4290_Rx. L'implantation des autres fonctions reste inchangée.

(4) La dernière configuration correspond aux contraintes d'implantation de (3), avec les contraintes applicatives de (2). Les choix d'implantation sont hétérogènes avec une contrainte applicative plus stricte (débit=950Kbits/s).

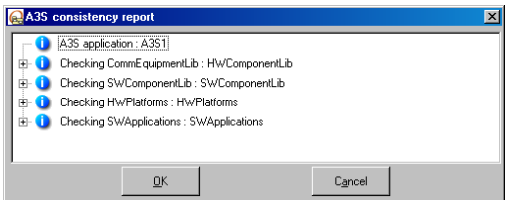
Ce sont donc ces différentes configurations qui vont venir éprouver le modèle et la méthodologie suivie pour l'analyse des modèles du système.

III.3 Résultats d'analyse

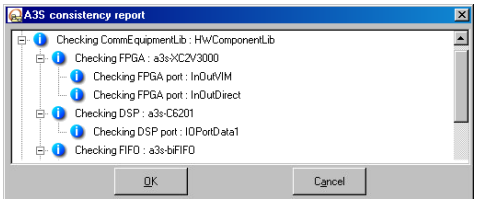
Différentes configurations donnant lieu à des systèmes UMTS différents, en terme de réalisation et de performances attendues, ont été effectuées afin d'obtenir des estimations de faisabilité et de performances pour les 4 possibilités de l'espace de conception présentées.

Des vérifications automatiques, développées dans le profil A3S, ont été lancées à chacune des étapes du flot de conception couvert par l'outil. La cohérence de chacune des modélisations a ainsi pu être établie, stipulant au concepteur qu'aucune erreur structurelle de modélisation et de spécification n'a été commise. Nous avons donc eu l'assurance que l'ensemble des représentations est conforme au profil et que l'intégralité des valeurs des attributs des éléments nécessaires à l'analyse ont été renseignées correctement (aucune valeur farfelue présente). Ces vérifications, présentées dans la section II.1.7, apparaissent sur les captures d'écran de la figure III.8.

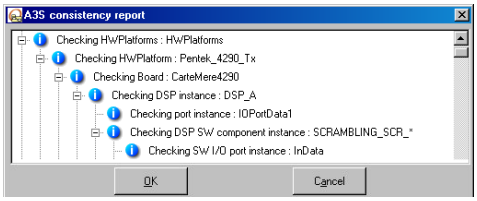
La traduction des contraintes d'implantation issues de la mise en œuvre des ObjectCode sur les instances des composants matériels, rentre en compte dans la phase d'analyse. En effet, les divers choix d'implantation effectués influent sur le calcul de l'ordonnancement des fonctions et des performances temporelles des temps d'exécution. En effet, dans le cas où toutes les fonctions sont implantées sur le même composant matériel, les temps de communication inter-fonction peuvent être considérés comme nul. Or dans le cas où deux fonctions qui interagissent entre-elles sont implantées sur deux composants matériels hétérogènes, les temps de communications ne sont plus considérés comme négligeables et un temps supplémentaire doit



(a) Vérifications globales



(b) Vérifications bibliothèque matérielle



(c) Vérifications plate-forme

FIG. III.8 – Extraits des vérifications effectuées

être ajouté aux temps d'exécution des fonctions. Ceci n'est possible qu'après analyse des choix d'implantation en étroite relation avec les graphes de tâches issus des diagrammes d'activité.

Les résultats des analyses sont issus de l'outil XAPAT qui intègre RTDT. Lorsque toutes les modélisations de l'application UMTS ont été effectuées, les vérifications dans l'outil Objecteering sont lancées. Ces vérifications ne faisant pas état d'erreur, le ficher XMI correspondant au système a été généré et envoyé dans l'outil XAPAT. XAPAT exécute alors l'analyse du fichier qui effectue les dernières vérifications et qui génère un graphe, exploité pour fournir à RTDT ce qu'il faut pour le calcul de l'ordonnancement (figure II.22). Cette étape automatique nous renvoie alors les résultats du tableau III.2. Ces résultats sont obtenus par le calcul de l'ordonnancement des blocs fonctionnels vu comme tâches indépendantes les unes des autres (algorithme du Rate Monotonic). Lorsqu'un ordonnancement est possible les résultats sont également renvoyés sous la forme d'un gantt.

	débit bit 117 kbits/s				débit bit 950 kbits/s			
	DSP_A	DSP_C	FPGA_A	temps (ms)	DSP_A	DSP_C	FPGA_A	temps (ms)
UMTStransmitter								
implantation tout logicielle	96,6 %	3,4 %		10	96,6 %	5,1 %		10,33
implantation logicielle/matérielle	11,4 %	3,4 %	66 %	7,96	11,4 %	5,1 %	66 %	8,29
UMTSreceiver								
implantation tout logicielle	185 %	4,6 %		19,27	185 %	5 %		19,33
implantation logicielle/matérielle	17,1 %	4,6 %	71,2%	9,44	17,2 %	5 %	71,2%	9,49

TAB. III.2 – Taux d'utilisation des composants matériels

L'analyse des résultats renvoyés, traduit la faisabilité du système dans la configuration (1) en ce qui concerne l'émetteur UMTS. En effet, les taux d'utilisation des DSP, bien qu'élevés, restent inférieurs à 100% et la contrainte temporelle du temps d'exécution d'une trame est respectée (9ms<10ms). En revanche, la même configuration du récepteur UMTS n'est pas envisageable. Il est impossible de faire fonctionner un DSP au delà de 100%. Il faut donc dans ce cas alléger la charge de travail du DSP_A. De plus la contrainte du temps d'exécution n'est pas res-

pectée. Cette solution d'implantation n'est donc pas envisageable dans le cadre de l'application visée avec ces contraintes.

Dans le cas d'une contrainte applicative plus forte (configuration (2)), il se trouve que l'implantation tout logicielle, suffisante pour l'application de l'émetteur fonctionnant avec un débit de 117 kbits/s, ne l'est plus. Le DSP_C voit sa charge de travail augmenter légèrement (+1,7%) et la contrainte applicative n'est plus respectée. Pour ce qui est de l'application du récepteur, elle n'était pas réalisable avec des contraintes plus faibles. Les résultats obtenus confirment qu'elle ne l'est pas davantage avec des contraintes plus fortes.

La solution repose donc sur un changement de choix d'implantation pris en compte dans la configuration (3). Le changement d'implantation de la fonction la plus critique, en matériel (sur FPGA par exemple), influe considérablement sur les performances des 2 applications. Celles-ci deviennent conformes aux attentes du concepteur. L'implantation précédente dans le cas du récepteur, était impossible de manière uniquement logicielle. Elle le devient avec l'implantation de la fonction MFL en matériel (FPGA_A). Les taux d'utilisation des DSP_A se réduisent avec l'allégement de la charge supportée, reportée sur le FPGA (71,2%), notamment celui utilisé pour l'application du récepteur UMTS qui passe de 185% (irréalisable) à 17,1%. Les temps d'exécution sont également revus à la baisse grâce au traitement effectué en matériel, ce qui conforte (dans le cas de l'émetteur) et rend possible (dans le cas du récepteur) le respect de contraintes temporelles imposées.

La quatrième configuration s'intéresse, tout comme la seconde à appliquer des contraintes applicatives plus sévères mais avec l'utilisation de ressources matérielles supplémentaires. Cette utilisation mixte de DSP et de FPGA apparaît une fois de plus comme une solution architecturale en adéquation avec l'application, en dépit des contraintes imposées, puisque les taux d'occupation sont tous corrects et la contrainte temporelle respectée pour les 2 applications.

Une partie du gantt traduisant ces résultats est donnée figure III.9. Il est aisé d'identifier la fonction la plus critique (MFL). Les résultats d'estimation obtenus corroborent ceux issus de l'implantation réelle. La nature séquentielle de l'application, et l'homogénéité des périodes de l'application nous conduit à un cas d'étude favorable et explique la qualité des résultats obtenus. L'élargissement des tests à des applications plus parallélisables avec des partages de données seraient nécessaires à la confirmation de ces résultats. Toutefois, un des objectifs du projet RNRT était, d'arriver à valider la faisabilité d'un système Radio Logicielle depuis une modélisation du système en UML. Cet objectif est donc atteint et l'environnement développé offre les outils nécessaires au concepteur.

III.4 Conclusion

L'outil développé satisfait donc aux attentes, car il permet la modélisation et la spécification de systèmes Radio Logicielle. Il apporte en outre des résultats d'aide à la décision concernant les choix d'implantation offerts à un concepteur pour une application type, en tenant compte des différentes contraintes, applicatives et architecturales, imposées. Comparativement aux résultats expérimentaux obtenus sur plate-forme réelle, cette conception à l'aide de l'outil A3S, pour ce degré de complexité, a été effectuée en quelques heures, alors que le développement réel de l'application à pris plusieurs mois chez Mitsubishi. L'analyse, quant a elle, est quasi immédiate (dans le cas présent quelques secondes), et le résultat obtenu, qui est la faisabilité du système, est juste, puisque le système a été réalisé sur la plate-forme. Un concepteur peut donc tester la faisabilité de réalisation d'une application sur une architecture donnée en une seule journée. Ce

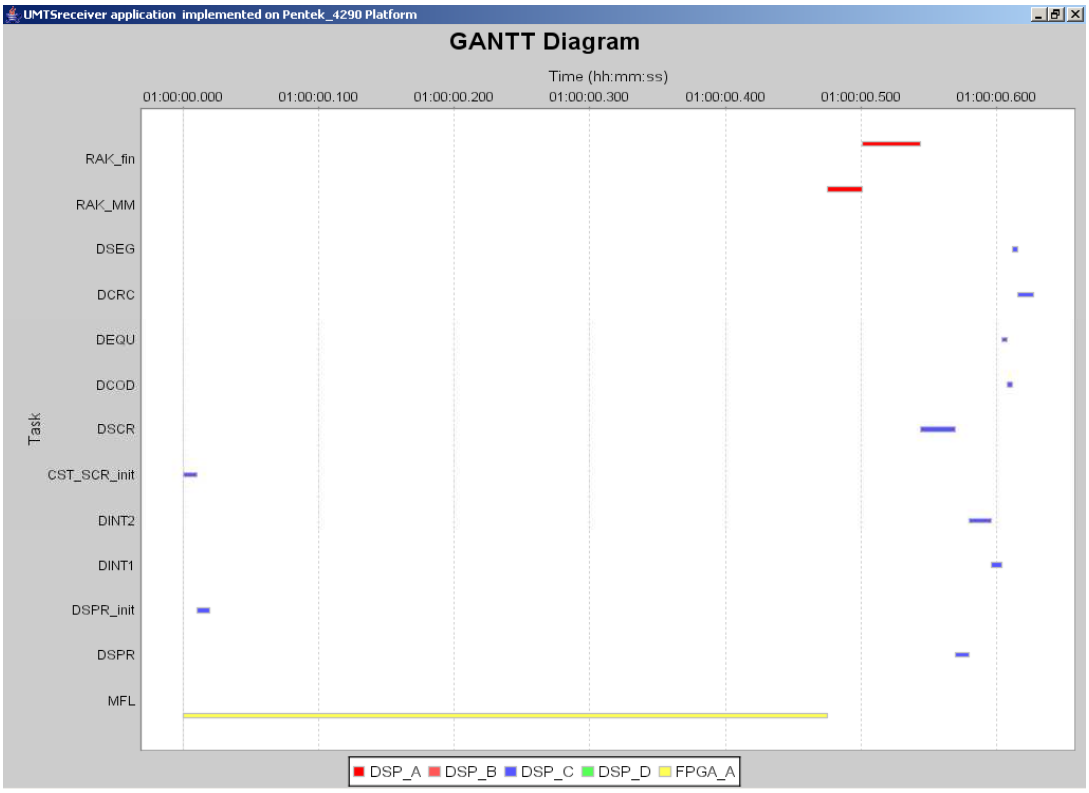


FIG. III.9 – Gantt du récepteur UMTS

temps minime, passé pour s'assurer de la faisabilité d'une implantation, est un gain de temps précieux dans un processus de conception qui réclame une dizaine de mois, car il assure au concepteur que la voie dans laquelle il s'engage (en terme de solution d'implantation effectuée) répondra à ses exigences.

Chapitre IV

Extension aux systèmes d'exploitation

L'exemple applicatif présenté dans le chapitre précédent ne nécessitait pas l'utilisation d'un système d'exploitation temps réel. Or le profil A3S dispose des éléments permettant la conception de tels systèmes. C'est pourquoi ce chapitre s'applique à présenter la méthodologie à mettre en œuvre pour disposer des lois permettant de prendre en compte le surcoût de l'utilisation d'un OS particulier. Les résultats obtenus par cette méthodologie sont ensuite utilisés dans une application de type benchmark, qui fait appel à l'ensemble des possibilités de conception, relatives aux systèmes d'exploitation, offertes par le profil. Les résultats obtenus sur le benchmark sont suivis de ceux obtenus sur une application de traitement d'image (application de tracking). Cette seconde expérience fait appel à moins de ressources mais correspond à une application réelle. Dans les deux cas, une présentation des différentes applications précède la présentation des étapes du flot de conception intégré au profil A3S. L'analyse des résultats d'estimation retournés par l'outil clôture chacune des expériences réalisées, en prononçant la faisabilité où non du système modélisé.

IV.1 Méthodologie d'acquisition de l'*overhead* d'un système d'exploitation

La faisabilité d'un système se détermine par l'existence d'un ordonnancement de ce système dans la configuration matérielle/logicielle choisie par le concepteur. Dès lors que l'application utilise un système d'exploitation, ce système d'exploitation ajoute des contraintes supplémentaires (coût temporel d'exécution, coût en surface, coût en ressources utiles pour son fonctionnement) dont il faut tenir compte dans le processus de vérification de faisabilité. Si jamais la configuration matérielle ne peut supporter l'ajout d'un OS, il faut pouvoir le détecter dans le processus de conception. De même, il faut tenir compte des contraintes et les intégrer au processus de calcul de l'ordonnancement. C'est pourquoi il est important de pouvoir identifier ces contraintes supplémentaires, les évaluer et les intégrer dans notre modèle d'exécution. Les contraintes supplémentaires, liées à l'OS, ajoutées dans le profil, ont été explicitées dans le paragraphe II.2.1. Il s'agit désormais de montrer comment estimer le surcoût temporel à partir de ces attributs. Des lois d'estimations des surcoûts engendrés par les différents services de l'OS ont été définies à partir de prises de mesures effectuées à travers un *benchmark* exécuté sur une plate-forme d'expérimentation. Nous nous intéresserons donc à présenter dans un premier temps les choix motivant l'utilisation de l'environnement d'expérimentation utilisé. Puis nous présenterons la méthodologie suivie pour obtenir les lois d'estimations des différents surcoûts

de l'OS qui seront explicités dans un dernier paragraphe.

IV.1.1 Choix de l'environnement expérimental

Pour obtenir les lois d'évaluation des différents surcoûts temporels dus à l'utilisation d'un système d'exploitation, il a été choisi un système d'exploitation représentatif. Ce choix s'est porté sur le système d'exploitation MicroC/OS-II de Jean-Jacques Labrosse [102]. Une des premières raisons justifiant ce choix est qu'il s'agit d'un système d'exploitation temps réel dont les sources sont disponibles, contrairement aux systèmes d'exploitation commerciaux Windows Ce.net ou VxWorks de Wind River. Nous disposons donc des codes sources qui nous permettent d'identifier dans les détails, les mécanismes mis en œuvre. Cela nous permet également de rajouter du code supplémentaire aux endroits stratégiques pour obtenir des résultats de performances précis sur des points particuliers de fonctionnement. Le second argument qui nous a fait retenir MicroC/OS-II (μcos), réside dans les nombreuses possibilités offertes par ce système d'exploitation, en terme de services proposés et en terme de plates-formes matérielles pouvant supporter ce système d'exploitation. Le profil développé se veut de pouvoir modéliser n'importe quelle application, n'importe quelle architecture matérielle (radio logicielle en particulier), et n'importe quel système d'exploitation. μcos est donc un bon exemple représentatif de ces possibilités car la liste des processeurs qui le supporte comprend aussi bien des processeurs physiques (80x86, powerPC, Arm, 68HC11, etc.) que des processeurs synthétisables (Xilinx MicroBlaze, Altera NIOS II). Il possède surtout les principaux services requis pour un système d'exploitation à savoir des services :

- liés à la gestion des tâches (création, suppression, suspension, changement de priorité, etc.),
- liés aux processus de synchronisation (service de sémaphore, de mutex),
- liés aux processus d'inter-communication (IPC) (service de mailbox, de message queue, de flag),
- liés à la gestion mémoire,
- liés à la gestion du temps,
- divers tels qu'une tâche de statistique, un service de dé/verrouillage de l'ordonnanceur.

ThreadX de la société Express Logic Inc., dont le code source est lui aussi disponible, est un autre système d'exploitation intéressant car il offre diverses politiques d'ordonnancement (prioritaire par fifo, round robin, time-slicing) [101]. Il reste néanmoins moins portable que μcos, et possède moins de services que ce dernier. La politique d'ordonnancement unique, proposée par μcos, repose sur les niveaux de priorités des tâches. Il est capable de supporter 64 niveaux de priorité (contre 32 seulement par ThreadX) qui lui permettent de supporter l'exécution et l'ordonnancement d'une soixantaine de tâches différentes car certaines priorités sont réservées (la priorité la plus haute est réservée à l'ordonnanceur, la plus faible à la tâche de statistique). La priorité des tâches avec μcos est décroissante, c.-à.-d. que la tâche la plus prioritaire se voit attribuer la valeur la plus faible, et inversement la tâche la moins prioritaire aura la valeur la plus élevée. MicroC/OS-II possède également un mécanisme de changement de priorité, dans le sens ou lorsque l'application utilise le service de synchronisation par mutex, ce service attribue une priorité au mutex lors de sa création. La tâche qui attend le mutex voit sa priorité héritée de celle attribuée au mutex lorsque celui-ci est signalé par une tâche, permettant ainsi à la tâche en attente d'être ordonnancée devant les autres tâches éligibles ayant une priorité supérieure à sa priorité de départ et inférieure à la priorité dont elle vient d'hériter. Cet héritage disparaît dès qu'elle libère le mutex et elle retrouve sa priorité de départ.

L'implantation de µcos a été effectuée sur la plate-forme ML310 de chez Xilinx [107]. Cette plate-forme de prototypage possède un virtex-II pro XC2VP30, comprenant 2 PowerPC 405 matériels, et dont les ressources logiques permettent de synthétiser des processeurs MicroBlaze [108]. Le choix s'est porté sur le processeur synthétisable MicroBlaze afin d'avoir une plate-forme hétérogène considérée comme un SoC, avec un seul composant matériel (le FPGA) encapsulant une matrice reconfigurable, un processeur et des ressources de communication. Cette configuration va ainsi nous permettre de vérifier si la modélisation utilisant le profil A3S, supporte la modélisation du système d'exploitation mais aussi l'instanciation d'un processeur sur une autre ressource matérielle (type FPGA). C'est un cas intéressant car le système d'exploitation est instancié sur un processeur qui lui même est instancié sur un FPGA. Cet environnement nous a servi pour valider les lois d'estimation des surcoûts. Une seconde plate-forme matérielle a également été utilisée pour confirmer les lois obtenues. Il s'agit d'un kit d'évaluation du NIOS II d'Altera [109] où µcos a une fois de plus été instancié sur un processeur synthétisable implanté sur FPGA. La même architecture SoC a donc été conservée, seuls les composants ont été modifiés, car µcos a été instancié sur le processeur NIOS II d'Altera, lui même implanté sur un FPGA Cyclone EP1C12. La même méthodologie a été suivie, et les prises de mesures ont été effectuées dans les mêmes conditions en utilisant le même code adapté au portage de µcos sur le NIOS II.

IV.1.2 Méthodologie employée

Dès lors qu'une application utilise un système d'exploitation, celui-ci impacte les performances temporelles du système par l'ajout de plusieurs surcoûts (*overhead*). En effet, les mécanismes de communication, de synchronisation et d'exécution des fonctions se retrouvent sous le contrôle d'un ordonnanceur. La figure IV.1 illustre l'effet que peut avoir un système d'exploitation sur le temps d'exécution d'une application. La configuration de l'OS, la création des tâches, les mécanismes à utiliser et leur utilisation ajoutent du temps à l'exécution de l'application. Ces *overhead* dépendent de plusieurs facteurs que nous avons identifiés lors d'une campagne de mesure. Cette campagne a été menée sur différentes cibles matérielles, Virtex-II pro de chez Xilinx et Cyclone EP1C12 de chez Altera, sur lesquels ont été instancié µcos. Elle nous a permis d'extraire des lois d'approximation des différents *overhead* suivant la configuration de l'OS et les services utilisés. Ce sont ces *overhead* que nous intégrons dans le calcul de l'ordonnancement lorsqu'un système d'exploitation est utilisé.

La méthode mise en place pour l'obtention de ces lois d'approximation, est la prise de mesures temporelles des surcoûts d'utilisation des différents services du système d'exploitation (dans différentes configurations), à l'aide d'un timer cadencé à la fréquence d'horloge, et dont la valeur du compteur nous sert de référence. Chaque point de mesure est le résultat du moyennage de 100 à 1000 mesures de la même exécution, ce qui garantit le relevé effectué. Les différentes configurations ont été judicieusement choisies pour mesurer l'ensemble des différents *overheads* dus à (aux) :

- l'initialisation du système d'exploitation (réservation d'espace pour les tables de tâches,)
- la fréquence tick de l'ordonnanceur (fréquence d'appel de l'ordonnanceur),
- changements de contexte,
- la création d'une tâche,
- l'utilisation des services de synchronisation (sémaphore, mutex, flag),
- l'utilisation des services d'inter-communications entre tâches (message queue, mailbox).

Les étapes de la méthode suivie sont illustrées sur la figure IV.3. Elles permettent de déter-

miner les paramètres qui influent sur les différents surcoûts que nous avons identifiés.

