UNIVERSITÉ DE MONTRÉAL

EXPÉRIMENTATION D'UNE SUITE D'OUTILS POUR AUTOMATISER LE PASSAGE D'UNE CONCEPTION BASÉE SUR UN MODÈLE VERS LA RÉALISATION D'UNE IMPLÉMENTATION, EN PASSANT PAR l'EXPLORATION ARCHITECTURALE

MATHIEU GAUDRON

DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION

DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES

(GÉNIE INFORMATIQUE)

MARS 2016

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

EXPÉRIMENTATION D'UNE SUITE D'OUTILS POUR AUTOMATISER LE PASSAGE D'UNE CONCEPTION BASÉE SUR UN MODÈLE VERS LA RÉALISATION D'UNE IMPLÉMENTATION, EN PASSANT PAR l'EXPLORATION ARCHITECTURALE

présenté par : GAUDRON Mathieu

en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées

a été dûment accepté par le jury d'examen constitué de :

M. BELTRAME Giovanni, Ph. D., président

M. BOIS Guy, Ph. D., membre et directeur de recherche

M. DAVID Jean Pierre, Ph. D., membre

REMERCIEMENTS

Je tiens à remercier :

- Mon directeur de recherche Guy BOIS pour son soutien et ses conseils tout au long de ma maîtrise. Sa disponibilité et sa proximité ont également été très appréciables pour travailler dans les meilleures conditions.
- Polytechnique Montréal, le consortium de recherche et d'innovation et aérospatiale du Québec (CRIAQ), le programme de stage du réseau de chercheurs en Mathématiques des technologies de l'information et des systèmes complexes (MITACS), le Conseil de recherches en sciences naturelles et en génie du Canada (CRSNG), ainsi que CMC Électronique et tous les partenaires industriels du projet AVIO 509, pour leurs contributions financière et matérielle qui m'ont permis de mener à bien mon projet de recherche.
- M. Jérôme HUGUES d'ISAE (Institute for Space and Aeronautics Engineering) Toulouse qui a toujours répondu à mes questions et qui m'a orienté au mieux sur mes choix d'implémentation.
- Les ingénieurs de SpaceCodesign et notamment Hubert GUÉRARD et Fellipe MONTEIRO qui ont toujours été prompts à répondre à mes questions.
- Mes collègues de bureau Étienne GAUTHIER et Arnaud DESSAULTY qui ont permis d'entretenir une ambiance de travail agréable, constructive et fort enrichissante.
- Ma famille qui m'a toujours soutenu, encouragé et motivé dans mes études et durant cette maîtrise.
- Les membres du jury qui prennent de leur temps pour évaluer mon travail.

RÉSUMÉ

Aujourd'hui, les systèmes embarqués sont de plus en plus complexes à développer surtout s'il s'agit de systèmes temps réel. Ces projets intègrent des technologies à la fine pointe de la recherche, qui sont compliquées à mettre en place. La complexité de conception de ces systèmes repose sur la nécessité de trouver un équilibre entre la puissance de calcul requise, la surface de carte et le nombre de ressources matérielles utilisées, ou encore la consommation du circuit. En ajoutant à tout cela des temps de mise en marché de plus en plus stricts pour ce genre de systèmes, les besoins d'outils et de flots de conception efficaces deviennent de plus en plus pressants.

Dans cette optique, de nombreux langages de spécification de système ont été mis au point. Ils sont échelonnés à différents niveaux d'abstraction allant des langages de haut niveau d'abstraction comme sysML ou AADL jusqu'au bas niveau RTL en passant par des spécifications pour ESL (Electronic system level) comme SystemC. Ces langages sont liés à des méthodologies basées sur les modèles.

Le projet de recherche présenté dans ce mémoire consiste à mettre en avant une méthodologie de conception d'un système embarqué. Cette méthodologie s'illustre au travers d'un flot de conception utilisant le langage de description de système AADL ainsi que la plateforme de codesign SpaceStudio. Elle vise à développer en parallèle des applications logicielles ainsi que les plateformes matérielles sur lesquelles ces applications doivent s'exécuter. Le défi de ce projet consiste donc à réaliser la jonction entre le langage AADL et la plateforme SpaceStudio. L'outil chargé de réaliser cette jonction compile du code AADL et génère un script python. Ce script est lu par l'API du logiciel SpaceStudio qui permet de générer un projet sur sa plateforme de coconception.