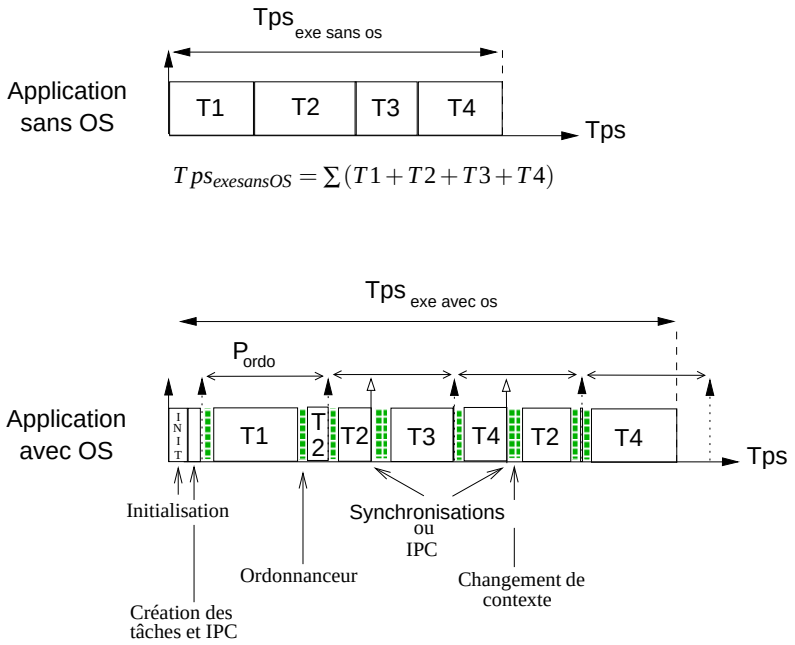


FIG. IV.1 – Effet de l'utilisation d'un système d'exploitation sur le temps d'exécution d'une application

- ☞ La première étape consiste à mesurer le temps d'exécution, sans système d'exploitation, de la tâche qui va être utilisée pour déterminer l'effet de l'ordonnanceur. Lorsqu'une tâche est exécutée sur un processeur, elle s'exécute pendant un temps t. Lorsqu'elle s'exécute sur le même processeur avec système d'exploitation, son temps d'exécution peut être supérieur suivant la période de l'ordonnanceur. Elle peut en effet être interrompue par celui-ci comme c'est illustrée sur la figure IV.2. Son temps d'exécution devient alors $tps_{exe} = t + \Delta$ où Δ est le nombre de fois que l'ordonnanceur s'exécute pendant l'exécution de la tâche $*$ le temps d'exécution de l'ordonnanceur.
- ☞ La seconde étape consiste à déterminer l'effet de l'ordonnanceur seul, sur le temps d'exécution d'une tâche. Elle est donc interrompue si la période de celui-ci est inférieure au temps d'exécution de la tâche, comme c'est illustré sur la figure IV.2 *c*), sinon non IV.2 *b*). Il est donc important de commencer par connaître l'effet de la fréquence tick de l'ordonnanceur, afin de la prendre en compte lorsque nous allons mesurer les effets des services. Les prises de mesures sont donc effectuées dans la configuration de la figure IV.2 *c*) de deux manières. La première revient à mesurer l'effet lorsque l'on fait varier la période de l'ordonnanceur en ayant toujours le même temps d'exécution de la fonction. La seconde revient à garder la même période de l'ordonnanceur, mais cette fois nous faisons varier le temps d'exécution de la fonction. Ces deux méthodes sont utilisées pour vérifier que l'effet est identique. Dans les deux cas les variations s'effectuent en respectant $t < P_{ordo}$, où P_{ordo} est la période des interruptions de l'ordonnanceur. Il faut toujours rester attentif aux temps d'exécution mesurés afin de savoir si l'ordonnanceur est intervenu lors de la mesure.

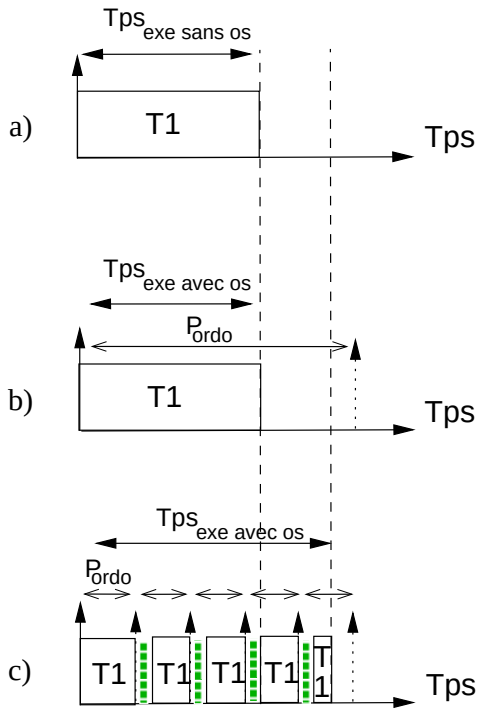


FIG. IV.2 – Effet de la période d'un ordonnanceur sur le temps d'exécution d'une tâche

☞ La troisième étape s'intéresse à la création des tâches dont le système d'exploitation à la gestion. Des mesures temporelles du temps de création de tâches sont donc effectuées en prenant soin de donner des priorités différentes à chacune des tâches (en accord avec l'ordonnanceur de μcos) et de faire varier le nombre de tâches créées afin de déterminer la loi liée à l'*overhead* de création des tâches. Ces tâches ne s'échangent aucune information et ne font appel à aucun autre service du système d'exploitation. Les mesures sont réalisées dans deux configurations. La première configuration revient à créer les tâches avant de lancer le système d'exploitation. La seconde configuration revient à créer les tâches durant l'exécution du système d'exploitation afin de vérifier que le temps de création est constant quelle que soit l'utilisation de l'OS.

☞ Ensuite le protocole suivi est redondant pour les mesures des lois d'*overhead* des différents services de synchronisation et d'IPC de μcos. Nous commençons par créer et initialiser un élément de synchronisation (sémaphore, mutex) ou d'inter-communication (*messagequeue*, *mailbox*). Nous faisons varier progressivement leur nombre, le nombre d'accès d'une et de plusieurs tâches afin d'obtenir les lois d'évolution des surcoûts temporels associés en fonction des différents paramètres identifiés.

☞ La détermination des lois d'*overhead* dues aux changements de contexte s'obtiennent en se positionnant dans les conditions de changements de contexte. Un changement de contexte a lieu lorsqu'une tâche est préemptée par une autre (car elle est en attente d'un sémaphore, où parce qu'elle libère un message attendu, où parce qu'elle rend la main pendant un court instant à l'ordonnanceur, etc.). L'ensemble des coûts des changements de contexte de toute nature sont ainsi mesurés.

☞ Durant toutes ces mesures, la configuration du système d'exploitation est restée inchangée pour ne pas influer sur les temps d'exécution et rendre impossible l'établissement des lois existantes. Les différentes mesures ont été répétées sur un nombre d'exécution suffi-

sant pour obtenir une moyenne des temps d'exécution afin de limiter les effets de bords. Seule la détermination de la loi concernant l'effet temporel de la configuration de l'OS (coût de l'initialisation) a été établie en modifiant à plusieurs reprises la configuration de l'OS (nombre de tâches que gère l'OS, nombre de priorité, nombre de sémaphore, activation des services à utiliser, etc.) afin de mesurer l'impact de l'ensemble des différents facteurs sur lesquels nous pouvons jouer. Le paramétrage de l'OS s'effectue en fonction des services de l'OS nécessaires à l'utilisateur, pour le fonctionnement de son application.

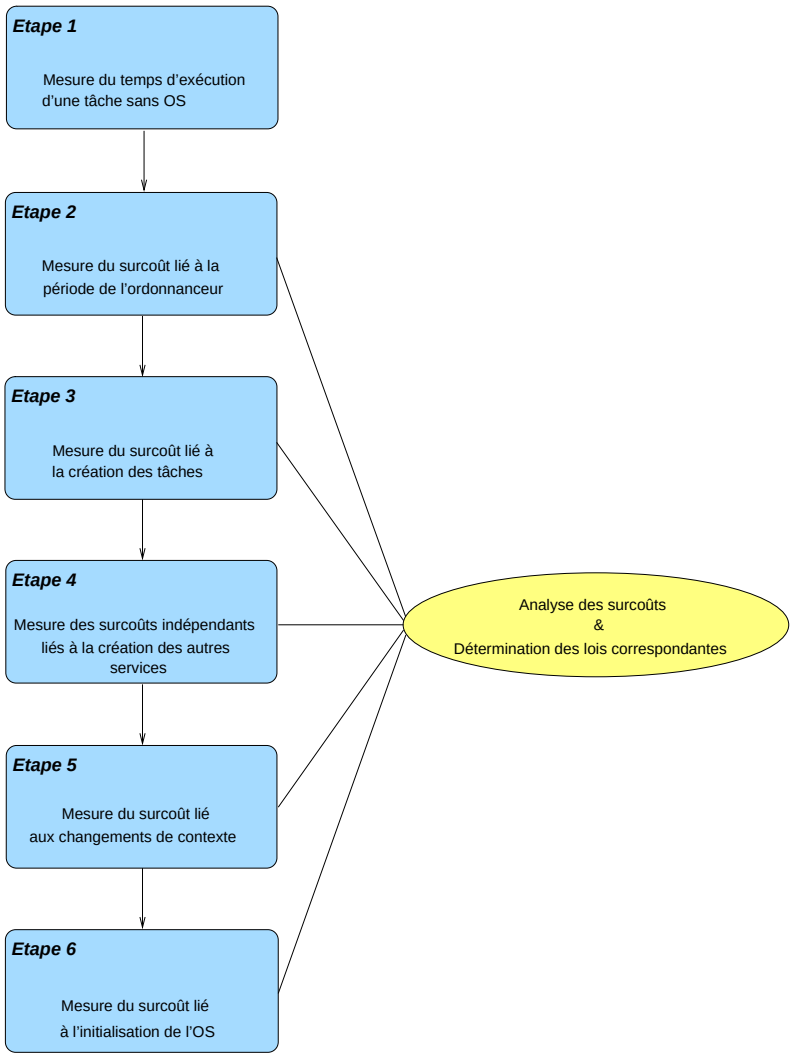


FIG. IV.3 – Les différentes étapes de la méthode d'estimation des surcoûts de l'OS

L'utilisateur spécifie donc le nombre de tâches, les services à utiliser, la profondeur des piles allouées aux tâches, le niveau de priorité maximum, etc., soit un nombre de paramètres qui déterminent la dimension des tables de gestion des tables, des événements, des Flags, des message queue. La configuration choisie impacte sur le temps d'exécution de l'initialisation de l'OS.

IV.1.3 Coût des overheads d'un système d'exploitation

Avec la méthodologie précédemment expliquée, ce sont 22 *overhead* différents qui sont à intégrer au calcul d'ordonnancement. Seulement certains *overhead* sont liés uniquement aux particularités offertes par μcos et n'existent pas (où sous une autre forme) dans les autres OS. Par exemple le service de file de messages (*message queue*) a la particularité de proposer quatre mécanismes d'envoi de messages : une tâche envoie un message à une autre tâche et le message est stocké dans une FIFO (OSQPost(...)), une tâche envoie un message à une autre tâche et le message est stocké dans une LIFO (OSQPostFront(...)), une tâche envoie un message à toutes les autres tâches et le stocke dans une FIFO (OSQOpt(...,OS_POST_OPT_BROADCAST)), une tâche envoie un message à toutes les autres tâches et le stocke dans une LIFO (OSQOpt(...,OS_POST_OPT_FRONT)). Nous avons donc déterminé un ensemble de lois résultant des mesures de surcoût effectuées d'après l'instanciation de μcos sur Microblaze (cadencé à 100 Mhz) sur un Virtex-II pro. Le tableau IV.1 ci-après, liste l'ensemble des *overhead* que nous avons identifiés avec les paramètres intervenant dans leurs calculs. L'ensemble des lois se rapportant à ce tableau, sont présentées dans l'annexe C. Les valeurs des paramètres explicités n'ont pas d'unité car elles correspondent au nombre de coups d'horloge écoulés pendant la mesure. Il suffit de multiplier ce nombre par la période de l'horloge pour obtenir le temps.

Un exemple représentatif de ces lois est la loi d'estimation du surcoût temporel provoqué par l'effet de l'ordonnanceur. Deux séries de mesures ont permis de préciser la loi. Dans la première, c'est le temps d'exécution de la tâche que nous faisons varier. Dans la seconde il s'agit de la période de l'ordonnanceur. Après avoir relevé les différents temps d'exécution de plusieurs tâches, présents sur la courbe de la figure IV.4, on détermine que la formule du temps d'exécution d'une fonction avec OS est de la forme :

$$T_{ps_{exeavecos}} = T_{ps_{exesansos}} + (\gamma * \left\lfloor \frac{T_{ps_{exesansos}}}{P_{tickordo}} \right\rfloor)$$

d'où un surcoût de la forme

$$O_{ordo} = \gamma * \left\lfloor \frac{T_{ps_{exesansos}}}{P_{tickordo}} \right\rfloor$$

avec

- γ correspondant au nombre de ticks d'horloge d'un overhead. Il est dépendant du nombre de tâches créées $Nb_{tâche}$. En effet la table des tâches de l'ordonnanceur est dimensionnée en fonction de ce nombre et met un certain temps à la scruter (ν lorsqu'il y a qu'une seule tâche). D'où $\gamma = \nu + ((Nb_{tâche} - 2) * \rho))$ avec $\nu = 1491$ et $\rho = 111$. ρ est le surcoût temporel ajouté par tâche créée supplémentaire. La soustraction par 2 tient compte de la tâche, à laquelle s'ajoute la tâche Idle par défaut (soit 2 tâches minimum pour un coût de ν).

Si on intègre les relevés obtenus en observant l'effet de la période de l'ordonnanceur sur la même tâche il apparaît que la formule appropriée se précise telle que

$$T_{ps_{exeavecos}} = \kappa + \left\lfloor \frac{\kappa}{P_{tickordo}} \right\rfloor * (\gamma + 4) \qquad \text{avec} \qquad \kappa = \gamma * \left\lfloor \frac{T_{ps_{exesansos}}}{P_{tickordo}} \right\rfloor + 4 * \left(\left\lfloor \frac{T_{ps_{exesansos}}}{P_{tickordo}} \right\rfloor - 1 \right)$$

Cette équation s'explique car le temps d'exécution avec OS correspond au temps d'exécution sans OS + le nombre d'interruption dû à l'ordonnancement * le temps que prend cet interruption (γ). Ce nombre dépend du rapport $\left\lfloor \frac{T_{ps_{exesansos}}}{P_{tickordo}} \right\rfloor$ qui détermine le nombre d'interruptions. Il

Overhead	Paramètres influants
Initialisation μcos	◇ le nombre de tâches ◇ la profondeur de pile allouée aux tâches ◇ le nombre de priorités ◇ la profondeur de la pile de la tâche IDLE ◇ le nombre d'événements possibles ◇ le nombre de message queue ◇ le nombre de flag possibles ◇ le coût d'initialisation des services (tâche, sémaphore, mutex, mailbox, flag, message queue ...)
Changement de contexte	◇ nombre de changements de contexte
Ordonnanceur	◇ période tick de l'ordonnanceur ◇ temps d'exécution de la tâche ◇ nombre de ticks d'horloge durant un *overhead* de l'ordonnanceur (γ)
Création tâche	◇ nombre de ticks d'horloge dus aux mécanismes de création s'il n'y a pas de pile associée à la tâche (α) ◇ profondeur de la pile attribuée aux tâches (s_{tss}) ◇ nombre de ticks supplémentaires imputés à l'augmentation de la profondeur de la pile (β) ◇ constante due au mécanisme de création de tâche (μ) ◇ nombre de tâches créées.
sémaphore	◇ nombre de sémaphores ◇ temps de création d'un sémaphore
mutex	◇ nombre de mutex ◇ temps de création d'un mutex
mailbox	◇ nombre de mailbox ◇ temps de création d'un mailbox
message queue	◇ nombre de message queue ◇ temps de création d'un message queue
flag	◇ nombre de flag ◇ temps de création d'un flag
Post*/Pend sémaphore	◇ nombre de Post* ou de Pend du service ◇ temps d'exécution du post* ou du pend
mutex	◇ "" "" "" ""
mailbox	◇ "" "" "" ""
messaque queue	◇ "" "" "" ""
flag	◇ "" "" "" ""

*PostFront/PostOpt(None/Broadcast/Front)
Les Post correspondent à l'envoi (expédition/libération/signalement) de messages ou signaux, alors que le pend concerne les destinataires qui demandent à l'obtenir (obtention/attente).

TAB. IV.1 – Synthèse des paramètres qui agissent sur les *overhead* ajoutés par un OS

se peut en outre que l'overhead ainsi ajouté, rajoute du temps qui provoque des interventions supplémentaires de l'ordonnanceur .

D'où l'*overhead* suivant :

$$T_{ps_{exeavecos}} = \kappa + \left\lfloor \frac{\kappa}{P_{tickordo}} \right\rfloor * (\gamma + 4)$$

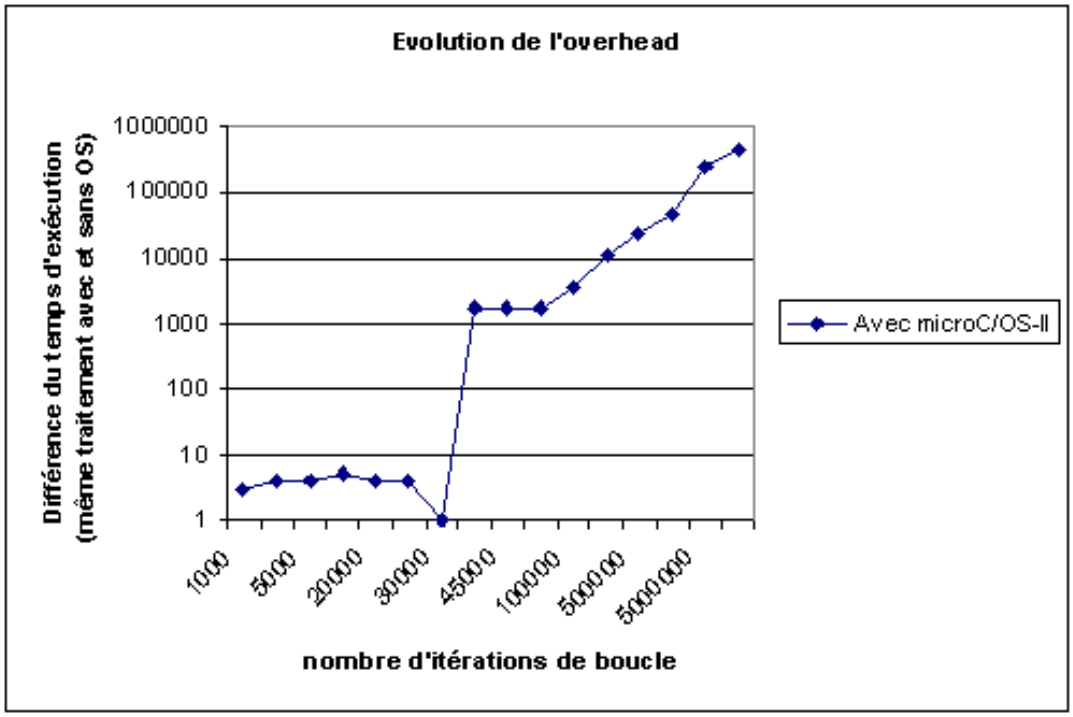


FIG. IV.4 – Courbe d'évolution du surcoût de l'ordonnanceur en fonction du temps d'exécution d'un traitement

Cette figure (IV.4) traduit bien que le temps d'exécution avec OS est, à quelques ticks près, identique au temps d'exécution de la même fonction sans OS, tant que le temps d'exécution (ici traduit par l'indice de boucle qui modifie le temps d'exécution) de celle-ci reste inférieure à une période de l'ordonnanceur. Pour un nombre de boucle inférieur à 30000, les temps d'exécution des fonctions avec et sans OS sont égaux à 5-6 ticks d'horloge près. En revanche, dès que le temps d'exécution devient supérieur, l'ordonnanceur vient interrompre la tâche en cours d'exécution et prolonge ainsi le temps d'exécution total de la fonction d'un facteur proportionnel aux nombres de ses interruptions. Le plat observé entre 40000 et 50000 itérations de boucle, signifie que l'ordonnanceur a interrompu la fonction un même nombre de fois. Ensuite la différence devient proportionnelle aux nombres d'interruption de l'ordonnanceur.

L'impact de l'utilisation d'un système d'exploitation sur le temps d'exécution d'une application donnée, exécutée sans système d'exploitation, revient à sommer le surcoût temporel de l'OS ($O_{osappli}$) aux temps d'exécution de l'application ($tps_{appli_{sansOS}}$).

$$Tps_{appli_{avecOS}} = Tps_{appli_{sansOS}} + O_{os_{appli}}$$

où

$$
\begin{aligned}
O_{os_{appli}} &= \sum Overheads\ de\ l'OS \\
&= O_{ordo} + O_{créationtâche} + O_{créationsémaphore} + O_{créationmutexgéné} + \\
&\quad O_{créationflaggéné} + O_{créationmailboxgéné} + \\
&\quad O_{créationmessagequeuegene} + O^{*}_{changementcontexte}
\end{aligned}
$$

*l'*overhead* de changement de contexte englobe la somme
des coûts de chaque changement de contexte de nature différentes.

Lorsque nous examinons les lois déterminées suivant l'utilisation des différents services et la configuration du système d'exploitation, il apparaît qu'elles n'impactent pas toutes avec la même importance le temps d'exécution des tâches. L'effet de certaines reste mineur vis à vis de celui d'autres dans une configuration où les valeurs des paramètres des lois seraient identiques. Les lois concernant les *overhead* des services (mutex, sémaphore, message queue, mailbox, flag) sont sensiblement les mêmes, le temps de changement de contexte vaut deux à trois fois plus. En ce qui concerne l'initialisation du système d'exploitation, l'effet de la profondeur de la pile et l'effet de la création des tâches ont un impact plus élevé que les effets dus à la configuration des services à utiliser. Mais au vu de toutes les lois, il apparaît que c'est l'ordonnanceur qui a l'impact le plus fort car il dépend directement du temps d'exécution des tâches qui est généralement égal à plusieurs centaines de fois l'ordre de grandeur des constantes intervenant dans les lois. Comme le temps d'exécution des tâches ordonnancées par le système d'exploitation est relativement important, comparativement aux constantes et aux valeurs des paramètres potentiellement attribuables dans le calcul des lois d'estimation, il est prévisible que l'impact de l'utilisation d'un système exploitation sera minime sur le temps d'exécution d'une application. Il le sera d'autant plus (minime) que le temps d'exécution des tâches sera important.

IV.1.4 Conclusion

La méthodologie suivie nous a permis d'identifier les principaux services d'un système d'exploitation qui génèrent un surcoût temporel. Elle nous a également mené à l'obtention des lois de calcul des différents *overhead* suite à l'analyse des mesures obtenues dans les conditions fixées par celle-ci. Les prises de mesures menant à l'obtention de ces résultats ont été réalisées sur deux processeurs différents sur lesquels était instancié le même système d'exploitation. Que ce soit sur le processeur Microblaze (Xilinx) ou sur le NIOS II (Altera), les mesures effectuées ont permis d'obtenir les mêmes lois d'estimation. Même si les résultats de mesures n'étaient pas identiques, l'architecture des deux processeurs étant différentes, les lois extraites sont identiques. Comme nous utilisons le même système d'exploitation, ce sont les mêmes mécanismes qui sont mis en œuvre. C'est à partir de ces lois, intégrées aux processus de calcul de l'ordonnancement, que les applications présentées ci-après ont pu être analysées. Ces lois ne peuvent cependant s'appliquer sans avoir les valeurs des paramètres définis, propres au système d'exploitation et au processeur sur lequel il est implanté et présent en librairie. Dans le cas présent les valeurs identifiées sont celles du duo μcos exécuté sur le processeur Microblaze (Xilinx).

Cependant l'ajout d'un OS entraîne également une consommation de ressources mémoires. En effet, un système d'exploitation nécessite de l'espace mémoire, à la fois pour stocker le code d'exécution du système d'exploitation mais aussi pour réserver l'emplacement mémoire

nécessaire à son bon fonctionnement (table de contrôle des tâches, table des évènements, profondeur de pile etc.). Ce coût, mesurable, se devrait également d'être pris en compte dans les vérifications de cohérences du mapping d'un système d'exploitation sur un processeur (dispose-t-il d'assez de mémoire interne ou de mémoire cache). De plus, l'utilisation des mémoires peut entraîner un surcoût temporel dû à un effet de seuil lorsque l'intégralité de la mémoire rapide (cache) est utilisée et qu'il faut aller l'information en mémoire lente. Ces aspects n'ont pas intégrés dans la méthodologie ni dans le profil, mais nous envisageons d'en tenir compte dans une évolution à venir de l'outil.

IV.2 *Benchmark* complet

IV.2.1 Présentation

Un *benchmark* a été réalisé afin de pouvoir utiliser et tester le plus grand nombre de possibilités proposées par le profil développé, pour la conception des systèmes avec système d'exploitation. Il repose sur un ensemble de tâches qui effectuent chacune une simple boucle d'attente, nous permettant ainsi de faire varier le temps d'exécution de chacune d'elle par simple modification de la valeur de l'indice de boucle. L'avantage de faire nous même un benchmark, est de cibler l'expérimentation sur l'utilisation de l'ensemble des services possibles généraux des OS. Ainsi nous pouvons vérifier si le profil développé adresse ces services et permet de les spécifier suffisamment pour évaluer la faisabilité d'un système en intégrant le surcoût de l'utilisation d'un OS. C'est pourquoi, dans le cas présenté ici les tâches font appel à l'ensemble des services généraux des OS. Le but est avant tout de vérifier la validité des lois d'estimation que nous avons déterminées. Pour que le test soit probant, l'application exploite de manière intensive les mécanismes de synchronisation en balayant différents cas de figures. Ces cas sont regroupés dans le tableau IV.2, qui relate l'ensemble des services utilisés par les tâches. Les possibilités couvertes sont donc l'utilisation de services entre deux tâches uniquement, la synchronisation ou l'échange de messages à plusieurs, et la combinaison de différents services entre plusieurs tâches.

Tâches	Action	Mécanismes
1 tâche	se synchronise	avec 1 autre via 1 sémaphore
		avec 1 autre via 1 mutex
		avec plusieurs autres via sémaphore et mutex
		avec plusieurs autres via plusieurs sémaphores
	se synchronise communique signalise	avec plusieurs autres via mutex, flag et message queue
	communique(nt)	avec 1 autre tâche via 1 mailbox
plusieurs tâches		avec plusieurs autres via plusieurs mailbox

TAB. IV.2 – Différents mécanismes utilisés dans le *benchmark*

IV.2.2 Modélisation

Modélisation de l'application

Le *benchmark* réalisé est donc modélisé par un diagramme d'activité. Le schéma fonctionnel de ce *benchmark* est présenté sur la figure IV.5. Cette application test utilise l'ensemble des services proposés par μcos, et que l'on retrouve généralement dans les autres OS. Elle possède également un nombre de tâches suffisant pour tester les différents cas de figure énumérés ci-dessus et engendrer ainsi des overheads plus importants. C'est pourquoi nous avons 13 tâches (dont la tâche de configuration des timers) de priorités différentes où les tâches possèdent des priorités décroissantes suivant leur numéro d'affectation. La tâche 1 (T1) est la plus prioritaire (priorité de 10) suivie de la tâche 2 (T2) (priorité de 11), de la tâche 3 (priorité de 12) etc. jusqu'à T13 de priorité 22. Les mécanismes de synchronisation, et d'échanges de messages mis en œuvre dans ce *benchmark* sont illustrés sur le schéma fonctionnel par les flèches orientées vers le haut (montante) et le bas (descendante) surmontées du type de mécanisme. Les flèches montantes signifient que la tâche émet un POST dont la nature est identifiée par le texte associé (ex : SEM1 signifie sémaphore 1), tandis que les flèches descendantes signifient qu'elles attendent un signal ou un message particulier, PEND, identifié par le texte associé. Les destinateurs et destinataires des signaux (synchronisation, signalisation, message) sont identifiables par les liens qui les relient. Ce sont donc : 10 sémaphores différents de profondeur 1 (valeur des compteurs des différents sémaphores initialisés à 1), 2 mutex, 2 mailbox et 1 message queue qui sont utilisés par les tâches du *benchmark*.

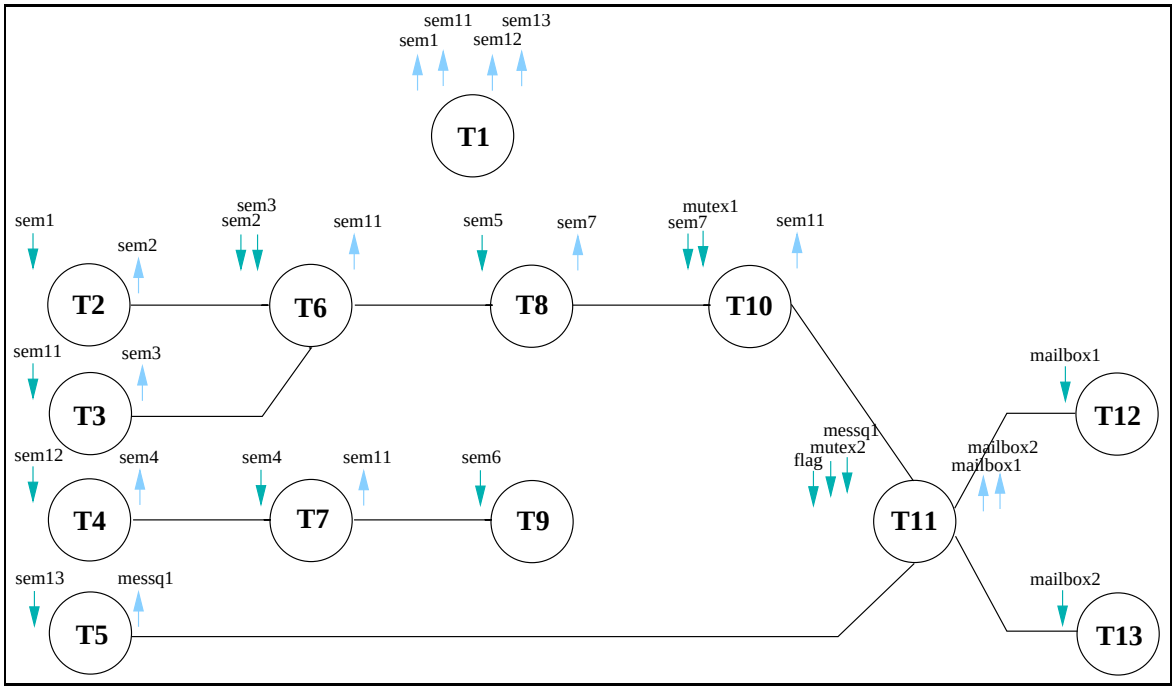


FIG. IV.5 – Schéma fonctionnel de l'application *benchmark*

Modélisation de la plate-forme matérielle

La plate-forme matérielle utilisée pour la réalisation des mesures est la plate-forme ML310 de chez Xilinx. Toute la plate-forme n'est pas modélisée, car seul le FPGA Virtex-II pro

XC2VP30 sur lequel est implanté un processeur Microblaze, assimilable à un SoC de faible complexité, nous intéresse. Ces deux composants matériels (le FPGA et par conséquent le processeur) peuvent accéder à de la mémoire externe DDRAM présente sur la carte via un bus OPB. Le diagramme de déploiement modélisé comprend donc un ensemble de 9 composants matériels que l'on retrouve sur le diagramme de déploiement après l'étape de mapping de la figure IV.6 :

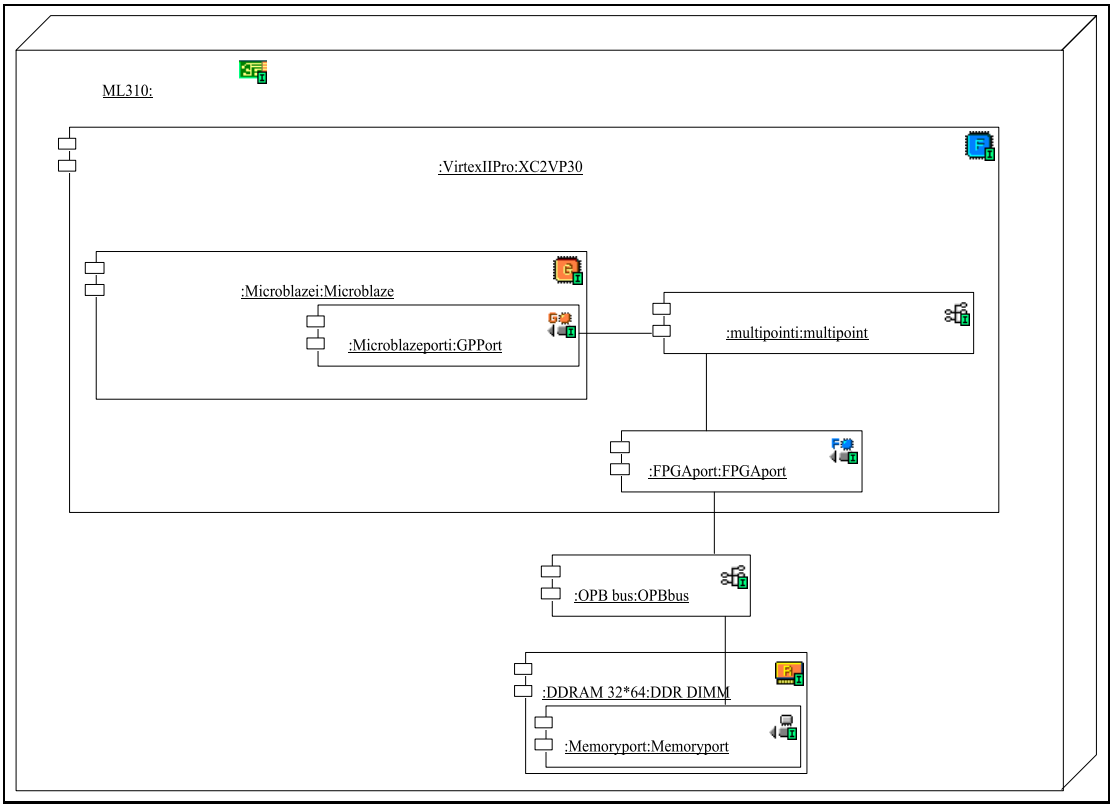


FIG. IV.6 – Plate-forme matérielle modélisée avec Objecteering et le profil A3S

- la carte mère ML310
- le FPGA Virtex-II pro et un de ses ports
- le processeur Microblaze et un de ses ports
- les composants d'interconnexion, un fil, le bus
- la mémoire DDRAM et un de ses ports

Le processeur tout comme le FPGA sont cadencés à une fréquence d'horloge de 100Mhz (ce qui nous donne une période de 10ns). Le paramétrage du Microblaze que nous avons effectué, lui confère une mémoire locale de données et d'instruction commune de 64Kbits. Le FPGA quant à lui possède 30816 blocs logiques, 8 blocs multiplieurs et 2448 Kbits de mémorisation. L'ensemble de ces caractéristiques renseignent les différents paramètres correspondants, qui permettent les vérifications de cohérence du mapping effectué par le concepteur. Les caractéristiques du processeur n'ont pas été prises en compte dans l'établissement des surcoûts liés au système d'exploitation car l'objectif est de considérer l'impact du système d'exploitation seul, indépendamment des caractéristiques du processeur. Il aurait été intéressant de vérifier les effets de l'architecture même du processeur sur lequel est instancié le système d'exploitation, de la présence ou non d'un cache de données et d'instructions, mais nous n'avons pas expérimenté ces cas. Si nous essayons tout de même d'évaluer cet impact, cela reviendrait à une variation

du temps d'exécution de l'ordonnanceur, en nombre de coups d'horloge (tick), en fonction de l'utilisation ou non du cache. Ce temps est le paramètre qui intervient directement dans la loi d'estimation du surcoût dû à l'ordonnanceur. Il faudrait donc ajouter une valeur supplémentaire en librairie qui serait utilisée le cas échéant.

Implantation

Dans le cadre de l'expérimentation de ce *benchmark*, l'application a donc été implantée sur la plate-forme ML310 présentée au IV.1.1. C'est dans l'étape de mapping que nous spécifions l'utilisation d'un système d'exploitation pour l'exécution des tâches de l'application. Cette application utilise le système d'exploitation μcos, représenté par l'instance d'un composant μcos, stéréotypé «*a3s-OSComponent*», et préalablement renseigné dans la bibliothèque des composants systèmes d'exploitation. Cette instance de composant est implantée sur l'instance du composant MicroBaze, lui-même implanté sur l'instance du Virtex-II pro. L'ensemble des 13 fonctions de l'application a alors été instancié sur l'instance du processeur comprenant le système d'exploitation. Au vu du schéma fonctionnel de la figure IV.5, les différents services de μcos que réclame l'application ont été également instanciés sur l'instance du système d'exploitation. Les paramètres des différents éléments UML représentés sont bien évidemment renseignés suivant les contraintes applicatives et architecturales connues. Dans cet exemple, chaque tâche n'est itérée qu'une seule fois, les temps d'exécution de chacune d'elles sans utilisation du système d'exploitation sont supposés connus et sont donnés au paragraphe suivant. Le système d'exploitation ainsi que ses services est lui aussi spécifié. Nous avons retenu de dénommer MicroC/OS-II en μcos, nous avons spécifié une profondeur de pile pour la tâche Idle de 256*32bits, avec pour chacune des tâches une profondeur de pile allouée de 256*32bits également (valeur par défaut). Le seul service à avoir des paramètres est le service d'ordonnancement [102]. Le dimensionnement de la taille des piles est soumis aux ressources mémoires disponibles que nous réservons à cet effet pendant la configuration du système d'exploitation. La fréquence tick spécifiée est de 100hz (période tick ordo fixée à 10ms), un seul ordonnancement est possible avec μcos, il s'agit d'un ordonnancement à priorité et qui est préemptif. Le niveau de priorité maximal (dans le cas présent cela correspond à la priorité la plus faible) configuré tout au long de l'expérience est de 22. Nous sommes dans un cas particulier avec une tâche par niveau de priorité (la priorité est indépendante de la période), cela dit avec d'autres types d'ordonnancement comme un *Rate Monotonic Scheduling*, basé sur la période des tâches, ou un ordonnancement dynamique comme un *Earliest Deadline First*, seul le critère de priorité diffère. Cela ne remet pas en cause les lois d'estimation obtenues mais uniquement les valeurs des paramètres de ces lois.

IV.2.3 Résultats d'analyse et d'estimation

Lorsque tous les paramètres ont été spécifiés et que les modélisations ont été vérifiées comme étant correctes, il est alors possible de vérifier la faisabilité du système. Les lois d'estimation déterminées n'étant pas encore intégrées à l'outil, c'est manuellement que nous pouvons vérifier si les valeurs d'estimation calculées à partir de ces lois, sont cohérentes avec les prises de mesures effectuées pendant l'exécution du benchmark sur la plate-forme ML310.

Le tableau de la figure IV.3, synthétise les résultats d'estimations effectués par l'outil et classés suivant les *overhead* générés par le système d'exploitation. Nous pouvons constater que l'ordonnanceur est le service qui représente la plus grande part des surcoûts entraînés par

l'OS. Les prises de mesures effectuées sur la plate-forme sans que le système d'exploitation

Nature du surcoût	paramètres	valeurs/nombre	Valeurs du surcoût en nombre de tick
Initialisation de l'OS	nombre de tâches	13	476
	profondeur de la pile des tâches	256	23460
	nombre d'événements	15	708
	nombre de message queue	1	43
	nombre de flag	1	46
	nombre de priorité	24	983
	Services de tâches		2423
	Services de message queue		285
Changements de contexte		14 (10+2+1+1)	26117
Création des tâches		13	333545
Création des sémaphores		10	3352
Création des mutex		2	896
Création des mailbox		2	704
Création des flag		1	325
Création des message queue		2	580
Ordonnanceur			6306548

TAB. IV.3 – Estimation des surcoûts engendrés par μcos

ne soit instancié sont résumées dans le tableau IV.4. Ces mesures servent à spécifier les temps d'exécution de chacune des tâches qui interviennent dans le calcul du temps d'exécution avec le surcoût engendré par l'OS. Le coût temporel d'une exécution complète de l'application, résulte de la somme des temps d'exécution mesurés pour chacune des tâches. Ce temps mesuré s'élève, sans utilisation de l'OS, à 2316641011 ticks d'horloge (la période de l'horloge est de 10 ns). La même expérience menée avec l'utilisation de μcos nous donne une mesure du temps d'exécution total de 2323339867 ticks d'horloge (toujours cadencé à une période de 10 ns).

Le temps estimé par l'outil d'estimation prend en compte toutes les valeurs des paramètres entrant dans le calcul de l'ordonnancement avec OS, et nous donne un temps de 2323341502 ticks. Ce dernier temps est issu de la somme des *overhead* dus à μcos (tableau IV.3) et du temps d'exécution de l'application sans OS. L'erreur d'approximation obtenue par notre estimation est donc de $7 * 10^{-5}$% pour cette application. C'est une estimation que l'on peut considérer très satisfaisante car il n'y a pas d'erreur commise sur l'estimation. Cette précision se doit cependant d'être vérifiée par d'autres expériences (Tracking) pour affirmer l'exactitude systématique de l'estimation. Si l'on regarde les deux temps d'exécution mesurés, avec et sans OS, il s'avère que le surcoût de l'OS est infime puisqu'il ne représente que +0.28%. L'impact de l'ajout d'un système d'exploitation dans les conditions expérimentales menées est très faible. Si l'on relativise la précision précédente, il faut ramener l'erreur par rapport au surcoût engendré par l'OS.

Tâches	Temps d'exécution (nb ticks)
T2	51000126
T3	51255126
T4	76500126
T5	102000199
T6	1326000126
T7	51459126
T8	54075426
T9	54570126
T10	112200126
T11	255000252
T12	178500126
T13	4080126

TAB. IV.4 – Temps d'exécution des tâches mesurés sur la plate-forme ML310 sans OS

Ce qui fait passer l'estimation obtenue à 0.024% ($7.10^{-5} * 100/0.28$), ce qui reste toujours une estimation parfaite pour ce cas.

IV.3 Application de Tracking

IV.3.1 Présentation

Détection reconnaissance et suivi d'objets

L'application tracking est une application de traitement d'image qui, à partir d'un flux vidéo, identifie les différents éléments en mouvement provenant du flux d'images. Le système se compose également d'un superviseur et d'un gestionnaire d'alimentation qui spécifie au superviseur l'état des batteries disponibles. Selon cet état, le superviseur prendra les dispositions qui s'imposent pour avertir l'utilisateur de l'autonomie restante, et limiter la consommation le cas échéant.

Ce système comprend donc diverses fonctions de traitement d'image :

- une fonction de moyennage sur les dernières images acquises, qui limite les perturbations subies par l'image pendant la réception, et qui permet d'identifier plus facilement le mouvement d'un élément,
- une fonction de soustraction qui permet d'isoler les éléments mouvants des éléments fixes,
- une fonction de seuillage qui vient affiner le résultat de la soustraction,
- une fonction d'érosion qui permet d'éliminer les pixels non significatifs d'un élément,
- une fonction de dilatation qui redonne forme aux éléments identifiés,
- une fonction de reconstruction, qui, à partir du résultat de dilatation et du résultat du seuillage redonne la forme initiale des éléments mouvants,
- une fonction d'étiquetage qui permet le comptage des différents objets mouvants,
- une fonction enveloppe qui vient reconnaître la forme de l'objet pour son identification,
- une fonction de supervision qui gère l'état de fonctionnement du système en fonction du niveau de batterie,
- une fonction de gestion de l'état de la batterie.