L'outil créé durant ce projet et nommé AADL2Space est testé à travers un exemple de modèle AADL disponible sur Internet. Par la suite, une application de décodage vidéo MJPEG est utilisée pour illustrer le flot de conception. Un modèle AADL de cette application a été développé afin de fournir la description architecturale du système. La partie applicative du système a été codée en C et associée au modèle AADL. Ainsi, un système complet est compilé par AADL2Space pour ainsi générer un projet SpaceStudio. Une fois le projet instancié sur la plateforme de coconception, celui-ci est simulé et analysé afin d'obtenir des métriques permettant de valider ou non l'architecture. De cette façon, plusieurs architectures sont testées afin de satisfaire les contraintes

d’ordonnancement temps réel, de taux d’utilisation des processeurs, d’utilisation des ressources matérielles, etc. L’architecture choisie est enfin synthétisée pour être implémentée sur carte.

Ce projet a conduit à l’écriture d’un article de conférence à EEE international Symposium on Rapid System Prototyping (RSP)

ABSTRACT

Nowadays, embedded systems are increasingly complex to design. These system's design complexity is based on the need to find a balance between the required power, the used area on ship and hardware resources, and the system consumption. This issue mainly occurs for real-time systems. For such systems, times to market are more and more demanding. Consequently, new tools and design flows are definitely needed. This project bridges and validates two of these technologies.

To reach our goal, numerous system description languages and libraries have been worked out. They have different abstraction levels from high abstraction level languages as SysML or AADL, to low level abstraction RTL, through ESL (Electronic system level) as systemC.

The aim of the research project introduced in this work is to show an embedded system design methodology. This methodology is illustrated through a design flow using the description language AADL and the SpaceStudio™ HW/SW co-design platform. It targets a parallel design of software applications and hardware platform on which applications will be executed. This project's challenge is to fill the gap between the description language AADL and SpaceStudio platform. SpaceStudio is a scriptable tool. All the graphic manipulations can also be achieved through a Python script. The proposed tool filling this gap acts as a compiler of an AADL code and generate a Python script that can be used as an input description of SpaceStudio.

The created tool called AADL2Space is tested thanks to an AADL model example available on Internet. Next, an MJPEG video decoder application is used to illustrate the design flow. An AADL model of this application has been designed to provide the system's architectural description. The software part of the system has been coded in C language and bound to the AADL model. Thereby, a complete system is compiled by the designed tool and generated as a SpaceStudio project. Once the project has been instantiated on the co-design platform, it is simulated and analyzed to validate metric performances. Different architecture configurations are tested to meet system's constraints as real time scheduling, processor's use rate, use of hardware resources, etc. The chosen architecture configuration is finally synthetized to be implemented on a FPGA.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

MBE	Model based engineering
AADL	Architecture and analysis design language
VSP	Virtual system prototype
MJPEG	Motion JPEG (Joint Photographic Expert Group)
RTL	Register transfer language
IDE	Integrated development environment
API	Application programming interface
CPU	Central processing unit
ASIC	Application specific integrated circuit
FPGA	Field programmable gate arrays
ESL	Electronic system level
TLM	Transaction-level modeling
UTF	Un-timed functional
TF	Timed functional
LT	Loosely-timed
AT	Approximate-timed
ISS	Instruction set simulator
OS	Operating system
UML	Unified modeling language
SysML	Systems modeling language
FIFO	First in first out

IPC	Inter-process communication
SoC	System on chip
MPSoC	Multiprocessor system on chip
SCoPE	SoC Co-simulation and performance estimation

LISTE DES ANNEXES

CHAPITRE 1 INTRODUCTION

1.1 Contexte

La complexité de développement des systèmes embarqués augmente de façon proportionnelle avec les progrès technologiques réalisés dans ce domaine. Alors que ces produits doivent respecter des calendriers de mise sur le marché de plus en plus stricts pour rester compétitifs, les temps de conception augmentent. Pour pallier cette tendance inverse d'évolution du temps de mise sur le marché et de temps de conception, les concepteurs doivent se servir de différents outils de représentation et d'analyse d'un système.

Ces outils se présentent sous la forme de différents langages de description de niveaux d'abstraction plus ou moins élevés : Un flot de conception classique d'un système embarqué débute par une représentation très haut niveau de manière à fournir un modèle du système clair et très visuel permettant de rapidement poser les fondations du système à concevoir. Ce flot se termine par une description matérielle très bas niveau pouvant ensuite être implémentée sur une carte reconfigurable (FPGA) ou directement sur un circuit ASIC.