Cette application tracking est issue d'une application développée au CEA de Saclay et traitée dans le cadre du projet RNTL Epicure [110] dans lequel le LESTER était impliqué. Pour rendre cette application opérationnelle sur la plate-forme ML310 avec un MicroBlaze, il a fallu modifier une partie du code. En effet, cette application s'exécutait sur un ordinateur, or ici nous sommes sur un processeur qui est instancié sur un FGPA et nous ne disposons pas du même ensemble de librairies que sur un PC.

IV.3.2 Modélisation

Modélisation de l'application

Dans le cas présent il ne s'agit pas d'une application de radiotélécommunication, mais le profil développé est assez riche pour couvrir les premières étapes du flot de conception d'une telle application temps réel. La démarche MDA mise en œuvre par le profil va également s'appliquer pour cette modélisation. La modélisation PIM de l'application, réalisée à l'aide d'un diagramme d'activité, fait apparaître la concurrence de certains traitements de cette application. La figure IV.7 décrivant la modélisation de l'application Tracking, retranscrit la séquentialité des opérations successives du traitement d'images (moysousseuil, resosdilrec, etiqu, envelop) en concurrence avec les tâches de supervision (manager et energycontrol). Comme il s'agit dans cette expérience de tester l'outil développé avec une application utilisant un système d'exploitation, différents traitements de l'application Tracking ont été rassemblés et représentés par un seul en fonction des résultats de traitements, ceci afin de former des tâches de granularité différentes. C'est le cas des traitements de moyennage, soustraction et seuillage qui se retrouvent représentés par le traitement moysousseuil et qui permet d'obtenir l'objet mouvant. Il en a été de même pour les traitements d'érosion et de dilatation regroupés sous l'«*a3s-ObjectCode*» erosdilrec qui permet de parfaire l'objet mouvant en éliminant des pixels parasites dus au premier traitement. Le traitement firstask a été rajouté pour les besoins de l'expérience. C'est dans cette tâche que sont configurés les deux timers, qui génère les ticks de l'ordonnanceur pour l'un, et qui compte le nombre de ticks d'horloge écoulé pour l'autre. L'ensemble des fonctions ne sont itérées qu'une seule fois durant une exécution du cycle de l'application. Aucune ne conserve de données résiduelles particulières outre les images en mémoire servant au moyennage. Le volume de données échangées entre tâches est dépendant de la taille des images traitées. Dans le cas présent, cette taille est de 235*155, soit 235 colonnes et 155 lignes (36425 pixels), où chaque pixel est codé sur 8 bits. Le volume des données échangées est donc de 291400 bits entre chaque tâche. Nous rappelons que le temps induit par le volume des données échangées est pris en compte dans le temps d'exécution des tâches. L'ensemble de ces informations renseignent les paramètres des "*ActionState*".

Modélisation de la plate-forme matérielle

La plate-forme matérielle utilisée est identique à celle présentée dans le cas de l'application de *Benchmark* au paragraphe IV.2.2. La flexibilité de l'outil nous permet d'implémenter l'application Tracking sur le même diagramme de déploiement de la plate-forme ML310. Cela sous-entend que les valeurs des paramètres matériels sont celles énumérées précédemment.

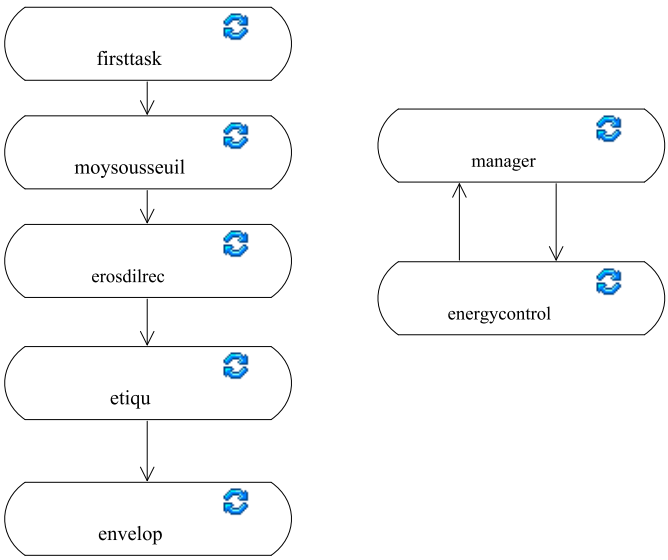


FIG. IV.7 – Diagramme d'activité de l'application Tracking

Implantation

Les choix d'implantation de l'application Tracking ont été d'implanter l'ensemble de l'application en logiciel sur processeur, en utilisant le système d'exploitation μcos. La réalisation de ce *mapping* revient à instancier le système d'exploitation μcos issu de la librairie de système d'exploitation sur l'instance du processeur Microblaze. Ensuite comme le montre le diagramme de déploiement après mapping de la figure IV.8, l'ensemble des traitements du diagramme d'activité sont liés aux composants logiciels de la librairie de composants logiciels (IP) instanciés sur le Microblaze. Au vu des services offerts par le système d'exploitation, chaque service, issu de la bibliothèque de services du système d'exploitation, et requis par l'application est instancié sur μcos. Dès lors, en fonction des services utilisés par chacune des tâches (utilisation de sémaphore, de mutex, etc.), un lien stéréotypé (a3s-Link) est ajouté entre la tâche et le service utilisé. Ce lien possède les attributs spécifiant la nature du service utilisé (ex : OS-SemPost), la profondeur (ex : valeur du compteur), et le nom du service associé instancié. Le nom du service permet l'identification des tâches qui sont, par exemple, synchronisées sur le même sémaphore. Dans le cas de l'application, chacune des tâches utilise un des services de l'OS pour se synchroniser, ou échanger un message. Les services utilisés sont donc : le service permettant de signaler (POST) et d'attendre (PEND) un sémaphore, le service de mailbox qui permet d'envoyer un seul message dans une *mailbox* et le service qui est le cœur d'un système d'exploitation, le service d'ordonnancement. Les paramètres des services instanciés sont identiques à ceux présentés au paragraphe IV.2.2 sauf pour certains énumérés ci-dessous. Le temps d'exécution de chaque tâche implantée, est renseigné d'après les mesures temporelles acquises sur la plate-forme ML310. Ces mesures ont été effectuées sans utiliser de système d'exploitation lorsque l'application était exécutée par le processeur MicroBlaze. Les temps d'exécution obtenus sont résumés dans le tableau IV.5. Ils permettent de déterminer le temps d'exécution total de l'application sans utiliser de système d'exploitation (soit : 195792614).

Le système d'exploitation issu de la bibliothèque de système d'exploitation renseigne sur la profondeur de pile de la tâche Idle utilisée par l'OS, qui est toujours de 256 (*32bits soit 1024 Bytes), et la profondeur de pile des tâches (256). Le service d'ordonnancement UCOSScheduling de μcos spécifie toujours la politique à priorité préemptive, avec un niveau de priorité

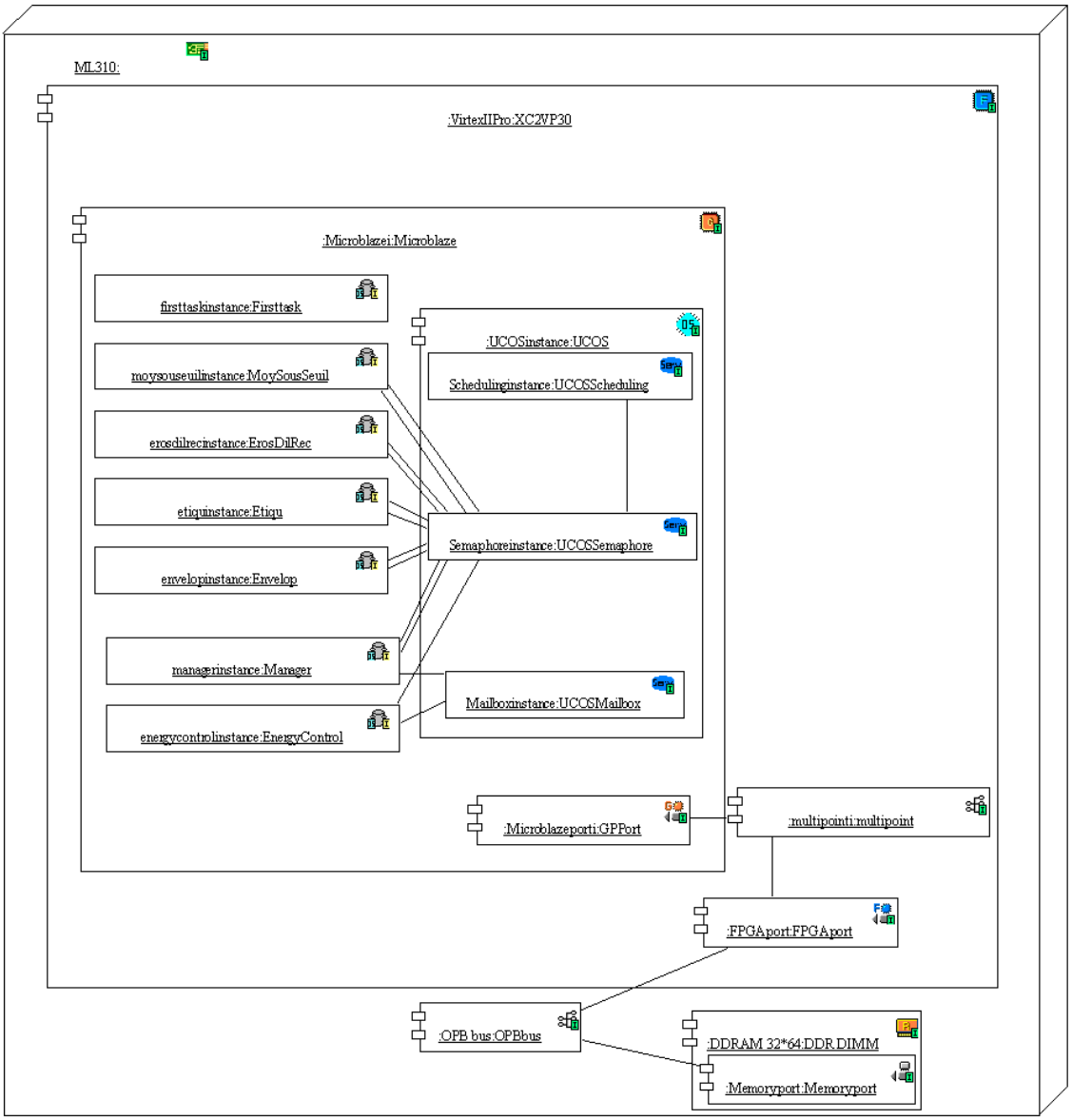


FIG. IV.8 – Application Tracking déployée sur la plate-forme

Tâches	Temps d'exécution en ms
MoySouSeuil	1354,35
ErosDilRec	461,02
Etiqu	104,44
Envelop	16,23
Manager	21,87
EnergyControl	0,00146
Application Tracking : 1957,92 ms	

TAB. IV.5 – Temps d'exécution des tâches mesurés sans OS

minimal de 15 (utilisé pour l'expérience car la priorité minimale donnée pour la tâche Envelop est de 14, or la tâche Idle à une priorité plus faible soit une priorité de 15).

IV.3.3 Résultat d'analyse et d'estimation

Les différentes modélisations PIM, PSM effectuées et vérifiées garantissent que les modélisations sont correctes et que nous disposons de toutes les informations utiles aux calculs de faisabilité et de performances. Les attributs des composants du système d'exploitation sont utilisés, ainsi que les informations provenant des modélisations. Il est ainsi possible de déterminer le nombre de tâches supportées par l'OS, les services mis en œuvre, les dépendances des tâches, et le nombre de changements de contexte. Dans l'exemple traité, les tâches ne sont exécutées qu'une seule fois et elles ont toutes un niveau de priorité différent, ce qui facilite le calcul. Le diagramme d'activité permet cependant de spécifier le nombre d'itérations (donc d'exécution) de chacune des tâches. Les mécanismes de synchronisation sont également spécifiés entre les tâches sur les transitions. Il est donc possible de déterminer le nombre de changements de contexte indépendamment de la politique d'ordonnancement. Seul les changements de contexte par interruption, autre que l'ordonnanceur, ne sont pas pris en compte dans notre méthode. Ce qui nous intéresse dans cette partie c'est de vérifier que les lois déterminées nous permettent d'estimer convenablement pour le niveau système, le surcoût d'un OS. C'est à dire que les résultats des estimations temporelles du surcoût de l'OS, présentés dans le tableau IV.6, avoisinent les résultats réels obtenus par les prises de mesures sur la plate-forme.

Nature du surcoût	paramètres	valeurs/nombres	Valeurs du surcoût en nombre de ticks
Initialisation de l'OS	nombre de tâches	7	242
	profondeur de la pile des tâches	256	23460
	nombre d'événements	7	308
	nombre de message queue	0	
	nombre de flag	0	
	nombre de priorité	14	554
	Services de tâches		2423
	Services de mailbox		39
Changements de contexte		14 (12+2)	21244
Création des tâches		7	14763
Création des sémaphores		6	2032
Création des mailbox		1	378
Ordonnanceur			399746

TAB. IV.6 – Estimation des surcoûts engendrés par μcos

Ces résultats d'estimation nous renvoie un temps total d'exécution de l'application Tracking avec μcos à 196257803 ticks d'horloge. Ce chiffre prend en compte le surcoût du système d'exploitation (465189 ticks d'horloge) s'ajoutant au temps d'exécution de l'application

sans utilisation de μcos (195792614 ticks d'horloge). La mesure réelle de l'exécution de l'application avec μcos relevée est de 196192927 ticks d'horloge. L'estimation trouvée par notre méthode nous amène à avoir une précision de 0.033% sur cet exemple ce qui semble correct. Cependant si on regarde l'effet de l'OS sur le temps d'exécution, celui-ci ne représente qu'un surcoût de 0,20% ce qui ramène notre erreur à une erreur relative de 16,16%. En comparaison des résultats de l'application *benchmark* l'erreur d'estimation relative est supérieure de 16%. Ce qui est difficile à justifier car le type de graphe modélisé est similaire, et les mécanismes de synchronisation utilisés le sont dans des conditions similaires (interdépendance des tâches similaires). La seule différence notable est dans les temps d'exécution des deux applications. Le *benchmark* a un temps d'exécution supérieur à celui de l'application de tracking, néanmoins les surcoûts respectifs d'utilisation de l'OS dans les deux cas sont similaires (0.2% contre 0.28%), donc les résultats devraient l'être également. Des travaux complémentaires doivent être menés afin d'affiner les lois et ainsi permettre de dégager un intervalle de confiance concernant l'erreur d'estimation. Mais avoir une erreur d'estimation de 15/20% au niveau système n'est pas préjudiciable car il s'agit de vérifier son dimensionnement, il est par la suite raffiné.

Au regard des deux expériences, il apparaît que l'impact de l'utilisation du système d'exploitation est faible (<0.3%) comme le laissait présager les lois d'estimation. Qu'un système d'expoitation temps réel n'ajoute qu'un faible *overhead* temporel à une application qui l'utilise est rassurant. Si ce n'était pas le cas les OS ne seraient pas forcément utilisés ou leurs fonctionnements demanderaient à être remis en cause. Nous pouvons tout de même nous demander l'intérêt de modéliser l'OS à ce niveau et de prendre en compte son surcoût dans l'estimation de performances au niveau système. Cependant, faire apparaître l'OS à ce niveau permet de pouvoir, par la suite, proposer des étapes de raffinement du modèle PSM. Cela permet également de pouvoir aller vers une étape de génération de code, puisque le type de système d'exploitation est défini, le processeur sur lequel il s'exécute est lui aussi identifié, et dans le cas où le système d'exploitation est portable, les codes sources sont communs pour toutes les cibles processeurs. Il reste à disposer des fichiers de portage particuliers en librairie et générer le code adéquat en fonction des différentes tâches modélisées (dont les IP sont en librairie).

IV.4 Conclusion

Ces expériences réalisées sur plate-forme matérielle nous ont permis d'obtenir des valeurs concrètes de performances. Des mesures de performances temporelles d'exécution d'applications sans utilisation de système d'exploitation ont dans un premier temps été relevées pour estimer le surcoût temporel d'exécution des mêmes applications dans un contexte d'utilisation de système d'exploitation sur la même architecture matérielle. D'autres mesures de temps d'exécution ont, dans un second temps, été effectuées pour des utilisations particulières d'un système d'exploitation en respectant une méthodologie. Ces mesures nous ont amenés à déterminer des lois d'estimation des surcoûts temporels induites par l'utilisation d'un système d'exploitation. Les finalités sont multiples :

- avoir des systèmes (réellement implantés) à modéliser pour vérifier que le profil UML développé est assez riche pour répondre aux besoins de modélisation des différents métiers des concepteurs,
- vérifier la robustesse et l'ergonomie de l'outil généré,
- confronter les résultats d'estimation avec les résultats réels pour valider les lois déterminées.

- intégrer les lois déterminées (une fois validées) dans l'outil d'analyse RTDT.

Ces systèmes réels ont donc servi de base de travail pour confronter le profil établi à des cas pratiques et ainsi évaluer le potentiel en terme de possibilité de modélisation, spécification, vérification et résultats d'analyse. Même si les expériences menées sont restées restrictives, à la fois dans les systèmes d'exploitations utilisés (limité à μcos) et dans les applications mises en œuvre, elles permettent de démontrer les concepts et la faisabilité d'utilisation du langage UML pour exprimer les systèmes d'exploitation au niveau système. Les caractéristiques majeures des systèmes d'exploitation sont introduites au profil et elles permettent d'être manipulées dans l'outil d'analyse. De part l'extensibilité du langage UML, ces concepts peuvent s'étendre à la prise en compte d'un OS dans le cadre d'application multi-processeurs. Dans ce cas, il sera nécessaire de faire apparaître des représentations et/ou (ou les deux) d'incorporer des attributs aux profils et lois supplémentaires dans l'outil d'analyse, afin de prendre en compte les communications inter-processeurs dues aux mécanismes de synchronisation et IPC de l'OS. Les possibilités actuelles du profil développé autorisent la modélisation de systèmes multi-processeurs multi-OS mais ne proposent que des vérifications de cohérence simples.

L'ensemble des résultats issus des expériences prouve que le profil UML développé possède les informations nécessaires, en terme de sémantique, et que les règles, établies à travers les métamodèles créés, et ajoutées à l'outil Objecteering, permettent la modélisation de système en accord avec le processus de conception. Au vu des résultats encourageants obtenus, nous pouvons assurer l'intérêt de l'utilisation d'un tel profil dans un processus de conception de système SoC.

Conclusions

Conclusion

Les avancées technologiques constantes en matière de composants électroniques élargissent de plus en plus l'espace de conception des systèmes temps réel embarqués. Pour pouvoir bénéficier de ces progrès permanents, la communauté doit disposer d'outils de conception en phase avec les possibilités offertes. C'est pourquoi elle cherche à y intégrer des méthodologies de conception qui lui permettent d'accélérer le temps de mise sur le marché des nouveaux produits tout en lui garantissant la validité et les performances de ces systèmes. Pour cela il est important d'aller vers une unification du langage adopté dans ces méthodologies, afin d'augmenter l'interopérabilité entre les différents outils existants et la ré-utilisation de conceptions ou parties de conceptions déjà réalisées. Ce langage doit bien sur posséder toute la sémantique (formalisme) adaptée aux différents domaines d'application que compte l'électronique. Il est donc important de définir cette sémantique par rapport au niveau d'abstraction considéré dans le but d'exprimer l'ensemble d'un système, de procéder aux calculs de ses performances et d'obtenir une implantation du système attendue (fiable, respect du cahier des charges). Le travail présenté dans ce mémoire s'intègre dans ce processus de proposition de méthodologie répondant aux critères pré-cités.

Réponse à la problématique et travail réalisé

La problématique présentée au début de ce document était de savoir s'il était possible de proposer aux concepteurs, un environnement de modélisation leurs permettant de disposer d'éléments caractérisés pour concevoir leurs systèmes (application + matériel) et de tester leurs choix d'implantation sans aller jusqu'au bout du processus de conception. Nous y répondons par l'affirmative en proposant au concepteur un outil qui lui permet, s'il dispose d'une banque de composants virtuels (IP) ou des propriétés qui leurs sont relatives :

- de modéliser à haut-niveau d'abstraction son application telle qu'il la considère sans se préoccuper de la fonctionnalité des traitements utilisés,
- de représenter les plates-formes du commerce existantes ou d'autres plus spécifiques à son système,
- de réaliser les choix d'implantation qu'il pense judicieux, et de vérifier s'ils le sont réellement suivant ses critères,
- de ré-utiliser toutes les représentations d'applications logicielles ou de plates-formes matérielles déjà réalisées dans d'autres conceptions, pour tester les performances de différentes implantations, ou les performances d'une architecture particulière pour différentes applications,
- de tester très rapidement la faisabilité d'un système par retour de résultats de vérifications

et de performances.
Cet outil est le fruit du travail d'identification des éléments logiciels et matériels, représentés au niveau d'abstraction système, et de leurs caractéristiques utiles à la traduction de leurs propriétés. Cette identification permet de définir la sémantique et les règles d'utilisation à introduire dans le langage de modélisation UML choisi. Cette recherche a donné lieu à la définition du profil UML A3S, qui intègre les résultats de ces travaux. Il permet notamment de disposer de bibliothèques d'éléments représentant des composants matériels, logiciels et de systèmes d'exploitation. Ce document démontre en outre le potentiel de ce langage pour la modélisation et la spécification au niveau système. Il met également en évidence ses capacités à s'adapter et à évoluer, autorisant l'ajout aisé de sémantiques particulières à celles déjà introduites dans un profil UML existant. C'est un des éléments clef de notre méthodologie PBD reposant sur la démarche de conception MDA où les plates-formes sont des modélisations UML (des modèles). L'outil s'intéresse également aux vérifications des modélisations effectuées, notamment leurs cohérences et inclut les outils développés utiles aux calculs des performances temporelles du système (XAPAT, RTDT), de manière totalement transparente pour le concepteur (aucun changement d'environnement).

Résultats

Le profil A3S a été réalisé pour combler une absence de sémantique, améliorant ainsi la modélisation et la caractérisation de plates-formes SoC hétérogènes à haut-niveau d'abstraction. Il a été intégré (ainsi que XAPAT et RTDT) dans l'atelier de génie logiciel pour UML, Objecteering, développé par la société Softeam (partenaire du projet) pour être évalué. Deux applications concrètes et un benchmark ont permis de confirmer que le profil répond bien à nos attentes et que l'outil remplit correctement sa fonction. Ces applications ont pour cela été modélisées, caractérisées, et des estimations de performances ont été réalisées pour être confrontées à la réalité. La complexité et la nature des applications ont volontairement été choisies différentes (traitement d'images et Radio Logicielle) afin d'utiliser au maximum les possibilités qu'offre le profil. Les résultats d'estimation de performances de ces applications, obtenus au niveau d'abstraction système, sont apparus conformes aux résultats réels des performances issues des prises de mesures effectuées sur les systèmes implantés. Néanmoins ces résultats sont mitigés dans la mesure où l'expérimentation sur les deux types d'applications ne permettent pas la détermination formelle d'un intervalle de confiance (1 à 16%) concernant la précision des estimations. Cependant ils valident partiellement les lois d'estimation de part les faibles *overhead* (prévisibles) obtenus. Une fois validé à ce niveau de conception, le concepteur conforté sur la faisabilité issue de ses choix, peut s'engager plus précisément dans la conception de son système. Il est alors envisageable de raffiner les modèles obtenus pour aller vers la génération de code synthétisable RTL (matériel) et exécutable (logiciel).

Perspectives

L'ensemble du profil a été réalisé sur une base UML 1.4 avec des métamodèles prenant en compte certains nouveaux aspects présents dans UML 2.0. Une première perspective d'extension est l'adaptation du profil à cette nouvelle version du langage afin de pouvoir pérenniser le profil dans un atelier toujours supporté. Les ambitions du début du projet A3S, envisageaient un environnement qui allait jusque la génération de code à partir d'une modélisation haut-niveau du

système. Bien qu'elle n'ait été réalisée, certains paramètres du modèle ont été intégrés pour faciliter cette génération. Ce serait donc une seconde voie d'extension des travaux. Dans le même esprit, le concepteur décrit son application et renseigne lui même les valeurs des paramètres non-fonctionnels issues des IP, ce qui sous-entend qu'une étape de vérification fonctionnelle des éléments a déjà eu lieu. Il serait donc intéressant de pouvoir créer une passerelle en amont de notre flot, permettant de récupérer et d'utiliser les résultats d'un outil de vérification fonctionnelle tel que Matlab Simulink par exemple. Ce serait une automatisation supplémentaire.

De plus il était aussi envisagé d'inclure les concepts d'intergiciels. Ces prétentions ont cependant été revues à la baisse au vu des efforts à fournir par rapport aux moyens disponibles. La possibilité d'ajouter de la sémantique au profil a été démontrée par l'ajout des systèmes d'exploitation et de leurs services. Une seconde extension serait donc de poursuivre dans la même voie afin de proposer une sémantique concernant les intergiciels afin d'élargir la gamme des applications modélisables dans l'outil. Cet ajout serait évidemment accompagné de la prise en compte de leurs surcoûts dans le calcul d'ordonnancement et des vérifications des modélisations associées. Il serait également intéressant de pouvoir intégrer notre profil au standard du profil SDR dans une prochaine *release* pour enrichir le modèle.

Une perspective d'extension supplémentaire déjà évoquée dans ce manuscrit serait d'étendre les possibilités de modélisation des systèmes d'exploitation, accompagnées de la définition des règles de vérification et des analyses d'ordonnançabilité adaptées, permettant de prendre en compte les architectures multi-processeurs.

Des estimations supplémentaires comme le coût d'utilisation des mémoires, ou le coût en consommation du système sont des perspectives d'enrichissement des analyses proposées au concepteur pour évaluer un système à haut niveau. Certains paramètres ont d'ailleurs déjà été introduits dans une perspective d'utilisation futur, c'est par exemple le cas des paramètres de consommation des composants matériels.

Les processus de conception des systèmes temps réels seront amenés à traiter des systèmes de plus en plus complexes et les niveaux de représentation utilisés dans le flot de conception se devront d'être élevés pour être raffinés par la suite. La ré-utilisation des IP deviendra incontournable pour garder des temps de mise sur le marché acceptable. Les concepteurs ont besoin d'outils prenant en compte ces considérations et leur garantissant au plus tôt dans le flot de conception que leurs systèmes respectent leurs contraintes. Nous avons montré que le langage UML pouvait être le langage support des premières étapes de ce flot, permettant d'obtenir rapidement des confirmations de faisabilité.

Lexique

A3S	Adéquation Architecture - Application Systèmes
ADD	Area-Driven Design
API	Application Programming Interface
BBD	Block-Based Design
CIM	Computation Independent Viewpoint
EDGE	Enhanced Data rates for GSM Evolution
FDD	Frequency Division Duplex
GPRS	General Packet Radio Service
GSM	Global Mobile Service
HDL	Hardware Description Language
IP	Intellectual Property
IPC	Inter-Process Communication
IPC	Interprocess Communication
ITU	International Telecommunication Union
JPEO	Joint Program Executive Office
JTRS	Joint Tactical Radio System
MDA	Model Driven Architecture
MDE	Model Driven Engineering
MMS	Multimedia Message Service
MOF	Meta-Object Facility
MSI	Medium Scale Integration
NoC	Network on Chip
OMG	Object Management Group
PBD	Platform-Based Design
PIM	Platform Independent Model
PSM	Platform Specific Model

R.N.R.T.	Réseau National de Recherche en Télécommunications
RTL	Register Transfert Level
RTOS	Real-Time Operating System
SCA	Software Communication Architecture
SDF	Synchronous Data Flow
SDR	Software Defined Radio
SDRF	Software Defined Radio Forum
SGML	Standard Generalized Markup Language
SMS	Short Message Service
SoC	System on Chip
TDD	Time Division Duplex
TDD	Time-Driven Design
TLM	Transactional Level Modeling
UMTS	Universal Mobile Telecommunication System
UTRA	Universal Terrestrial Radio Access
Wi – Fi	Wireless Fidely
XAPAT	Xmi A3S Profile Analysis Tool
XML	eXtensible Markup Language

Bibliographie

[1] Henry Chang et al. *Surviving the SoC Revolution*. Kluwer Academic Publishers, November 1999.

[2] OMG. *UML Profile for Schedulability, Performance and Time*, january 2005.

[3] P. Kukkala et al. UML 2.0 profile for Embedded System Design. In *Design, Automation and Test in Europe (DATE'05)*, volume 2, pages 710–715, 2005.

[4] M. Edwards and P. Green. *UML for Real : Design of Embedded Real-Time Systems*. Kluwer Academic Publishers, 2003. Chapter UML for Hardware and Software Object Modeling,127-147.

[5] J. Russ. *Les Chemins de la Pensée / Terminales*. 2003.

[6] Upmal et Lackey. SPEAKeasy, the Military Software. *IEEE Communications Magazine (NY : IEEE Press)*, 1995.

[7] J. Mitola. The Software Radio Architecture. *IEEE Communications Magazine*, 33 :26–38, none 1995.

[8] Fujistu web page. http ://www.fme.fujitsu.com.

[9] Software defined radion forum web page. http ://www.sdrforum.org.

[10] *SDR forum YearBook 2005*. SDR Forum, 2005.

[11] Srikathyayani Srikanteswara et al. A Soft Radio Architecture for Reconfigurable Platform. *IEEE Communications Magazine*, 38 :140–147, February 2000.

[12] Pangan Tin et al. An Adaptive Hardware Platform for SDR. SDR Forum Contribution, august 2001.

[13] Intel. Intel Dual Core Processor Power Intel VIIV Technology Platforms. Technical Fact Sheet, 2006.

[14] International Technology Roadmap For Semiconductors 2005 Edition - Design.

[15] Joint tactical radio system web page. http ://enterprise.spawar.navy.mil.

[16] S. Bernier. *SCA Reference Implementation 2 (SCARI2) Certification Update*, June 2005.

[17] OMG. *Unified Modeling Language Superstructure (version 2.0)*, august 2005.

[18] Sunao Torri et al. Asymmetric Multi-Processing Mobile Application Processor MP211. *NEC Journal of Advanced Technology 2005*, march 2005.

[19] OMG. *PIM and PSM for Software Component Radio*, 2004.

[20] C. Moy et al. Uml profiles for waveform signal processing systems abstraction. Software Defined Radio Technical Conference, november 2004.

[21] S. Rouxel et al. Uml framework for pim and psm verification of sdr systems. Software Defined Radio Technical Conference, november 2005.

[22] A. Sowmya V. D'silva, S. Ramesh. Synchronous Protocol Automata : A Framework for Modelling and Verification of SoC Communication Architectures. In *Design, Automation and Test in Europe Conference and Exhibition*, volume I, page p. 10390, 2004.

[23] F. Doucet et al. System-on-Chip Modeling Using Objects and Their Relationships, 1999.

[24] Nec web page. http ://www.necel.com (application processor).

[25] Elie Najm Isabelle Demeure. *Les Intergiciels développements récents dans CORBA, Java RMI, et les agents mobiles*. Hermes, 2002.

[26] Xilinx. Virtex-II Pro and Virtex-II Pro X Platform FPGAs. Complete Data Sheet, 2004.

[27] Altera. MAX+PLUS II : Programmable Logic Development System. Technical Paper, 1995.

[28] Cadence. *Right On Time. Requierements For Advanced Custum Design - white paper*, 2003.

[29] Synopsys. *Design Compiler Technology Backgrounder*, april 2006.

[30] Synplicity web page. http ://www.synplicity.com.

[31] Mentor Graphics Corporation. *Catapult C Synthesis User's and Reference Manual, release 2004b edition*, June 2004.

[32] K. Bondalapati et al. Defacto : A design environment for adaptive computing technology. In *In Proceedings of the 11 IPPS/SPDP'99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, pages 570–578, London, UK,, 1999. Springer-Verlag.

[33] S. Gupta et al. Coordinated transformations for high-level synthesis of high performance microprocessor blocks. In *In DAC '02 : Proceedings of the 39th conference on Design automation*, pages 898–903, New York, NY, USA, 2002. ACM Press.

[34] O. Santieys JL. Philippe E. Martin. Gaut, an architecture synthesis tool for dedicated signal processors. In *In Proceedings IEEE International European Design Automation Conference (Euro DAC),*, 1993.

[35] Y. Le Moullec et al. Algorithmic-level specification and characterization of embedded multimedia applications with design trotter. *Jour. of VLSI Signal Processing*, 42(2), Feb 2006.

[36] Alberto Sangiovanni Vincentelli. Defining platform-based design. *EEDesign of EE-Times*, February 2002.

[37] Li Li et al. A New Platform-based Orthogonal SoC Design Methodology. In *The 5th International Conference on ASIC*, volume 1, pages 428–432, 2003.

[38] A. A. Jerraya et al. Multilanguage Specification for System Design and Codesign. In *In System Level Synthesis*, volume 1, pages 428–432, NATO ASI, 1998.

[39] E. Riccobene, P. Scandurra, A. Rosti, and S. Bocchio. A SoC Design Methodology Involving a UML 2.0 Profile for SystemC. In *DATE '05 : Proceedings of the conference on Design, Automation and Test in Europe*, pages 704–709, Washington, DC, USA, 2005. IEEE Computer Society.

[40] A. Habibi and S. Tahar. A Survey on System-On-a-Chip Design Languages. In *Proceedings of the 3rd IEEE International Workshop on System-on-Chip for Real-Time Applications (IWSOC'03)*, pages 212–215, Calgary, Alberta, Canada, 2003. IEEE Computer Society.

[41] IEEE Computer Society. *IEEE Standard VHDL Language Reference Manual*, ieeestd 1076-2002 edition, may 2002. revision of IEEE std 1076-2000 edition.

[42] IEEE Computer Society. *IEEE Standard for Verilog Hardware Description Language*, IEEEStd 1364-2005 edition, avril 2006. revision of IEEE std 1364-2001 edition.

[43] IEEE Computer Society. *IEEE Standard for SystemVerilog- Unified Hardware Design, Specification, and Verification Language*, IEEEStd 1800-2005 edition, november 2005.

[44] R. Roth and D. Ramanathan. A Higher-Level Hardware Design Methodology Using C++. In *4th High Level Design Validation and Test Workshop*, pages 73–80, San Diego, 1999.

[45] P. Bellows and B. Hutchings. JHDL - An HDL for Reconfigurable Systems. In *IEEE Symposium on FPGAs for Custom Computing Machines*, 1998.

[46] IEEE Computer Society. *IEEE Standard SystemC Language Reference Manual*, IEEEStd 1666 edition, march 2006.

[47] P. Alexander and D. Barton. A Tutorial Introduction to Rosetta . In *Hardware Description Languages Conference*, San Jose, CA, 2001.

[48] OMG. *OMG Unified Modeling Language Specification 1.4 (Action Semantics)*, january 2002.

[49] OMG. *MDA Guide Version 1.0.1*, june 2003.

[50] OMG. *Meta Object Facility (MOF) Specification*, april 2002.

[51] H Espinoza et al. A General Structure for the Analysis Framework of the UML MARTE Profile. In *MARTES :Modeling and Analysis of Real-Time and Embedded Systems*, October 2005.

[52] Bran Selic. Using uml for modeling complex real-time systems. In *LCTES '98 : Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 250–260, London, UK, 1998. Springer-Verlag.

[53] OMG. *UML Profile for Modeling QoS and Fault Tolerance Characteristics and Mechanisms*, mai 2006.

[54] OMG. *UML Profile For CORBA Specification*, april 2002.

[55] S. King J.Kwon, A. Wellings. Ravenscar-Java : A High Integrity Profile for Real-Time Java. Technical report, may 2002.

[56] M. Roux-Rouquié. Métamodèle et Langage de Modélisation pour la Biologie Intégrative. *revue trimestrielle du Réseau ECRIN*, 2004.

[57] Object management group web page. http ://www.omg.org.

[58] Franck Vallée Pascal Roques. *UML 2 en action*. 3ième edition, 2004.

[59] Bran Selic Luciano Lavagno, Grant Martin. *UML for Real : Design of Embedded Real-Time Systems*. Kluwer Academic Publishers, 2003.

[60] *UML Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE)*, february 2005. Request For Proposals.

[61] G. Gullenkson B. Selic and P. Ward. *Real-Time Object-Oriented Modeling*. 1994.

[62] E. Riccobene et al. A UML 2.0 Profile for SystemC : Toward High Level SoC Design . In *EMSOFT'05*, Jersey City, New-Jersey, USA, september 2005.

[63] E. Riccobene et al. Improving SoC Design Flow by means of MDA and UML Profiles. In *3rd Workshop in Software Model Engineering (WISME 2004)*, Lisbon, Portugal, october 2004.

[64] OMG. *OMG SysML Specification*, may 2006.

[65] Association française d'ingénierie système web page. http ://www.afis.fr.

[66] OMG. *UML Profile for System on a Chip (SoC)*, august 2006.

[67] R. Chen et al. *UML for Real : Design of Embedded Real-Time Systems*. Kluwer Academic Publishers, 2003. Chapter UML and Platform-Based Design,107-126.

[68] W.H. Tan et al. Synthetisable SystemC Code from UML Models. In *International Worshop on UML for SoC (uSoC), DAC'2004 workshop*, pages 459–468, 2004.

[69] K. D. Nguyen et al. Model-driven SoC Design Via Executable UML to SystemC. In *25th IEEE International Real-Time Systems Symposium (RTSS)*, pages 459–468, december 2004.

[70] Velocity web page. http ://jakarta.apache.org/velocity.

[71] Synopsys. *Synopsis CoCentric SystemC compiler behavioral user and modeling guide*, 2001.

[72] Qiang Zhu et al. An object-oriented design process for system-on-chip using uml. In *ISSS '02 : Proceedings of the 15th international symposium on System Synthesis*, pages 249–254, New York, NY, USA, 2002. ACM Press.

[73] A. S. Basu M. Lajolo and M. Prevostini. UML in an Electronic System Level Design Methodology. In *UML-SOC'04 - International Workshop on UML for SoC Design*, pages 47–52, San Diego, CA (U.S.A), June 6 2004.

[74] A. S. Basu M. Lajolo and M. Prevostini. UML Specifications Toward a Codesign Environment. In *FDL'04 - Forum on Specification and Design Languages*, 2004.

[75] A. S. Basu and M. Lajolo. Design and Synthesis of Reusable Platforms with Programmable Interconnects. In *UML-SOC'05 - International Workshop on UML for SoC Design*, Los Angeles, CA (U.S.A), June 2005.

[76] M. Oliveira et al. Embedded SW Design Exploration Using UML-based Estimation Tools. In *UML-SOC'05 - International Workshop on UML for SoC Design*, Los Angeles, CA (U.S.A), June 2005.

[77] B. Steinbach T. Beierlein, D. Fröhlich. Model-driven compilation of uml-models for reconfigurable architectures. In *2nd RTAS Workshop on Model-Driven Embedded Systems (MoDES '04)*, 2004.

[78] F. Balarin et al. Modeling and designing heterogeneous systems. In *Concurrency and Hardware Design, Advances in Petri Nets*, pages 228–273, London, UK, 2002. Springer-Verlag.

[79] F. Balarin et al. Metropolis : An Integrated Electronic System Design Environment. In *IEEE Computer Society*, pages 45–52, April 2003.

[80] Peter Green, Paul Rushton, and Ronnie Beggs. An example of applying the codesign method MOOSE. In *CODES '94 : Proceedings of the 3rd international workshop on Hardware/software co-design*, pages 65–72, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[81] Telelogic web page. http ://www.telelogic.com.

[82] Gnesi et al. Modular semantics for a UML statechart diagrams kernel and its extension to multicharts and branching time model-checking. *Journal of Logic and Algebraic Programming 51*, pages 43–75, April-Mayseptember 2002.

[83] P. Kukkala et al. Performance Modeling and Reporting for the UML 2.0 Design of Embedded Systems. In *Proceedings of the International Symposium on System-on-Chip (SOC 2005)*, pages 50–53, november 2005.

[84] E. Salminen et al. HIBI v.2 Communication Network for System on Chip. In *Proceedings of the International Workshop on Systems, Architectures, Modeling and Simulation*, pages 413–422, July 2004.

[85] P. Boulet et al. Visual data-parallel programming for signal processing applications. In *9th Euromicro Workshop on Parallel and Distributed Processing, PDP 2001*, pages 105–112, Mantova, Italy, February 2001.

[86] P. Boulet et al. MDA for SoC Design, UML to SystemC Experiment. In *UML-SOC'04 - International Workshop on UML for SoC Design*, San Diego, CA (U.S.A), June 6 2004.

[87] K. Asari et al. A Difference of Model Driven Process between SoC and Software Development and a Brief Demonstartion of the tool XModlink. In *International Worshop on UML for SoC (uSoC), DAC'2004 workshop*, 2004.

[88] Communication and technology systems web page. http ://www.zipc.com/english/.

[89] Sparx systems web page. http ://www.sparxsystems.com.

[90] F. Bordeleau M. Hermeling, J. Hogg. *Developing SCA Compliant System*, 2005.

[91] W. Rosenstiel A. Viehl, O. Bringmann. Performance analysis of sequence diagrams for soc design. In *2nd UML for SoC Design Workshop at 42nd Design Automation Conference (DAC)*, Anaheim, USA,, 2005.

[92] K. A. Kettler, D. I. Katcher, and J. K. Strosnider. A modeling methodology for real-time/multimedia operating systems. In *RTAS '95 : Proceedings of the Real-Time Technology and Applications Symposium*, page 15, Washington, DC, USA, 1995. IEEE Computer Society.

[93] R. Le Moigne O. Pasquier and J.-P. Calvez. A Generic RTOS Model for Real-time Systems Simulation with SystemC. In *Automation and Test in Europe Conference and Exhibition Designers Forum (DATE'04)*, volume 3, pages 82–87, 2004.

[94] V. J. Mooney and D. M. Blough. A hardware-software real-time operating system framework for socs. *IEEE Design and Test of Computers*,, none :44–51, November-December 2002.

[95] D. Messerschmitt E. Lee. Synshronous data flow. In *Proceedings of IEEE*, volume 75, pages 1235–1245, September 1987.

[96] A3S consortium. Typologie des ressources et exigences non fonctionnelles à vérifier. Technical Paper A3S project, 2005.

[97] Hedi Tmar et al. RTDT : a Static QoS Manager, RT Scheduling, HW/SW Partitioning CAD Tool. In *Proceedings of ICM*, december 2004.

[98] Zhengting He, Aloysius Mok, and Cheng Peng. Timed rtos modeling for embedded system design. In *RTAS '05 : Proceedings of the 11th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS'05)*, pages 448–457, Washington, DC, USA, 2005. IEEE Computer Society.

[99] T. Kamiuchi, H. Nakanishi, and K. Hayashi. Position paper : Operating system structure model for real-time systems. In *WORDS '96 : Proceedings of the 2nd Workshop on Object-Oriented Real-Time Dependable Systems (WORDS '96)*, page 116, Washington, DC, USA, 1996. IEEE Computer Society.

[100] F. Gauthier. Les systèmes d'exploitation temps réel. *Electronique*, pages 69–77, 2003.