À partir de ces langages de description, les concepteurs peuvent procéder à une compilation permettant de valider la sémantique d'un modèle et de réaliser une simulation de celui-ci. Différentes analyses sont alors effectuées pour valider les choix architecturaux et affiner le système au fil des étapes.

1.1.1 Enjeux des systèmes embarqués

Le principal enjeu de ces systèmes est de jouer sur l'équilibre entre la puissance requise, la surface de circuit utilisée, ou encore la consommation maximum. L'amélioration de l'un de ces paramètres entraine bien souvent la détérioration des autres, mais ces effets sont difficilement prévisibles et l'exploration de différentes architectures est coûteuse en temps.

Un bon flot de conception d'un système embarqué doit permettre d'analyser rapidement les métriques décrites ci-dessus afin de cibler aux mieux l'architecture souhaitée et d'être en mesure de valider les différentes contraintes fixées par un cahier des charges.

1.1.2 Ingénierie basée sur les modèles

Dans le but de pouvoir répondre à ces contraintes de temps de mise sur le marché et de complexité des systèmes, et donc de répondre aux besoins de l'industrie, de nouvelles approches sont utilisées comme

la conception basée sur les modèles (MBE) [1, 2]. Cette approche consiste en l'élaboration de modèle de haut niveau en utilisant uniquement les éléments nécessaires à la compréhension du contexte de conception. Ces méthodes se basent donc sur une abstraction importante des détails d'un système, afin de faciliter la compréhension de celui-ci. L'ingénierie basée sur les modèles amène une représentation d'un système sous forme de composants qui sont soumis à une hiérarchisation qui permet de définir certains composants comme internes à d'autres.

La conception dirigée par les modèles, en début de flot permet d'exprimer les requis non fonctionnels d'un système [3]. De façon à rapidement aboutir à une première description du système qui permet de définir les lignes directrices de la conception, celle-ci est réalisée habituellement de manière incomplète puis est affinée dans la suite du flot de conception.

De plus, ces méthodes de développement permettent souvent de réaliser des tests et des vérifications très tôt dans le processus de conception, donnant ainsi la possibilité de détecter rapidement des problèmes liés au modèle et de les régler en limitant les coûts.

1.2 Problématique

Dans ce contexte, la problématique de recherche repose sur la question suivante : « Est-il possible de bâtir un flot de conception de systèmes embarqués, basé sur des modèles et capable d'explorer différentes architectures de plateformes matérielles d'exécution à des fins de validation et d'amélioration des performances du système ? »

Le problème est motivé par le besoin de plus en plus important de méthodologies de développement des systèmes embarqués efficaces, afin de compenser la complexité grandissante de conception de ce genre de systèmes. En effet, des efforts doivent être réalisés à tous les niveaux du développement (conception, validation, production) pour pallier les contraintes de temps de mise sur le marché de plus en plus serrées.

La problématique a rapidement été affinée avec la volonté d'utiliser dans le flot de conception, le langage de description de système à haut niveau AADL ainsi que la plateforme de coconception logicielle/matérielle SpaceStudio. La nouvelle problématique peut donc être résumée par la question suivante : « Comment effectuer le lien entre le langage de description basé sur les modèles AADL et la plateforme de codesign SpaceStudio afin d'intégrer ces outils à un flot de conception de systèmes embarqués ? »

1.3 Objectifs

Les travaux effectués dans le cadre de cette maîtrise visent à améliorer un flot de conception pour les systèmes embarqués. Ce flot [4] a été conçu pour traduire la partie logicielle d'une description AADL vers un modèle SystemC. Le problème de ce flot est qu'il occulte complètement la description matérielle qu'il est possible de fournir avec un modèle AADL. L'amélioration proposée se base sur la conception d'une « passerelle » entre le langage AADL [5] qui permet la description et la modélisation d'un système, et un prototype virtuel de ce système (VSP : Virtual System Prototype) [6].

Pour ce faire, les objectifs suivants ont été dégagés de la problématique :

- Le premier objectif est de développer les connaissances sur les outils disponibles pour la conception de systèmes embarqués.

Cet objectif permet d'approfondir les connaissances sur les recherches effectuées dans le domaine et également de valider les choix d'outil explicités précédemment.