[101] Express Logic Inc. User Guide. ThreadX the high performance embedded kernel. Technical report, may 2003.

[102] Jean-J. Labrosse. *Microc/OS II : The Real Time Kernel*. CMP Books, 2002.

[103] Softeam. Profiles UML et langage J : Contrôlez totalement le développement d'applications avec UML. White Paper - UML Profile Builder, 1999.

[104] Ali Dasdan, Dinesh Ramanathan, and Rajesh K. Gupta. Rate derivation and its applications to reactive, real-time embedded systems. In *Proceedings of the 1998 Conference on Design Automation (DAC-98)*, pages 263–268, Los Alamitos, CA, June 15–19 1998. ACM/IEEE.

[105] 3^{rd} generation partnership project web page. http ://www.3GPP.org.

[106] A3S consortium. Modèle de l'application système. Technical Paper A3S project, 2005.

[107] Xilinx ML310 User Guide 2005. Virtex-II pro Embedded Development Platform.

[108] Xilinx. Microblaze RISC 32-Bit Soft Processor. Logic Core, 2002.

[109] Altera. NIOS II Processor Reference Handbook. Technical Paper, 2003.

[110] J.P. Diguet et al. EPICURE : A partitioning and co-design framework for reconfigurable computing. *Microprocessors and Microsystems*, pages 367–387, 2006.

Annexe A

Les Vérifications

Lorsque le concepteur a terminé de modéliser et de spécifier une partie où l'ensemble de son système sous Objecteering, il doit s'assurer qu'il n'a pas commis d'erreur de modélisation et qu'il a bien spécifié tous les composants de son système. L'ensemble des vérifications réalisées par Objecteering pour chacune des modélisations est présenté dans le listing suivant. Une explication sur le mode de lecture de l'arbre est donnée à la fin du listing.

Listing A.1 – Liste des vérifications effectuées dans Objecteering

```
 + Check A3S project
2 |
 +---+ Check CommEquipmentLib
 |   |
 |   +---+ Check HWComponent :
6 |       |   - vérification de son stéréotype erreur s'il n'hérite pas de
 |       |     CommEquipment ou de CommEquipment.
 |       |
 |       +---+ Check tagged value :
10|       |       - si on n'est pas en mode IHM avancé et si la tagged value
 |       |         est avancée, la vérification ne sera pas faite.
 |       |       - vérification si toutes les tagged value sont renseignées
 |       |         (i.e. leur valeur ne vaut pas "").
14|       |       - si c'est une valeur entière vérification qu'elle est bien
 |       |         entre un min (facultatif) et un max (facultatif).
 |       |       - si c'est une valeur énumérée vérification que sa valeur
 |       |         fait partie des valeurs énumérées.
18|       |       - si c'est une valeur booléenne vérification que sa valeur
 |       |         fait partie des valeurs true et false.
 |       |
 |       |
22|       +---+ Check Port :
 |           |   - vérification de son stéréotype.
 |           |   - vérification que ce HWComponent peut avoir ce type de
 |           |     port.
26|           |
 |           +---+ Check tagged value :
 |                   - même vérification que les tagged value d'un
 |                     HWComponent.
30|
 +---+ Check SWComponentLib
 |   |
 |   +---+ Check SWComponent :
34|       |   - vérification de son stéréotype.
 |       |
 |       +---+ Check tagged value :
 |       |       - même vérification que les tagged value d'un HWComponent.
38|       |
 |       +---+ Check SWIOPort :
```

```
|                - vérification de son stereotype.
|
|
+---+ Check HWPlatForms
|   |
|   +---+ Check HWPlatForm :
|   |       - vérification de son stéréotype.
|   |
|   +---+ Check Board :
|       |       - vérification de son stereotype (soit motherboard soit
|       |         daughterboard).
|       |
|       +---+ Check HWComponentInstance :
|       |   |   - vérification de son stereotype.
|       |   |   - vérification de l'instantiation du bon composant.
|       |   |   - vérification d'avoir le même nombre de ports que le
|       |   |     composant instancié.
|       |   |   - si c'est un digitalConnectorInstance vérification d'avoir
|       |   |     au moins un port d'entrée et un de sortie connectés.
|       |   |   - vérification que le datawidth d'un digitalPort soit égale
|       |   |     au datawidth du DigitalConnector connecté.
|       |   |
|       |   +---+ Check PortInstance :
|       |   |       - vérification de l'instantiation du bon port.
|       |   |       - vérification qu'un port est connecté à un
|       |   |         commEquipmentConnector analogique (resp. numérique)
|       |   |         s'il est analogique (resp. numerique)
|       |   |       - vérifier que les data width des ports connectés
|       |   |         les uns aux autres (soit à un a3s-
|       |   |         CommEquipmentConnector) soit cohérents (même data
|       |   |         width).
|       |   |
|       |   +---+ Check SWComponentInstance :
|       |       |   - vérification de son stéréotype.
|       |       |   - vérification de l'instantiation du bon composant.
|       |       |   - vérification d'avoir le même nombre de ports que le
|       |       |     composant instancié.
|       |       |
|       |       +---+ Check tagged value :
|       |       |       - même vérification que les tagged value d'un
|       |       |         HWComponent.
|       |       |       - vérifier que suivant l'implémentation choisie :
|       |       |          .FPGA : la somme du nombre de blocs ram des
|       |       |         fonctions implémentées sur un FPGA n'est pas
|       |       |         supérieure au nombre de blocs ram disponible sur
|       |       |         ce FPGA
|       |       |          .FPGA : idem pour le nombre de blocs multiplieur
|       |       |          .FPGA : idem pour le nombre de cellules logiques
|       |       |          .DSP : la somme des tailles de code compilé des
|       |       |         fonctions implémentées sur un même DSP ne doit
|       |       |         pas être supérieure à la taille mémoire programme
|       |       |         (ou à la dual memory size), dans le cas ou cette
|       |       |         somme est supérieure il faut s'assurer que le DSP
|       |       |         est relié à une mémoire d'un capacité suffisante.
|       |       |
|       |       +---+ Check SWIOPortInstance :
|       |           |   - vérification de son stéréotype.
|       |           |   - vérification de l'instantiation du bon port.
|       |           |
|       |           +---+ Check tagged value :
|       |                   - même vérification que les tagged value d'un
|       |                     HWComponent.
|       |                   - vérifier que les tagged value
|       |                     soient cohérentes (identique) à
|       |                     celles des ports du HWComponent
|       |                     sur lequel il est implémenté si ce
|       |                     port appartient à un SWComponent
|       |                     communiquant avec un SWComponent
```

```
|         |                       instancié sur un HWComponent
|         |                       différent.
|         |
|         +---+ Check Functional Bloc :
|               |       - vérification de son stéréotype
|               |
|               +---+ Check HWComponentInstance (parcours récursif)
|               |
|               +---+ Check "Functional Bloc" (parcours récursif)
|
+---+ Check SWApplications
      |
      +---+ Check SWApplication :
            |   - vérification de son stéréotype
            |
            +---+ Check ObjectCode :
            |   |   - vérification de son stéréotype
            |   |
            |   +---+ Check tagged value :
            |   |       - même vérification que les tagged value d'un
            |   |         HWComponent.
            |   |       - vérifier pour tout Objectcode Obj1 connecté à un
            |   |         autre ObjectCode Obj2 on est :  (iterate_number_Obj1
            |   |         * output_packet_size_transitionObj1->Obj2)
            |   |         = (iterate_number_Obj2 *
            |   |         input_packet_size_transitionObj1->Obj2)
            |   |         et le cas échéant si Obj1 et Obj2 appartiennent
            |   |         à 2 modules différent (iterate_number_Obj1 *
            |   |         output_packet_size_transitionObj1->Obj2)*
            |   |         module_Obj1_iterate_number = (iterate_number_Obj2 *
            |   |         input_packet_size_transitionObj1->Obj2)*
            |   |         module_Obj2_iterate_number
            |   |
            |   +---+ Check Transition :
            |       |   - vérification que la transition pointe vers un
            |       |     ObjectCode ou un FinalState.
            |       |   - vérification du même nombre de paquets en sortie
            |       |     et en entrée.
            |       |
            |       +---+ Check tagged value :
            |               - même vérification que les tagged value d'un
            |                 HWComponent.
            |
            +---+ Check Module :
            |   - vérification de son stéréotype.
            |
            +---+ Check tagged value :
            |       - même vérification que les tagged value d'un
            |         HWComponent.
            |
            +---+ Check ObjectCode (parcours récursif)
            |
            +---+ Check Module (parcours récursif)
```

La lecture de cet arbre se fait comme avec l'exemple suivant :
A la ligne 82, la somme des valeurs (représentée par un élement TaggedValue) des paramètres de taille de bloc ram des composants logiciels implantés sur le même composant de type FPGA, ne doit pas être supérieure à la valeur du paramètre de taille du bloc Ram de ce même composant FPGA. La hiérarchie des branches nous informe qu'il s'agit de vérifications effectuées après le mapping, car elle s'effectue sur un composant logiciel instancié sur une instance de composant matériel qui est implanté sur une carte dans une modélisation de plate-forme. Ce sens de lecture est le même pour chacune des vérifications exprimées dans ce listing.

Annexe B

Composition de la plate-forme Pentek

La plate-forme matérielle utilisée est issue de l'assemblage de différents modèle de carte pentek. Il s'agit des modèles

- **4290** : comprenant une architecture à base de 4 DSP TMS320C6203 de chez Texas et de mémoires de type SDRAM, SBSRAM, DPSRAM de capacités différentes,
- **6250** : comprenant une architecture à base de 2 FPGA Virtex II XC2V3000 de chez Xilinx et de mémoires de type SRAM,
- **6229** : correspondant à un module d'émission, comprenant le convertisseur numérique à fréquence ascendante ainsi que le convertisseur numérique/analogique utile pour l'émission de l'onde radio UMTS,
- **6216** : correspondant au récepteur large bande, comprenant les étages d'amplification avec également le convertisseur numérique à fréquence descendante ainsi que le convertisseur analogique/numérique utile pour la réception de l'onde radio UMTS.

Les datas sheet de chacun des modèles fournis par pentek sont incorporés ci-après.

Model 4290
Model 4291

Quad TMS320C6201 Processor - VME
Quad TMS320C6701 Processor - VME

General Information

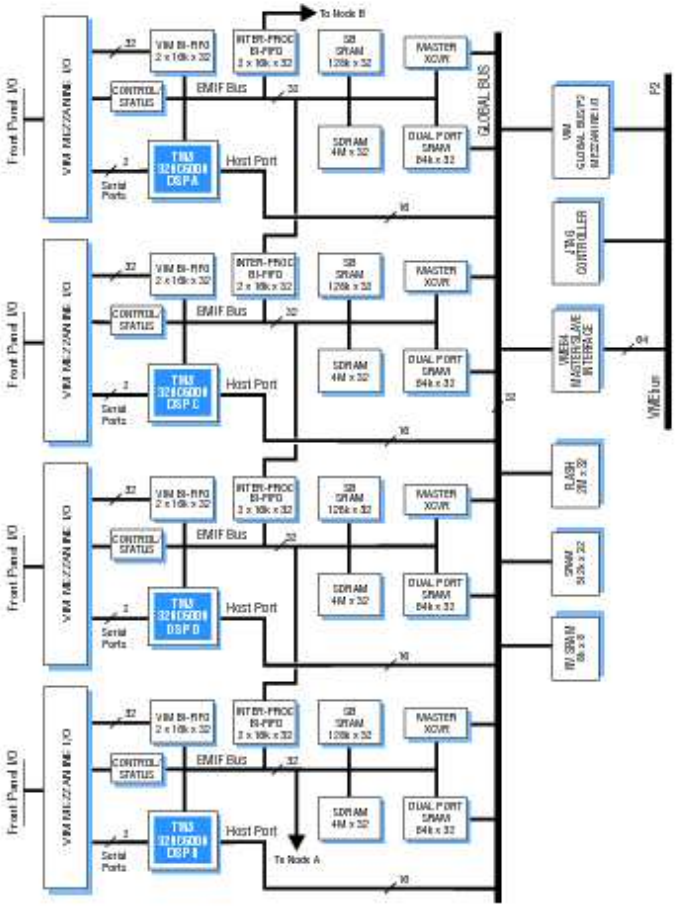
Model 4290: Model 4290 features Texas Instruments' TMS320C6201 fixed-point digital signal processor which employs the VelociTI ™ architecture to achieve a remarkable 1600 MIPS performance.

Incorporating four of these devices, the Model 4290 is a single-slot 6U VMEbus board delivering up to 6400 MIPS of processing power and a wealth of high-speed interface options to handle the 'C6201's voracious appetite for data.

Model 4291: Model 4291 features Texas Instruments' TMS320C6701 floating- point digital signal processor which achieves 1 GFLOPS performance.

The Model 4291 incorporates four of these DSP's in a single-slot 6U VMEbus board delivering up to 4 GFLOPS of processing power. A range of high-speed interface options perfectly compliments the C6701's impressive processing performance.

The boards are organized as four identical processing nodes, each equipped with several types of I/O and memory resources. This unique architecture has been optimized for handling the most demanding real-time signal processing applications in high-performance VMEbus systems. ➤



Features

- **Model 4290:** Four TMS320C6201 DSPs operating at 200 MHz
- 6400 MIPS performance
- **Model 4291:** Four TMS320C6701 DSPs operating at 167 MHz
- 4 GFLOPS performance
- **Both Models:** High performance VIM mezzanine module sites

PENTEK

Pentek, Inc. One Park Way • Upper Saddle River • New Jersey 07458
Tel: 201-818-5900 • Fax: 201-818-5904 • Email: info@pentek.com
www.pentek.com

FIG. B.1 – Data sheet du modèle de carte pentek 4290 1/2

Model 4290
Model 4291

Quad TMS320C6201 Processor - VME
Quad TMS320C6701 Processor - VME

Support Software

Pentek's **SwiftNet** supports a network of distributed VMEbus systems and allows the developer to run development tools on the host, while maintaining remote access to the VMEbus system.

Pentek's **ReadyFlow** Board Support Libraries reduce development time by providing C-language callable functions for hardware initialization, control and operation of board resources.

TI's **Code Composer Studio** provides a comprehensive set of tools for software development including an optimizing C Compiler, an interactive debugger, the DSP/BIOS operating system and an assortment of profiling and optimizing tools.

Ordering Information

Model	Description
4290	Quad C6201 Processor - VME
Options:	
-320	16 kB VIM and Interprocessor BI-FO
-330	400 MB/sec FIFO operation
Model	**Description**
4291	Quad C6701 Processor - VME
Options:	
-320	16 kB VIM and Interprocessor BI-FIFO
-330	333 MB/sec FIFO operation

Processor Node Memory

Each processor node features three major memory sections: the SB SRAM (synchronous burst SRAM), the SDRAM (synchronous DRAM) and the DP SRAM (dual port SRAM). These resources are all attached to the 32-bit External Memory Interface (EMIF) bus.

The SB SRAM is the fastest memory on the board delivering zero-wait state performance for maximum utilization of the C6000 data bus at 800 MB/sec for the Model 4290 or 667 MB/sec for the 4291. This 512 kB memory may be used to store critical data or program code which must be accessed frequently.

The SDRAM provides a large, fast 16 MB work space operating at transfer rates of 400 MB/sec for the Model 4290 and 333 MB/sec for the 4291.

The DP SRAM allows the C6000 to efficiently move data quickly in and out of a 256 kB region using one port while the other port enables direct memory-mapped access from the VMEbus, global bus expansion masters and by other C6000 processors.

VIM Mezzanine

Each processor is equipped with its own VIM (Velocity Interface Mezzanine) connector, providing three types of interfaces. A high-speed synchronous bidirectional FIFO (BI-FIFO) buffers 32-bit parallel data transfers between the mezzanine and the processor's expansion bus at rates up to 400 MB/sec for the 4290, and 333 MB/sec for the 4291. Two of the C6000's synchronous serial ports are also brought to the mezzanine connector. The EMIF bus of the C6000 provides memory-mapped control and status functions to the mezzanine circuitry.

Processor Node FIFOs

The EMIF bus of each processor node connects to three BI-FIFOs, one for VIM mezzanine I/O and two for interprocessor communication. Each BI-FIFO is organized internally as a pair of 1k x 32 FIFOs, (optionally 16k x 32), one FIFO for each direction.

The VIM BI-FIFO connects data streaming devices located on the mezzanine boards to the EMIF bus and supports C6000 data transfers up to 400 MB/sec. The BI-FIFO effectively decouples the mezzanine device data flow and allows the processor to efficiently move data in blocks for maximum processor utilization.

The interprocessor BI-FIFOs connect the EMIF bus of adjacent C6000's and are ideal for pipelined processing applications. Block transfers at rates up to 400 MB/sec for the 4290 and 333 MB/sec for the 4291minimize data movement overhead and maximize interprocessor communications speed.

Global Bus Resources

The global bus acts as the data path to shared slave resources including the four DP SRAMs, the global SRAM, the NV SRAM, the Boot/User flash, the Global Bus Expansion Port and the 16-bit Host Port of all four processors.

Global Bus Memory

The global bus is supported by 2 MB of fast SRAM accessible by all processors, the I/O interface and the VME64 interface. This shared resource facilitates data passing between global bus masters.

8 MB of FLASH memory provides non-volatile memory for factory boot code and user-defined parameters for self-booting application programs.

8 kB of fast non-volatile SRAM for storing and sharing critical parameters is accessible by all processors and the VMEbus.

VME64 Master Interface

All four processors can directly utilize the VME64 master interface to become full VMEbus masters. This allows the C6000's to move data or perform control and status transfers to virtually any VMEbus device. The VME interface supports all data widths up to 64 bits.

Specifications

Processor Node Resources: 4 total
Processor: Model 4290: TMS320C6201;
Model 4291: TMS320C6701
Processor Clock: Model 4290: 200 MHz
Model 4291: 167 MHz
SDRAM: 4M x 32, 1-wait state
SBSRAM: 128k x 32
DPSRAM: 64k x 32
Mezzanine: VIM site
Mezzanine and Interprocessor BI-FIFO: 2 x 1k x 32; 2 x 16k x 32 optional
Shared Global Resources:
Global SRAM: 512k x 32
FLASH: 2M x 32
NVSRAM: 8k x 8
VME64 Interface: master/slave
Mezzanine: VIM site, global bus/P2
RACEway Interface: optional
Size: standard 6U VMEbus board, single slot; board 160 mm (6.3 in.) x 233.5 mm (9.2 in.), panel 0.8 in. wide

PENTEK
Pentek, Inc. One Park Way • Upper Saddle River • New Jersey 07458
Tel: 201-818-5900 • Fax: 201-818-5904 • Email: info@pentek.com
www.pentek.com

FIG. B.2 – Data sheet du modèle de carte pentek 4290 2/2

Model 6250 — Configurable Logic FPGA with FPDP I/O - VIM-2

ReadyFlow Board Support Libraries

GateFlow

Features

- Supports high-performance custom signal processing
- One or two FPDP (Front Panel Data Port) interfaces
- Xilinx Virtex-II Series FPGAs
- Two gate densities available: 1000K & 3000K
- On-board auxiliary SRAM
- GateFlow FPGA Design Kit
- Factory-installed cores available

General Information

Model 6250 is a Configurable Logic FPGA VIM-2 module with FPDP I/O. It features up to two FPDP (Front Panel Data Port) interfaces, and two Xilinx Virtex II FPGAs (Field Programmable Gate Arrays).

The Model 6250 supports custom, high-performance signal processing and computing functions for any VIM-compatible processor board.

FPGA Devices

The Model 6250 may be equipped with two FPGAs from either Xilinx Virtex-II family: Model XC2V1000 (standard), or XC2V3000 (Option -300).

Each FPGA interfaces directly to all three sections of the associated VIM interface: both serial ports, the 32-bit BI-FIFO parallel port and the control/status port. The BI-FIFO buffers on the processor board allow the processors to move blocks of data efficiently to and from each FPGA.

The two FPGAs are interconnected with up to 68 (XC2V3000) programmable user I/O lines to support data and control passing between the two devices. An optional connector is available for FPGA front panel I/O (Option -102).

SRAM

Each FPGA is equipped with two 256k x 16 SRAMs. Each SRAM is connected to the FPGA with separate address and data buses so they can be used independently. The SRAMS can be used for storing data without consuming internal FPGA logic cells to implement RAM.

FPDP Interfaces

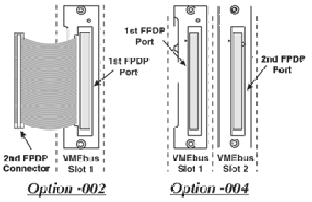
One FPDP interface is provided as standard with an additional interface available in two different optional configurations: single-slot, with the second connector accessible through a front panel ribbon cable (Option -002), or two slots with the second connector in an adjacent slot (Option -004).

FPGA Programming

An optional FPGA design kit to be used in conjunction with the Xilinx Foundation development tool suite is provided. It includes VHDL source files for the VIM interface and control registers for the clock functions. Templates for implementing custom signal processing blocks are included with detailed instructions. FPGA code may be downloaded from the processor node or permanently stored in non-volatile memory.

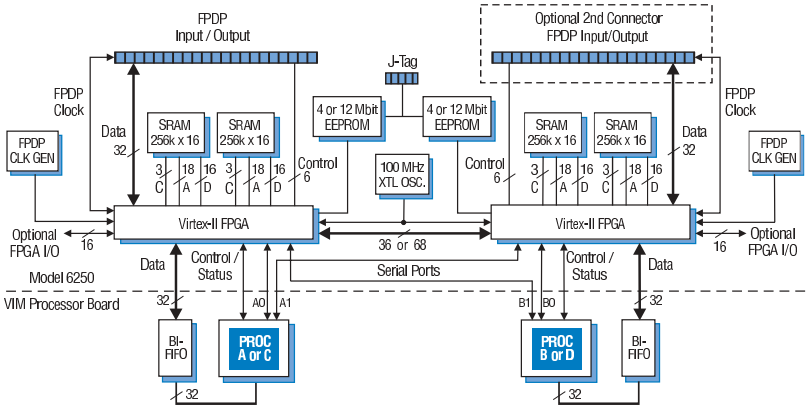
Numerous third-party sources for IP signal processing core libraries compatible with the Virtex-II support a wide range of popular algorithms and functions. These include FFTs, FIR filters, compression and decompression algorithms, software radio blocks, decryption, telemetry functions, decoders, encoders, and convolution.

In addition, the GateFlow FPGA Design Kit is optionally available from Pentek.



Ordering Information

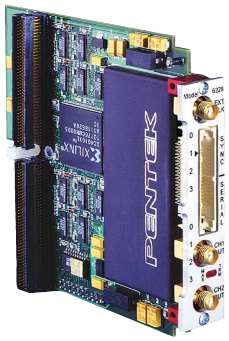
Model	Description
6250	Configurable Logic FPGA with FPDP I/O - VIM-2
Options:	
-002	Dual FPDP, plug and cable
-004	Dual FPDP, two slots
-102	Front panel FPGA I/O (additional slot)
-300	Xilinx XC2V3000 FPGAs
-404	4k-point quad FFT core installed



Pentek, Inc. One Park Way ◆ Upper Saddle River ◆ New Jersey 07458
Tel: 201-818-5900 ◆ Fax: 201-818-5904 ◆ Email: info@pentek.com
www.pentek.com

FIG. B.3 – Data sheet du modèle de carte pentek 6250

Model 6229 — Dual Digital Upconverter and D/A - VIM-2



General Information

Model 6229 is a VIM-2 module which attaches directly to VIM-compatible processor boards. The Model 6229 contains two complete channels of interpolation and frequency translation suitable for linking a DSP system to a radio transmitter.

Applications include generation of communication and radar test signals, electronic countermeasures, and implementation of transmit functions for advanced software radio communications systems.

Digital Upconverter

Both channels use the AD9856 Quadrature Digital Upconverter which includes half-band and CIC interpolation filters, a programmable local oscillator, a complex mixer, and a 12-bit D/A converter.

An on-chip multiplier accepts a reference clock from 5 to 50 MHz and multiplies it to a maximum 200 MHz internal sampling clock.

The reference clock can be driven from a local 50 MHz crystal oscillator or from an external input reference. A front panel ribbon cable allows multiple slave 6229's to be driven from a single designated master 6229.

Complex baseband data samples consisting of 12-bit I+Q pairs are accepted at rates as high as 25 Msamples/sec. The baseband signal must be band-limited to 45% of the input sampling rate, or approximately 11.25 MHz for 25 MHz input rate.

Three halfband interpolation filters upsample the baseband input by either 4x or 8x. Additional upsampling is performed by a CIC filter which interpolates by x2 to x63 in steps of 1, providing an overall interpolation range from x8 to x504.

Complex local oscillator samples are generated by a direct digital frequency synthesizer, programmed with 32-bit resolution to cover a LO range from DC to 80 MHz.

The translated single sideband signal, now sampled at 200 MHz, is fed into an on-chip 12-bit D/A converter, capable of producing analog output signals anywhere up to 80 MHz. An optional inverse sinx/x filter can be inserted in the signal path prior to the D/A to compensate for its zero-order hold frequency response.

Oversampled D/A Converter

For applications requiring a non-translated baseband D/A output, samples from the VIM processor board can be upsampled with no frequency translation and then sent to the D/A converter. This minimizes the complexity of an output smoothing filter.

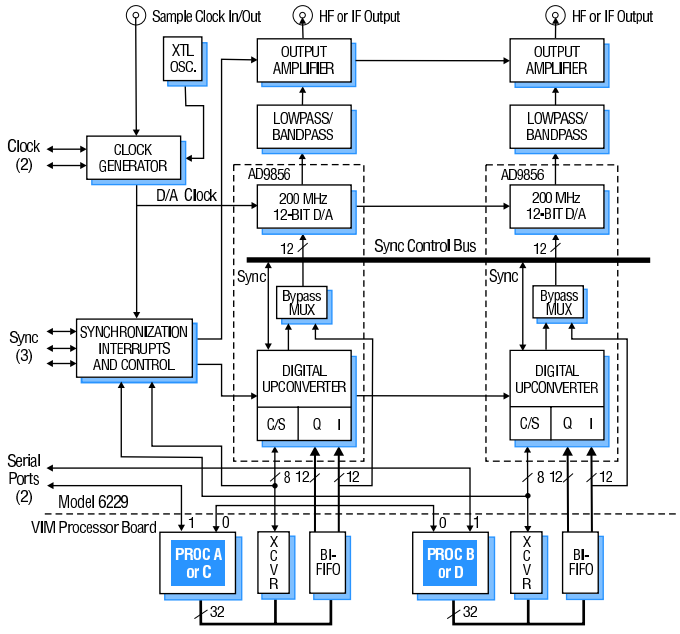
VIM Interface

Complex baseband samples generated by the baseboard processor are sent into the 6229 through the 32-bit parallel BI-FIFO path. Data may be packed with both I and Q components in a single 32-bit word for more efficient transfers.

ReadyFlow Board Support Libraries

Features

- VIM-2 module for VIM-compatible processor boards
- Translates digital signals to HF or IF frequencies, as high as 80 MHz
- Provides two identical, independent channels
- Interpolation filter, complex digital mixer, and programmable digital LO
- 12-bit 200 MHz D/A covers DC to 80 MHz output range
- Lowpass or bandpass output filters for baseband or IF analog outputs
- Synthesized signal generator mode
- Synchronization across multiple modules



Ordering Information

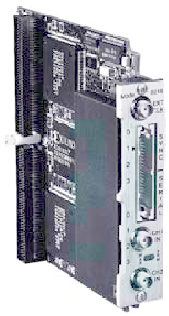
Model	Description
6229	Dual Digital Upconverter and D/A - VIM-2

PENTEK

Pentek, Inc. One Park Way ◆ Upper Saddle River ◆ New Jersey 07458 www.pentek.com
Tel: 201·818·5900 ◆ Fax: 201·818·5904 ◆ Email: info@pentek.com

FIG. B.4 – Data sheet du modèle de carte pentek 6229

Model 6216 — Dual Wideband Receiver and A/D VIM-2 Module



Two VIM-2 modules may be attached to VIM-compatible processor boards.

ReadyFlow Board Support Libraries

Features

- VIM-2 module for VIM-compatible processor boards
- Two identical channels include amplifier, A/D and digital receiver
- Up to 65 MHz A/D sampling with 12-bit accuracy
- Programmable-gain amplifiers and anti-aliasing filters
- Decimation range from 2 to 64 for output bandwidths up to 25 MHz
- Serial port interprocessor communication bus
- Synchronization across channels and other 6216's

Ordering Information

Model	Description
6216	Dual Wideband Receiver and A/D VIM-2 module

General Information

Model 6216 is a VIM-2 module which attaches directly to VIM-compatible processor boards. It features two complete channels of signal processing, ideal for HF software radio applications.

Two Model 6216's may be attached to a VIM-compatible processor board to form a 4-channel software radio which utilizes all four processors while occupying only one VMEbus slot. Alternatively, the Model 6216 may be combined with another VIM-2 module to provide additional I/O functions.

Input Section

Each channel includes a wideband input amplifier followed by a programmable gain amplifier for HF analog inputs with bandwidths up to 30 MHz. Analog inputs are accepted through front panel SMA connectors.

An anti-aliasing filter removes out-of-band frequency components and can be tailored for specific signal types. The standard filter has a cutoff frequency of 25 MHz.

The programmable-gain amplifier and filter may be bypassed to support undersampling applications.

A/D Converters

Each channel employs a 12-bit A/D converter capable of operating at up to 65 MHz sampling. The A/D sample clock is derived either from an external reference supplied to a front panel SMA connector or an internal 64 MHz crystal oscillator. The converters are Analog Devices type AD6640.

Both A/D converters operate synchronously from the same sampling clock to support multichannel applications, such as in direction finding, where phase between channels must be maintained.

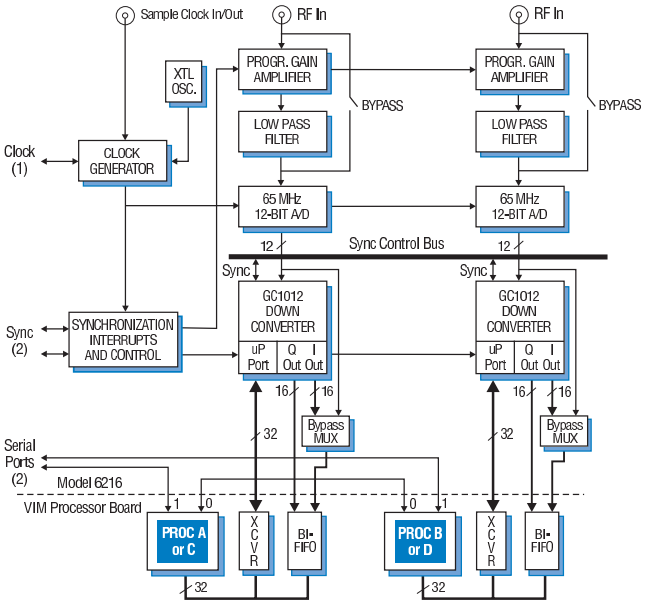
Digital Downconverters

The digitized output of each A/D converter feeds the Graychip GC1012 programmable downconverter. This device is designed for wideband output operation with decimation values ranging from 2 to 64 for output bandwidths as high as 25 MHz.

The output section delivers direct I and Q complex outputs to the mezzanine FIFO of the processor board. A bypass MUX provides a direct path from the A/D converter output directly into the FIFO buffer for direct capture of input data at rates up to 65 MHz.

A front panel ribbon cable bus allows multiple 6216's to share a common sample clock and synchronize the phase of digital receivers across modules.

Block Diagram, Model 6216



PENTEK
Pentek, Inc., One Park Way ◆ Upper Saddle River, NJ 07458 ◆ Tel 201-818-5900
Fax 201-818-5904 ◆ Email: info@pentek.com
www.pentek.com

FIG. B.5 – Data sheet du modèle de carte pentek 6216

Annexe C

Lois d'estimation de l'*overhead* d'un système d'exploitation

La campagne de mesure des temps d'exécution des tâches exécutées sur μcos a permis d'extraire des paramètres du système d'exploitation qui influent sur ce temps d'exécution. Les lois présentées ci-après, sont issues de l'analyse des multiples mesures relevées dans différentes conditions d'utilisation du système d'exploitation. Les impacts des différents services et des particularités de fonctionnement du système d'exploitation sont donc présentés ci-dessous. Les valeurs des paramètres sont celles obtenues avec l'exécution de μcos sur le processeur Microblaze implanté sur un Virtex-II pro (XC2VP30) de Xilinx.

C.0.1 Effet de l'ordonnanceur sur le temps d'exécution

Après avoir relevé les différents temps d'exécution de plusieurs tâches on détermine que l'*overhead* lié à l'ordonnancement est de la forme :

$$T_{ps_{exeavecos}} = T_{ps_{exesansos}} + (\gamma * \left\lfloor \frac{T_{ps_{exesansos}}}{P_{tickordo}} \right\rfloor) \tag{C.1}$$

d'où

$$O_{ordo} = \gamma * \left\lfloor \frac{T_{ps_{exesansos}}}{P_{tickordo}} \right\rfloor \tag{C.2}$$

avec

- γ correspondant au nombre de ticks d'horloge d'un overhead. Il est dépendant du nombre de tâches crées $Nb_{tâche}$. En effet la table des tâches de l'ordonnanceur est dimensionnée en fonction de ce nombre et met un certain temps à la scruter (ν lorsqu'il y a qu'une seule tâche). D'où $\gamma = \nu + ((Nb_{tâche} - 2) * \rho))$ avec $\nu = 1491$ et $\rho = 111$. ρ est le surcoût temporel ajouté par tâche créée supplémentaire. La soustraction par 2 tient compte de la tâche minimum à avoir à laquelle s'ajoute la tâche Idle par défaut (soit 2 tâches minimum pour un coût de ν).

Si on intègre les relevés obtenus en observant l'effet de la période de l'ordonnanceur sur la même tâche il apparaît que la formule appropriée se précise tel que

$$O_{ordo} = \kappa + \left\lfloor \frac{\kappa}{P_{tickordo}} \right\rfloor * (\gamma + 4) \qquad \text{avec} \qquad \kappa = \gamma * \left\lfloor \frac{T_{ps_{exesansos}}}{P_{tickordo}} \right\rfloor + 4 * \left(\left\lfloor \frac{T_{ps_{exesansos}}}{P_{tickordo}} \right\rfloor - 1 \right) \tag{C.3}$$

Cette équation s'explique car l'overhead correspond au temps d'exécution sans os plus le nombre d'interruption du à l'ordonnancement fois le temps que prend cet interruption (γ). Ce nombre dépend du rapport $\left\lfloor \frac{T_{ps_{exesansos}}}{P_{tickordo}} \right\rfloor$ qui détermine le nombre d'interruption. Il se peut que l'overhead ajouté, rajoute du temps qui provoque des interventions supplémentaires de l'ordonnanceur .

D'où l'*overhead* suivant :

$$O_{ordo} = \kappa + \left\lfloor \frac{\kappa}{P_{tickordo}} \right\rfloor * (\gamma + 4) \tag{C.4}$$

C.0.2 Effet de la création des tâches sur le temps d'exécution

$$O_{créationtâche} = \left[\left[\alpha + (s_{tss} * \beta) \right] * Nb_{tâchecréées} \right] + \mu \tag{C.5}$$

avec

- α : nombre de ticks s'il n'y avait pas de pile associée à chaque tâche
- s_{tss} : profondeur de la pile (commune à toutes les tâches)
- β : nombre de ticks supplémentaire imputé à l'augmentation de la profondeur de la pile d'une unité
- μ : constante du au mécanisme même de création de tâche.

ici α=2101 β=92 μ=56

D'où l'*overhead* suivant :

$$O_{créationtâche} = \left[\left[\alpha + (s_{tss} * \beta) \right] * Nb_{tachecrees} \right] + \mu \tag{C.6}$$

C.0.3 Effet de la création de sémaphores sur le temps d'exécution

Si tous les sémaphores sont créés à la suite les uns des autres :

$$O_{créationsuitesémaphore} = Tps_{csi} + (n-1)Tps_{csc} \tag{C.7}$$

Si tous les sémaphores sont créés indépendamment dans le code :

$$O_{créationsémaphoresindépendant} = n * Tps_{csi} \tag{C.8}$$

ici Tps_{csi}=382 Tps_{csc}=330

avec

- n : le nombre de sémaphores créés
- Tps_{csi} : temps d'exécution de la création d'un sémaphore
- Tps_{csc} : temps d'exécution de la création d'un sémaphore lorsque l'instruction d'avant était également la création d'un sémaphore

D'où l'*overhead* suivant :

$$Tps_{csi} + (n-1)Tps_{csc} \leq O_{créationesémaphore} \leq n * Tps_{csi} \quad \text{(C.9)}$$

C.0.4 Effet des pends de sémaphores sur le temps d'exécution

$$O_{pendsémaphore} = n * Tps_{cspei} + k * Tps_{cspec} \quad \text{(C.10)}$$

Si tous les pends de sémaphores sont appélés indépendamment dans le code :

$$O_{pendsémaphoregéné} = z * Tps_{cspec} \quad \text{(C.11)}$$

avec

- z : le nombre de pend appelé
- n : le nombre de pend appelé une seule fois par tâche
- k : le nombre de pend appelé plusieurs fois par tâche
- Tps_{cspei} : temps d'exécution d'un pend dans un sémaphore
- Tps_{cspec} : temps d'exécution d'un pend lorsque plusieurs pend de sémaphore sont appelés dans la même tâche

ici Tps_{cspei}=297 et Tps_{cspei}=293

D'où l'*overhead* suivant :

$$z * Tps_{cspec} \leq O_{pendsemaphoregéné} \leq z * Tps_{cspei} \quad \text{(C.12)}$$

C.0.5 Effet des posts de sémaphores sur le temps d'exécution

Si tous les posts de sémaphores sont appelés à la suite les uns des autres :

$$O_{postsémaphore} = Tps_{cspoi} + (n-1)Tps_{cspoc} \quad \text{(C.13)}$$

Si tous les posts de sémaphores sont créés indépendamment dans le code :

$$O_{postsémaphoreindépendant} = n * Tps_{cspoi} \quad \text{(C.14)}$$

avec

- n : le nombre de sémaphores créés
- Tps_{cspoi} : temps d'exécution d'un post d'un sémaphore
- Tps_{cspoc} : temps d'exécution d'un post d'un sémaphore lorsque l'instruction d'avant était également le post d'un sémaphore

ici Tps_{cspoi}=244 Tps_{cspoc}=196

D'où l'*overhead* suivant :

$$Tps_{cspoi} + (n-1)Tps_{cspoc} \leq O_{postsémaphore} \leq n * Tps_{cspoi} \quad \text{(C.15)}$$

C.0.6 Effet de la création de mutex sur le temps d'exécution

Plusieurs cas de figures peuvent de produire en fonction du codage de l'application. Soit

	ils peuvent tous avoir la même priorité	ils peuvent avoir des priorités différentes
Tous les mutex créés le sont les uns à la suite des autres	×	×
Tous les mutex créés le sont séparément au fil de l'application	×	×

Si tous les mutex sont créés à la suite les uns des autres :

- avec des priorités différentes :

$$O_{créationsuitemutexpriodif} = Tps_{cmidif} + (n-1)Tps_{cmcpriodif} \quad \text{(C.16)}$$

- avec des priorités identiques :

$$O_{créationsuitemutexprioidem} = Tps_{cmidif} + (n-1)Tps_{cmcprioidem} \quad \text{(C.17)}$$

avec

- n : le nombre de mutex créés
- Tps_{cmidif} : temps d'exécution de la création d'un mutex avec des priorités différentes des autres mutex
- $Tps_{cmiidem}$: temps d'exécution de la création d'un mutex avec des priorités identiques des autres mutex
- $Tps_{cmcpriodif}$: temps d'exécution de la création d'un mutex lorsque l'instruction d'avant était également la création d'un mutex de priorité différente
- $Tps_{cmcprioidem}$: temps d'exécution de la création d'un mutex lorsque l'instruction d'avant était également la création d'un mutex de priorité identique

Si tous les mutex sont créés indépendamment dans le code :

- avec des priorités différentes :

$$O_{créationmutexindépendant} = n * Tps_{cmidif} \quad \text{(C.18)}$$

- avec des priorités identiques :

$$O_{création_mutex_indépendantid} = Tps_{cmidif} + (n-1)Tps_{cmiidem} \quad \text{(C.19)}$$

ici Tps_{cmidif}=474 $Tps_{cmiidem}$=312 $Tps_{cmcpriodif}$=422 $Tps_{cmcprioidem}$=260

D'où l'*overhead* suivant :

$$Tps_{cmidif} + (n-1)Tps_{cmcprioidem} \quad \leq \quad O_{créationemutexgéné} \quad \leq \quad n * Tps_{cmidif} \tag{C.20}$$

C.0.7 Effet des pends de mutex sur le temps d'exécution

$$O_{pendmutex} = n * Tps_{cmpei} \tag{C.21}$$

avec

- n : le nombre de pend appelé
- Tps_{cmpei} : temps d'exécution d'un pend dans un mutex

ici Tps_{cmpei}=347

D'où l'*overhead* suivant :

$$O_{pendmutexgéné} = n * Tps_{cmpei} \tag{C.22}$$

C.0.8 Effet des posts de mutex sur le temps d'exécution

Les mutex correspondent à une forme limitée de sémaphores, les sémaphores binaires. De ce faite ils n'y a pas d'intérêt à lancer plusieurs posts à la suite.

$$O_{postmutexindépendant} = n * Tps_{cmpoi} \tag{C.23}$$

avec

- n : le nombre de mutex créés
- Tps_{cmpoi} : temps d'exécution d'un post d'un mutex

ici Tps_{cmpoi}=513

D'où l'*overhead* suivant :

$$O_{postmutexgéné} = n * Tps_{cmpoi} \tag{C.24}$$

C.0.9 Effet de la création des flags sur le temps d'exécution

Si tous les flags sont créés à la suite les uns des autres :

$$O_{créationsuiteflags} = Tps_{cfi} + (n-1)Tps_{cfc} \tag{C.25}$$

Si tous les flags sont créés indépendamment dans le code :

$$O_{création_flag_indépendant} = n * Tps_{cfi} \tag{C.26}$$

ici Tps_{cfi}=325 Tps_{cfc}=273

avec

- n : le nombre de flags créés
- Tps_{cfi} : temps d'exécution de la création d'un flag
- Tps_{cfc} : temps d'exécution de la création d'un flag lorsque l'instruction d'avant était également la création d'un flag

D'où l'*overhead* suivant :

$$Tps_{cfi} + (n-1)Tps_{cfic} \quad \leq \quad O_{création\,flaggéné} \quad \leq \quad n * Tps_{cfi} \qquad \text{(C.27)}$$

C.0.10 Effet des pends de flag sur le temps d'exécution

Si tous les pends de flag sont appelés à la suite les uns des autres :

$$O_{pend\,flag} = Tps_{cfpei} + (n-1)Tps_{cfpec} \qquad \text{(C.28)}$$

Si tous les pends de flag sont appelés indépendamment dans le code :

$$O_{pend\,flagindépendant} = n * Tps_{cfpei} \qquad \text{(C.29)}$$

avec

- n : le nombre de flags créés
- Tps_{cfpei} : temps d'exécution d'un pend d'un flag
- Tps_{cfpec} : temps d'exécution d'un pend d'un flag lorsque l'instruction d'avant était également le pend d'un flag

ici Tps_{cfpei}=445 Tps_{cfpec}=393

D'où l'*overhead* suivant :

$$Tps_{cfpei} + (n-1)Tps_{cfpec} \quad \leq \quad O_{pend\,flaggéné} \quad \leq \quad n * Tps_{cfpei} \qquad \text{(C.30)}$$