- Le second objectif consiste à analyser et à capturer toute l'architecture logicielle du système décrit en AADL.

Ce second objectif implique d'être capable d'analyser et d'enregistrer toutes les communications qui existent entre les différents composants logiciels du système ainsi que de récupérer les différents codes sources associés, le cas échéant. Dans le cas contraire, une structure ou un «squelette» de code est généré pour guider l'utilisateur dans la réalisation des modules logiciels.

- Le troisième objectif vise à porter la plateforme matérielle du système décrite en AADL, sur la plateforme SpaceStudio.

Cet objectif implique pour chaque type de composant de la plateforme matérielle (processeur, bus, etc.) de faire transiter l'information du modèle du composant souhaité par l'utilisateur pour l'implémenter avec la bonne technologie sur la plateforme SpaceStudio.

- Le quatrième objectif a pour but de réaliser les branchements entre les modules logiciels créés sur SpaceStudio et la plateforme matérielle.

Cet objectif implique de connaitre les différents branchements indiqués dans le modèle pour chaque composant logiciel, mais également pour les composants matériels qui ont besoin d'être connectés les uns aux autres

- Le cinquième et dernier objectif consiste à valider les passerelles créées aux objectifs 2, 3 et 4 et ainsi valider l'ensemble du flot de conception, allant du modèle à l'implémentation.

Cet objectif implique donc la création d'un exemple complet pour illustrer l'ensemble du flot de conception, identifier ses différentes caractéristiques et valider son bon fonctionnement.

1.4 Méthodologie

La réalisation du premier objectif sera faite par l'intermédiaire d'une revue de littérature mettant l'accent sur la conception basée sur les modèles et sur les langages ESL (Electronic System Level).

La réalisation du second objectif ainsi que celle du troisième et du quatrième consistera en l'utilisation et l'amélioration du logiciel Ocarina (Open Source) qui agit comme un compilateur de code AADL en effectuant une analyse syntaxique et sémantique. À partir d'un arbre abstrait généré par le logiciel seront générés tous les codes sources qui décriront le comportement du système sur SpaceStudio ainsi qu'un script Python. Ce sera ce script qui grâce à une interface de programmation (API) dédiée à SpaceStudio permettra la création du projet sous la forme de modules logiciels et d'une plateforme matérielle.

Finalement, pour la réalisation du cinquième objectif, l'expérimentation du flot de conception se fera par l'intermédiaire d'une application de décodage vidéo MJPEG (Motion JPEG). Un modèle de l'application AADL décrivant la partie logicielle du système ainsi que deux plateformes matérielles possibles, seront élaborés. Ensuite, ce modèle sera amené sur SpaceStudio par l'intermédiaire de l'outil présenté puis raffiné grâce à la plateforme de codesign. La validation de différentes caractéristiques de l'outil se fera à l'aide d'un modèle AADL disponible sur Internet [7] qui sera légèrement modifié pour effectuer les tests nécessaires.

1.5 Contribution

Pour atteindre les différents objectifs fixés, les contributions suivantes découleront des travaux de recherche effectués :

- Une revue de littérature présentant les principaux outils et langages utilisés dans le domaine des systèmes embarqués
- Un outil permettant de réaliser la traduction et le portage d'une description AADL vers la plateforme de codesign SpaceStudio.

- Une proposition de flot de conception de systèmes embarqués utilisant le langage AADL ainsi que la plateforme SpaceStudio et mettant en œuvre l'outil réalisé.
- Une Génération de résumés clairs des simulations réalisées sur la plateforme SpaceStudio dans l'optique de les fournir à un concepteur AADL pour qu'il puisse modifier son modèle en conséquence.
- La réalisation d'un modèle AADL d'une application de décodage vidéo MJPEG pouvant être réutilisé et permettant d'illustrer le flot de conception.
- Écriture d'un article scientifique soumis pour la conférence ERTS 2016 (Embedded Real Time Software and Systems).
- Écriture d'un papier présenté lors du symposium Rapid System Prototyping à Amsterdam en octobre 2015.