C.0.11 Effet des posts de flag sur le temps d'exécution

Si tous les posts de flag sont appelés à la suite les uns des autres :

$$O_{post\,flag} = Tps_{cfpoi} + (n-1)Tps_{cfpoc} \qquad \text{(C.31)}$$

Si tous les posts de flag sont appelés indépendamment dans le code :

$$O_{post\,flagindépendant} = n * Tps_{cfpoi} \qquad \text{(C.32)}$$

avec

- n : le nombre de flags créés
- Tps_{cfpoi} : temps d'exécution d'un post d'un flag
- Tps_{cfpoc} : temps d'exécution d'un post d'un flag lorsque l'instruction d'avant était également le post d'un flag

ici Tps_{cfpoi}=411 Tps_{cfpoc}=359

D'où l'*overhead* suivant :

$$Tps_{cfpoi} + (n-1)Tps_{cfpoc} \quad \leq \quad O_{post\,flaggéné} \quad \leq \quad n * Tps_{cfpoi} \qquad \text{(C.33)}$$

C.0.12 Effet de la création des mailbox sur le temps d'exécution

Si toutes les mailbox sont créées les unes à la suite des autres :

$$O_{créationsuitemailbox} = Tps_{cmbxi} + (n-1)Tps_{cmbxc} \quad \text{(C.34)}$$

Si toutes les mailbox sont créées indépendamment dans le code :

$$O_{création_mailbox_indépendant} = n * Tps_{cmbxi} \quad \text{(C.35)}$$

ici Tps_{cmbxi}=378 Tps_{cmbxc}=326

avec

- n : le nombre de mailbox créés
- Tps_{cmbxi} : temps d'exécution de la création d'une mailbox
- Tps_{cmbxc} : temps d'exécution de la création d'une mailbox lorsque l'instruction d'avant était également la création d'une mailbox

D'où l'*overhead* suivant :

$$Tps_{cmbxi} + (n-1)Tps_{cmbxc} \quad \leq \quad O_{créationmailboxgéné} \quad \leq \quad n * Tps_{cmbxi} \quad \text{(C.36)}$$

C.0.13 Effet des pends de mailbox sur le temps d'exécution

Si tous les pends de mailbox sont appelés à la suite les uns des autres :

$$O_{pendmailbox} = Tps_{cmbxpei} + (n-1)Tps_{cmbxpec} \quad \text{(C.37)}$$

Si tous les pends de mailbox sont appelés indépendamment dans le code :

$$O_{pendmailboxindépendant} = n * Tps_{cmbxpei} \quad \text{(C.38)}$$

avec

- n : le nombre de mailbox créés
- $Tps_{cmbxpei}$: temps d'exécution d'un pend d'un mailbox
- $Tps_{cmbxpec}$: temps d'exécution d'un pend d'un mailbox lorsque l'instruction d'avant était également le pend d'un mailbox

ici $Tps_{cmbxpei}$=308 $Tps_{cmbxpec}$=256

D'où l'*overhead* suivant :

$$Tps_{cmbxpei} + (n-1)Tps_{cmbxpec} \quad \leq \quad O_{pendmailboxgéné} \quad \leq \quad n * Tps_{cmbxpei} \quad \text{(C.39)}$$

C.0.14 Effet des posts de mailbox sur le temps d'exécution

Si tous les posts de mailbox sont appelés à la suite les uns des autres :

$$O_{postmailbox} = Tps_{cmbxpoi} + (n-1)Tps_{cmbxpoc} \quad \text{(C.40)}$$

Si tous les posts de mailbox sont appelés indépendamment dans le code :

$$O_{postmailboxindépendant} = n * Tps_{cfpoi} \quad \text{(C.41)}$$

avec

- n : le nombre de mailbox créés
- $Tps_{cmbxpoi}$: temps d'exécution d'un post d'un mailbox
- $Tps_{cmbxpoc}$: temps d'exécution d'un post d'un mailbox lorsque l'instruction d'avant était également le post d'un mailbox

ici $Tps_{cmbxpoi}$=255 $Tps_{cmbxpoc}$=203

D'où l'*overhead* suivant :

$$Tps_{cmbxpoi} + (n-1)Tps_{cmbxpoc} \leq O_{postmailboxgéné} \leq n * Tps_{cmbxpoi} \quad \text{(C.42)}$$

C.0.15 Effet des postOpt de mailbox sur le temps d'exécution

Si tous les postOpt de mailbox sont appelés à la suite les uns des autres :

$$O_{postOptmailbox} = Tps_{cmbxpoOpti} + (n-1)Tps_{cmbxpoOptc} \quad \text{(C.43)}$$

Si tous les postOpts de mailbox appelés indépendamment dans le code :

$$O_{postOptmailboxindépendant} = n * Tps_{cfpoOpti} \quad \text{(C.44)}$$

avec

- n : le nombre de mailbox créés
- $Tps_{cmbxpoOpti}$: temps d'exécution d'un postOpt d'un mailbox
- $Tps_{cmbxpoOptic}$: temps d'exécution d'un postOpt d'un mailbox lorsque l'instruction d'avant était également le postOpt d'un mailbox

ici $Tps_{cmbxpoOpti}$=278 $Tps_{cmbxpoOptc}$=226

D'où l'*overhead* suivant :

$$Tps_{cmbxpoOpti} + (n-1)Tps_{cmbxpoOptc} \leq O_{postOptmailboxgéné} \leq n * Tps_{cmbxpoOpti} \quad \text{(C.45)}$$

C.0.16 Effet de la création des messages queue sur le temps d'exécution

Si toutes les messages queue sont créées les unes à la suite des autres :

$$O_{créationsuitemessagesqueue} = Tps_{cmsgqi} + (n-1)Tps_{cmsgqc} \quad \text{(C.46)}$$

Si toutes les messages queue sont créées indépendamment dans le code :

$$O_{création_messagesqueue_indépendant} = n * Tps_{cmsgqi} \quad \text{(C.47)}$$

ici Tps_{cmsgqi}=580 Tps_{cmsgqc}=528

avec

- n : le nombre de messages queue créés
- Tps_{cmsgqi} : temps d'exécution de la création d'une messages queue
- Tps_{cmsgqc} : temps d'exécution de la création d'une messages queue lorsque l'instruction d'avant était également la création d'une messages queue

D'où l'*overhead* suivant :

$$Tps_{cmsgqi} + (n-1)Tps_{cmsgqc} \quad \leq \quad O_{créationmessagequeuegéné} \quad \leq \quad n * Tps_{cmsgqi} \quad \text{(C.48)}$$

C.0.17 Effet des pends de message queue sur le temps d'exécution

Si tous les pends de message queue sont appelés à la suite les uns des autres :

$$O_{pendmessagequeue} = Tps_{cmsgqpei} + (n-1)Tps_{cmsgqpec} \quad \text{(C.49)}$$

Si tous les pends de message queues sont appelés indépendamment dans le code :

$$O_{pendmessagequeueindépendant} = n * Tps_{cmsgqpei} \quad \text{(C.50)}$$

avec

- n : le nombre de pend appelé
- $Tps_{cmsgqpei}$: temps d'exécution d'un pend d'une message queue
- $Tps_{cmsgqpec}$: temps d'exécution d'un pend d'une message queue lorsque l'instruction d'avant était également le pend d'un message queue

ici $Tps_{cmsgqpei}$=352 $Tps_{cmsgqpec}$=300

D'où l'*overhead* suivant :

$$Tps_{cmsgqpei} + (n-1)Tps_{cmsgqpec} \quad \leq \quad O_{pendmessagequeuegéné} \quad \leq \quad n * Tps_{cmsgqpei} \quad \text{(C.51)}$$

C.0.18 Effet des posts de message queue sur le temps d'exécution

Il y a trois sortes de posts possibles pour les message queue. Au fil des versions de ucos de nouvelles possibilités de posts sont apparues (présentées ci-après). Le post simple correspond à la mise en place des messages dans une FIFO.

$$O_{postmessagequeue} = Tps_{cmsgqpoi} + (n-1)Tps_{cmsgqpoc} \tag{C.52}$$

Si tous les posts de message queues sont appelés indépendamment dans le code :

$$O_{postmessagequeueindépendant} = n * Tps_{cmsgqpoi} \tag{C.53}$$

avec
- n : le nombre de message queue créés
- $Tps_{cmsgqpoi}$: temps d'exécution d'un post d'une message queue
- $Tps_{cmsgqpoc}$: temps d'exécution d'un post d'une message queue à la suite d'une autre

ici $Tps_{cmsgqpoi}$=329 $Tps_{cmsgqpoc}$=277

D'où l'*overhead* suivant :

$$Tps_{cmsgqpoi} + (n-1)Tps_{cmsgqpoc} \leq O_{postmessagequeuegéné} \leq n * Tps_{cmsgqpoi} \tag{C.54}$$

C.0.19 Effet des postFront de message queue sur le temps d'exécution

Le postFront est apparu un peu plus tard. Il est similaire au post précédemment décrit à ceci près qu'il correspond la mise en place des messages dans une LIFO. Ceci entraîne un mécanisme qui fait que la première exécution d'un postFront concernant une file de message particulière prendra plus de temps que le second postFront sur cette même file de message. Ceci s'explique par le fait que le pointeur désignant le prochain message à sortir doit se positionner au même endroit que le pointeur du prochain message à entrer. Ceci se fait lors du premier postFront, car ensuite il est déjà positionné.

Nous avons alors 4 possibilitées :

	ils sont appelés indépendamment	ils sont appelés à la suite
Tous les postFront concernent les mêmes files de messages	×	×
Tous les postFront concernent des files de messages différentes	×	×

Si tous les postFront sont appelés les uns à la suite des autres :
- ils concernent les mêmes files de messages :

$$O_{suitepostFrontmessagequeueidem} = Tps_{cmsgqpofidif} + (n-1)Tps_{cmsgqpofcidem} \tag{C.55}$$

• ils concernent des files de messages différentes :

$$O_{suitepostFrontmessagequeueidem} = Tps_{cmsgqpofidif} + (n-1)Tps_{cmsgqpofcdif} \quad \text{(C.56)}$$

Si tous les postFront sont appelés indépendamment les uns à la suite des autres :

• ils concernent les mêmes files de messages :

$$O_{suitepostFrontmessagequeueidptidem} = Tps_{cmsgqpofidif} + (n-1)Tps_{cmsgqpofiidem} \quad \text{(C.57)}$$

• ils concernent des files de messages différentes :

$$O_{suitepostFrontmessagequeueidptidem} = n * Tps_{cmsgqpofidif} \quad \text{(C.58)}$$

avec

- n : le nombre de postFront de message queue appelé
- $Tps_{cmsgqpofidif}$: temps d'exécution d'appel d'un postFront pour une file de messages (1ère appel)
- $Tps_{cmsgqpofcidem}$: temps d'exécution d'appel d'un postFront pour une file de messages déjà appelée dont l'appel succède à un autre appel de postFront
- $Tps_{cmsgqpofcdif}$: temps d'exécution d'appel d'un postFront pour une file de messages différentes dont l'appel succède à un autre appel de postFront
- $Tps_{cmsgqpofiidem}$: temps d'exécution d'appel indépendant de postFront pour des files de messages identiques

ici $Tps_{cmsgqpofidif}$=348 $Tps_{cmsgqpofcidem}$=337 $Tps_{cmsgqpofcdif}$=296 $Tps_{cmsgqpofiidem}$=285

D'où l'*overhead* suivant :

$$Tps_{cmsgqpofidif} + (n-1)Tps_{cmsgqpoficidem} \leq O_{postFrontmessagequeuegéné} \leq n * Tps_{cmsgqpofidif} \quad \text{(C.59)}$$

C.0.20 Effet des postOpt de message queue sur le temps d'exécution

Le postOpt est le dernier des posts à être intégré dans ucos. Il intègre les possibilités des deux précédents avec en plus la possibilité d'envoyer un message à toutes les tâches. Il permet donc d'avoir les mécanismes de FIFO, LIFO et de messages postés pour tous. Il possède un argument supplémentaire qui permet de spécifier le fonctionnement souhaité : opt. Cet argument prend les valeurs : OS_POST_OPT_NONE, OS_POST_OPT_BROADCAST, OS_POST_OPT_FRONT et OS_POST_OPT_FRONT + OS_POST_OPT_BROADCAST.

opt = OS_POST_OPT_NONE et opt = OS_POST_OPT_BROADCAST

Ces deux opérations fournissent les mêmes résultats au niveau des temps d'exécution. avec

- L'opération OS_POST_OPT_NONE correspond à un fonctionnement comme un simple post.
- L'opération OS_POST_OPT_BROADCAST permet de poster un message à toutes les tâches. correspond à un fonctionnement comme un simple post.

$$O_{postOptNBmessagequeue} = Tps_{cmsgqpoOptNBi} + (n-1)Tps_{cmsgqpoOptNBc} \quad \text{(C.60)}$$

Si tous les postOpt avec opt = OS_POST_OPT_NONE et opt = OS_POST_OPT_BROADCAST sont appelés indépendamment dans le code :

$$O_{postOptNBmessagequeueindependant} = n * Tps_{cmsgqpoOptNBi} \quad \text{(C.61)}$$

avec
- n : le nombre de postOpt de message queue appelé
- $Tps_{cmsgqpoOptNBi}$: temps d'exécution d'un postOpt avec les opérations OS_POST_OPT_BROADCAST et OS_POST_OPT_NONE d'une message queue
- $Tps_{cmsgqpoOptNBc}$: temps d'exécution d'un postOpt avec les opérations OS_POST_OPT_BROADCAST et OS_POST_OPT_NONE d'une message queue à la suite d'une autre

ici $Tps_{cmsgqpoOptNBi}$=364 $Tps_{cmsgqpoOptNBc}$=312

opt = OS_POST_OPT_FRONT

Cet argument permet comme pour le service OSQPostFront d'avoir une LIFO. On retrouve donc les 4 mêmes configurations possibles transposées au OSQPostOpt.

Si tous les postOpt (Front) sont appelés les uns à la suite des autres :

- ils concernent les mêmes files de messages :

$$O_{suitepostOptFmessagequeueidem} = Tps_{cmsgqpoOptFidif} + (n-1)Tps_{cmsgqpoOptFcidem} \quad \text{(C.62)}$$

- ils concernent des files de messages différentes :

$$O_{suitepostOptFmessagequeueidem} = Tps_{cmsgqpoOptFidif} + (n-1)Tps_{cmsgqpoOptFcdif} \quad \text{(C.63)}$$

Si tous les postOpt (Front) sont appelés indépendamment les uns des autres :

- ils concernent les mêmes files de messages :

$$O_{suitepostOptFmessagequeueidptidem} = Tps_{cmsgqpoOptFidif} + (n-1)Tps_{cmsgqpoOptFiidem} \quad \text{(C.64)}$$

- ils concernent des files de messages différentes :

$$O_{suitepostOptFmessagequeueidptidem} = n * Tps_{cmsgqpoOptFidif} \quad \text{(C.65)}$$

avec
- n : le nombre de postOpt de type Front de message queue appelé
- $Tps_{cmsgqpoOptFidif}$: temps d'exécution d'appel d'un postOpt de type Front pour une file de messages (1ère appel)
- $Tps_{cmsgqpoOptFcidem}$: temps d'exécution d'appel d'un postOpt de type Front pour une file de messages déjà appelée dont l'appel succède à un autre appel de postOpt de type Front
- $Tps_{cmsgqpoOptFcdif}$: temps d'exécution d'appel d'un postOpt de type Front pour une file de messages différentes dont l'appel succède à un autre appel de postOpt de type Front
- $Tps_{cmsgqpoOptFiidem}$: temps d'exécution d'appel indépendant de type Front pour des files de messages identiques

ici $Tps_{cmsgqpoOptFidif}$=384 $Tps_{cmsgqpoOptFcidem}$=376 $Tps_{cmsgqpoOptFcdif}$=335 $Tps_{cmsgqpoOptFiidem}$=324

D'où l'*overhead* suivant :

$$Tps_{cmsgqpoOptidif} + (n-1)Tps_{cmsgqpoOpticidem} \leq O_{postOptmessagequeuegéné} \leq n * Tps_{cmsgqpoOptidif} \quad \text{(C.66)}$$

C.0.21 Effet du changement de contexte sur le temps d'exécution

Le temps d'exécution d'une sauvegarde/restitution de contexte est fixe. Seulement le temps total de changement de contexte est dépendant du service qui provoque ce changement de contexte (sémaphore, mutex, mailbox, flag etc.).

$$O_{contextswitch} = Tps_{ctxtsw} * Nb_{ctxtsw} \quad \text{(C.67)}$$

avec

- Nb_{ctxtsw} : le nombre de sauvegarde/restitution de contexte
- Tps_{ctxtsw} : temps d'exécution d'une sauvegarde/restitution de contexte

ici Tps_{ctxtsw}=957

C.0.22 Coût de l'initialisation de ucos

On se met tout d'abord dans une configuration minimale du système d'exploitation afin de mesurer les différents coûts de chaque paramètres et services. Cette configuration minimale correspond à l'utilisation du seul service de tâches, d'un nombre de tâche de 1, de priorité dont la valeur maximale est 2, d'une profondeur de pile minimum de 1. Ceci afin de connaître le coût minimum de l'initialisation.

Dépend des services actifs puis :

- du nombre de tâches
- de la profondeur de pile allouer aux tâches
- du nombre de priorités
- de la profondeur de la pile de la tâche IDLE
- du nombre d'événements possible
- du nombre de message queue
- du nombre de flags possible

ce qui rajoute :

coût du service gestion des tâches :

$$C_{taskservice} = 4376 \quad \text{(C.68)}$$

coût du service gestion des flags :

$$C_{flagservice} = 25 \quad \text{(C.69)}$$

coût du service gestion des mailbox :

$$C_{mailboxservice} = 14 \quad \text{(C.70)}$$

coût du service gestion des message queue :

$$C_{messagequeueservice} = 21 \tag{C.71}$$

coût du nombre de tâches :

$$C_{tasknb} = Nb_{task} * 39 \tag{C.72}$$

coût suivant la profondeur de la pile suppposé des tâches :

$$C_{taskstascksize} = Ss * 92 \tag{C.73}$$

coût suivant la profondeur de la pile de la tâche idle :

$$C_{idletaskstacksize} = ISs * 92 \tag{C.74}$$

coût suivant le nombre de priorité :

$$C_{prioritynb} = Pnb * 39 + (39 * nblignes \in [0;7]) \tag{C.75}$$

coût suivant le nombre d'événements spécifiés :

$$C_{evntnb} = Nb_{evt} * 50 + 8_{sinbevnt>2} \tag{C.76}$$

coût suivant le nombre de flags spécifiés :

$$C_{flagsnb} = Nb_{flag} * 46 + 1_{sinbevnt>2} \tag{C.77}$$

coût suivant le nombre de message queue spécifiés :

$$C_{mssqnb} = Nb_{mssq} * 43 + 5_{sinbevnt>2} \tag{C.78}$$

Soit un coût d'initialisation de ucos de : D'où l'*overhead* suivant :

$$C_{ucosinit} = \sum C_{servicescost} + C_{tasknb} + C_{taskstacksize} + C_{idletaskstacksize} + C_{prioritynb} + C_{evntnb} + C_{flagsnb} + C_{mssqnb} \tag{C.79}$$

Résumé

Les travaux de cette thèse, menés dans le cadre du projet RNRT A3S, intègrent la notion de composants au sein d'une méthodologie de conception de plates-formes SoC (System on Chip), basée sur le langage de modélisation UML (Unified Modeling Language). Cette méthodologie propose un environnement de conception haut-niveau, basé sur le profil UML A3S, développé pour apporter la sémantique du domaine des systèmes temps réel embarqués et en particulier celle relative aux applications Radio Logicielle. Elle repose sur une approche MDA ou l'architecture est dirigée par les modèles où chaque modèle correspond à un niveau d'abstraction, à un niveau de raffinement particulier.

Une chaîne UMTS a permis la validation de l'outil réalisé, en confrontant les résultats estimés de l'outil, à ceux mesurés sur une plate-forme temps réel hétérogène (multi-DSP, multi-FPGA). Une partie du travail s'est concentré sur l'identification des composants utiles à la conception des systèmes SoC, et de leurs caractéristiques, en adéquation avec le niveau d'abstraction considéré. Une autre partie des travaux a porté sur la définition des modèles UML, et donc du profil, qui définissent la sémantique des différents composants identifiés en fonction de la configuration (PIM, PSM), ainsi que leurs relations. Une réflexion a été nécessaire afin d'élaborer les diverses règles de vérification et modèles d'exécution qui permettent d'informer le concepteur de ses erreurs et de la faisabilité du système modélisé. Un modèle de système d'exploitation a également été inclus, enrichissant la liste des éléments (composants) déjà définis et démontrant l'extensibilité du profil.

Abstract

The work of this PhD has been carried out within the framework of the A3S project and relies on component aspects integrated within a SoC platform design methodology, which is based on the UML language. This methodology proposes a high-level design framework based on the A3S UML profile developed to provide real-time embedded system semantic especially in SDR domain. An MDA approach has been considered to deal with different abstraction levels when specifying systems.

First part of the work focused on identifying the component required designing a SoC system, and their characteristics depending on the component abstraction levels. Several types of component (software and hardware) whose characteristics depend on their modelling (PIM or PSM models) have been considered. Second part of the work focused on the definition of UML metamodels, which are grouped to define the A3S UML profile that establish the semantic of identified components depending on their modelling and their relations. We have defined extensive verification rules and applied a model of computation to inform designers about errors that have been done and to ensure the feasibility of their systems. Finally an operating system model has been included to demonstrate the scalability and the extension mechanisms of the UML language and profile which improve the list of components that have been already integrated within our framework. An UMTS application has validated our approach by comparing the estimated results computed by the tool with measured results obtained on a heterogeneous real-time platform (with several DSP and FPGA).