1.6 Structure du mémoire

Ce mémoire est structuré comme décrit ci-dessous :

- Le chapitre 2 présente la revue de littérature montrant différents outils de conception de systèmes embarqués.
- Le chapitre 3 introduit les concepts technologiques et l'environnement dans lequel se trouve l'outil créé
- Le chapitre 4 présente et détaille différents aspects de l'outil ainsi que sa validation.
- Le chapitre 5 présente une mise en application de l'outil en illustrant le flot de conception au travers d'un exemple d'application de décodage vidéo MJPEG et en effectuant divers tests de validation de l'outil.
- Le chapitre 6 présente les résultats obtenus lors de l'exploration architecturale du système de décodage vidéo MJPEG.

Finalement, une conclusion est apportée avec une synthèse générale des travaux effectués dans le cadre de ce projet et les perspectives envisagés pour les futures recherches.

CHAPITRE 2 REVUE DE LITTÉRATURE

2.1 Conception basée sur les modèles

La conception et l'évolution des systèmes sont souvent ralenties par leur taille et leur complexité [8]. D'un côté, ces systèmes embarquent des applications logicielles utilisant de plus en plus de nœuds de calcul devant collaborer et communiquer les uns avec les autres. Et d'un autre côté, la capacité d'adaptation des plateformes matérielles est de plus en plus importante notamment grâce aux cartes FPGA.

L'ingénierie basée sur les modèles consiste à fournir à l'utilisateur des interfaces de représentation pour ce genre de systèmes. Les nombreux aspects de la plateforme matérielle et de l'application logicielle sont abstraits sous la forme de composants. La réutilisation et la hiérarchisation de ces composants permettent de modéliser un système entier de façon claire et précise.

Afin d'augmenter la précision des modèles, il est possible d'associer différents types de propriétés aux composants pouvant, ou non, être utilisés par des outils d'analyse.

Les objectifs ciblés par les méthodologies de conception basée sur les modèles sont multiples :

- Les modèles réalisés doivent permettre d'avoir une vision globale du système pour permettre aux différents acteurs de sa conception de s'entendre sur l'architecture et les caractéristiques de celui-ci.
- Ces méthodologies doivent valider une cohérence du système assurant robustesse, fiabilité et sécurité lors de son exécution.
- Les modèles mis en place doivent permettre de réaliser des analyses préliminaires sur le système et ainsi détecter des problèmes de conception très tôt dans un flot permettant ainsi d'économiser beaucoup de temps.

Plusieurs méthodologies ont ainsi été mises au point pour répondre à ces objectifs, comme des extensions du langage UML très portées sur une modélisation des systèmes à travers différents diagrammes ou le langage AADL développé à la base pour les domaines de l'avionique et de l'automobile qui nécessitent beaucoup de vérification et de validation.

2.1.1 UML/MARTE

La méthodologie de modélisation UML/MARTE [9] suit une approche orientée vers les composants et centrée sur la partie logicielle d'un système.

De façon très classique à l'UML, cette représentation s'appuie sur plusieurs diagrammes de base : diagramme de données, diagramme fonctionnel, diagramme de communication et de concurrence, diagramme de description de plateforme qui décrit à la fois la partie matérielle et logicielle, diagramme d'architecture, ou encore un diagramme de vérification.

La méthodologie permet donc de représenter la plateforme logicielle/matérielle au travers des diagrammes de plateforme et d'architecture. Le diagramme de plateforme répertorie les déclarations des composants logiciels et matériels sous la forme de classes UML. L'architecture du système est représentée dans le diagramme d'architecture qui utilise des instances des différents composants et qui met en place les communications port à port entre eux.

Il est également possible de mettre en place un scénario d'exécution, ce qui différencie cette méthodologie des autres approches basées sur les modèles. Ceci est fait en venant connecter les composants du système à son environnement et en déclarant une séquence ordonnée des interactions avec celui-ci.

Le diagramme de concurrence et de communication définit les composants logiciels ainsi que l'application vue comme un agglomérat de composant communiquant entre eux. Dans ce diagramme, des propriétés peuvent être déclarées pour chaque composant comme des propriétés temps réels non fonctionnels. Il est également possible d'assigner une classe fonctionnelle à un composant qui va permettre de définir son comportement fonctionnel.

Une infrastructure de simulation SCoPE+ est utilisée pour réaliser quelques analyses de performances et de temps d'exécution à partir des diagrammes UML d'un système

UML/MARTE permet, grâce à tous ses diagrammes, de représenter précisément et à haut niveau les spécificités d'un système. Cette méthodologie hérite des quelques défauts de l'UML à savoir un grand nombre de diagrammes et donc une modélisation laborieuse. De plus, beaucoup de références sont à mettre en place entre les différents diagrammes pour avoir un modèle complet et cohérent.

2.1.2 SysML

SysML [10] est un standard dans le domaine de la conception de systèmes complets qui embarquent du logiciel dit critique. Ce langage est basé sur l'UML 2 et y ajoute des extensions. Par rapport à l'UML, le SysML apporte des avantages pour la conception de systèmes embarqués : pour une meilleure représentation de la sémantique des systèmes, SysML utilise des blocs d'appel à la place des blocs de classe utilisés par UML. De plus, SysML permet de réaliser des déclarations de composants de façon transverse aux différents diagrammes permettant à l'utilisateur de faire du référencement de composant.

Le diagramme de définition de bloc du SysML permet de mettre en place l'environnement du système. Il permet de déclarer tous les blocs extérieurs au système et de les interconnecter.

Le diagramme de bloc interne permet de raffiner les différentes connexions qui existent entre le système et son environnement. Il permet notamment de renseigner les types de données qui sont échangées par le système et son environnement et de nommer les ports de communication qui font transiter ces données.

Pour décrire l'aspect comportemental d'un bloc logiciel, l'utilisateur peut se servir d'un diagramme d'état ou d'un diagramme d'activité. Si le comportement du bloc est très séquentiel alors la représentation à l'aide d'un diagramme d'état est judicieuse et constitue un processus important du développement pour modéliser les états et les transitions entre eux. À contrario, l'utilisation d'un diagramme d'activité est plus adaptée pour un comportement très peu séquentiel.

Pour ajouter des spécificités au modèle, l'utilisateur peut se servir d'un diagramme de besoins ou d'exigence qui permet d'indiquer quelle partie du modèle permet de répondre à quelle exigence du système.

Globalement, le SysML permet de réaliser une bonne description d'un système logiciel critique. De plus, cette représentation permet de décrire l'environnement du système très en détail en pouvant même lui associer des modèles physiques. SysML permet donc de modéliser un système de façon précise, mais il ne permet pas d'effectuer des analyses directement à partir du modèle. De plus, l'accumulation des diagrammes rend la conception du modèle laborieuse et la lecture de ces diagrammes n'est pas toujours claire.

En bref, SysML se situe dans la lignée d'UML et permet de réaliser une bonne modélisation des spécificités d'un système, mais n'offre pas la possibilité d'effectuer des tests et des analyses à haut niveau.

2.1.3 AADL

Le langage AADL [5, 11] (Architecture Analysis & Design Language) a été développé par SAE international et approuvé en 2004 comme le standard AS 5506. Ce langage a été conçu dans l'optique de fournir un standard et une syntaxe précise pour des systèmes critiques et de permettre une représentation simple et complète d'une architecture pouvant être analysée très tôt dans le flot de conception. AADL est de plus en plus utilisé dans des domaines porteurs et à la pointe de la technologie comme l'avionique, l'automobile, l'aérospatial, la robotique et plus généralement dans le domaine des systèmes temps réels.

Des composants logiciels et matériels sont déclarés et instanciés pour composer l'architecture d'un système. Ils sont dans un premier temps modélisés comme « boites noires » en renseignant leurs interfaces. Dans un second temps, la composition interne et les détails de l'implémentation d'un composant sont déclarés. C'est ainsi que la hiérarchie de l'architecture est mise en place puisqu'un composant peut être conçu à partir de plusieurs autres composants.

Le langage AADL reste un langage de description d'architecture. Et même s'il permet de modéliser de nombreux aspects du système grâce, notamment, à des schémas sur les échanges de données, ou encore en apportant des précisions sur les caractéristiques temps réel des composants, le langage ne permet pas de réaliser l'implémentation comportementale du système. Cependant, il est possible de lier les différents composants logiciels du système à des implémentations applicatives extérieures au modèle [12].

Les composants appelés « subprograms » sont associés à des fonctions provenant de fichiers extérieurs au modèle AADL. Des appels à ces composants sont ensuite effectués par les composants « threads » du modèle qui représentent les tâches du système. Les fonctions ainsi associées à un modèle AADL servent uniquement à indiquer à l'utilisateur les différents appels de fonction du système, mais ne sont pas exécutés lors des analyses puisque le langage AADL ne permet pas de réaliser des vérifications comportementales, mais bien des analyses architecturales.

Comme cela a été dit, l'un des buts de ce langage est de permettre à l'utilisateur de réaliser plusieurs types d'analyses à partir d'un modèle AADL, donnant ainsi la possibilité à celui-ci de repérer très tôt d'éventuelles erreurs structurelles. Plusieurs outils ont été mis au point pour effectuer ces analyses comme le projet CHEDDAR [13] qui vérifie l'ordonnancement du système ou encore le projet REAL [14, 15] (Requirement Enforcement Analysis Language) qui sert à valider l'intégrité d'un modèle AADL.

De nombreux projets s'appuient sur le langage AADL pour concevoir et analyser une architecture afin de la valider ou bien d'améliorer les aspects de fiabilité, performance, et sécurité du système. C'est le

cas du système de vol de la sonde Juno [16] qui devrait atteindre Jupiter courant juillet 2016. Avoir un modèle AADL du système dès les premières phases de conception a permis aux équipes de la mission de s'en servir comme support pour le développement et pour bien comprendre les besoins opérationnels des instruments.

Deux mécanismes d'extension du langage AADL sont proposés. Il s'agit du mécanisme d'extension pure et du mécanisme de définition de propriétés (de l'anglais *Property Set*). Le mécanisme d'extension permet de réutiliser des composants et de compléter leurs définitions à la manière d'une extension de classe d'un langage de programmation orientée objet. Le mécanisme de définition de propriété permet de définir des propriétés AADL personnalisées qui viennent se greffer à l'ensemble des propriétés de base du langage AADL.

Les systèmes temps réel peuvent être modélisés efficacement grâce à ce langage qui permet d'appliquer des propriétés aux différents composants d'une architecture. Certaines de ces propriétés sont notamment utilisées pour indiquer des spécifications temporelles comme la période d'exécution d'un thread, son pire temps d'exécution, ces échéances, etc.

En résumé, le langage AADL permet de modéliser l'architecture d'un système de façon simple avec un haut niveau d'abstraction. Plusieurs outils peuvent ensuite se servir de cette modélisation dans le but d'analyser et de vérifier l'implémentation et l'intégration, mais surtout la certification du système.

La modélisation d'un système en langage AADL peut être réalisée de façon textuelle dans un éditeur de texte, mais également par l'intermédiaire d'une interface graphique. De nombreux éditeurs existent, mais on retrouve très souvent l'outil OSATE qui est greffé sur l'IDE Eclipse. OSATE intègre les différents outils d'analyse d'un modèle AADL qui sont directement accessibles à partir de l'interface de l'outil.

2.2 Electronic system level (ESL)

Comme les systèmes embarqués sont de plus en plus complexes, ils ne peuvent plus être décrits manuellement au niveau des transferts de registres (RTL) sans introduire de nombreuses erreurs de conception [17]. De plus, les SoCs embarquent de nombreux processeurs possédant des ensembles d'instructions différents qui complexifient les interfaces matérielles/logicielles.

L'ESL est une approche récente qui se place un niveau d'abstraction du système [18] et qui permet d'effectuer les vérifications fonctionnelles et structurales d'un système tôt dans le flot de conception.

Cette approche utilise les techniques de modélisation au niveau des transferts (TLM) [19] qui fournissent des abstractions permettant de concilier précision des modèles et vitesse de simulation.

Le but de cette approche est de rapidement mettre au point une plateforme virtuelle sur laquelle l'application logicielle doit s'exécuter. Pour cela, les TLM sont utilisés pour modéliser les différents blocs matériels. La plateforme virtuelle permet d'utiliser des composants décrits à différents niveaux d'abstraction en utilisant des transacteurs. Le déploiement matériel dépend de la partie logicielle du système, car, suivant si un thread s'exécute sur un processeur ou comme un accélérateur matériel, le branchement de ces ports de communications seront différents.

Les unités de calculs des processeurs sont modélisées grâce à différents simulateurs de niveau jeu d'instruction (de l'anglais *Instruction Set Simulator* ou ISS). Chaque ISS est fourni avec une interface de bus. Ce sont donc ces ISS qui jouent le rôle d'interface entre le matériel et le logiciel.

Une fois la plateforme en place, une cosimulation matérielle/logicielle est possible. Cette simulation permet de réaliser une validation fonctionnelle du système, mais renseigne également sur les performances du système.

Suivant les résultats de simulation, une exploration architecturale peut être nécessaire pour faire coïncider les performances du système avec les différentes spécifications demandées. Cette exploration permet d'analyser un certain nombre de performances identifiées, et au besoin d'effectuer des modifications sur l'architecture afin de faire converger ces mêmes performances avec celles demandées dans la spécification.

La suite de cette approche consiste en la réalisation d'un raffinement au niveau RTL appliqué à chaque composant du système. La description RTL permet ensuite de récupérer des informations sur les ressources matérielles nécessaires à la conception du système.

Pour réaliser une description de système au niveau d'abstraction ESL, différents outils sont décrits ci-dessous.

2.2.1 SystemC

SystemC [20] est une librairie C++ qui permet notamment de mettre en place, dans un code compilable, les notions de thread et de composants matériels. Il permet donc à l'utilisateur de représenter un système à travers son architecture qui met en place sa plateforme matérielle et sa description fonctionnelle (codes logiciels purs).

Dans la plateforme de cosimulation proposée, la partie software et modélisée grâce au langage de haut niveau C++. La simulation du code ainsi réalisée est ensuite simulée à l'aide d'un simulateur-ISS.

Ces ISS peuvent être codés directement en C++ ou en utilisant la librairie SystemC. La façon dont ils sont codés peut influencer sur la vitesse de simulation.

SystemC permet également de modéliser la plateforme matérielle du système. Cette architecture permet de modéliser les ports d'entrées/sorties des différents composants pour avoir un contrôle sur les flots de données échangées avec l'ISS via le protocole de communication interprocessus (IPC) et des encapsulations de bus.

Les protocoles IPC permettent de décrire les différents échanges de données possibles entre les threads d'un système. Ces protocoles sont de deux types : Les IPCs à hôte unique qui peuvent se présenter sous la forme de mémoires partagées, échange de message par « tube », FIFO ou les IPCs qui communiquent au travers d'un réseau et qui se présentent sous la forme d'un « socket ».

Pour permettre de synchroniser les communications de données entre la plateforme matérielle décrite avec systemC et l'ISS décrit en C++ et contrôlé par le noyau de simulation systemC, deux types d'encapsulation sont utilisés : Une encapsulation de bus ISS et une encapsulation IPC. Pour compléter la synchronisation entre les parties logicielles et matérielles du système, l'utilisateur peut implémenter un pilote ou un API pour communiquer avec un composant.

SystemC permet donc de réaliser une coconception et une cosimulation d'un système embarqué. Il permet de décrire le comportement logiciel du système, d'implémenter dans le même temps la plateforme matérielle et de synchroniser le tout grâce à des encapsulations de communication.

2.2.2 SCoPE

SCoPE [21] (SoC Co-simulation and Performance Estimation) est aussi une librairie C++ qui étend le SystemC. Cette librairie permet de simuler à la fois du code logiciel C/C++ basé sur les interfaces de système d'exploitation POSIX et uC/OS, et des composants matériels décrits en SystemC.

Grâce à cette librairie, l'utilisateur peut réaliser la simulation d'un ou de plusieurs blocs de code logiciels sur une plateforme matérielle spécifique. De nombreuses données peuvent être analysées à la suite de cette cosimulation comme le nombre de threads et de changements de contexte, l'utilisation des processeurs et les défauts de cache, ou encore l'évaluation de la consommation de la plateforme.

Un atout important de SCoPE est qu'il permet de renseigner en détail le comportement temps réel des blocs logiciels et des OS. Les concepts d'ordonnancement et de synchronisation sont pris en compte dans le modèle. SCoPE permet également de gérer la préemption et les priorités des tâches sous différentes politiques d'ordonnancement ce qui n'est pas le cas avec SystemC.

Du côté logiciel, SCoPE intègre l'API basée sur les standards POSIX qui est à ce jour la principale interface avec les systèmes d'exploitation. Ce standard permet de réaliser et d'exécuter la plupart des systèmes logiciels développés de nos jours. L'interface uC/OS est également gérée par la librairie.

La plateforme matérielle est modélisée à travers plusieurs modules matériels génériques codés avec SystemC.Des bus qui utilisent les standards TLM-2.0 permettent la communication avec les périphériques et la transmission des interruptions, avec des mémoires et des interfaces matérielles, etc.

Le lien entre les blocs logiciels et la plateforme matérielle est réalisé grâce un ensemble conséquent d'une centaine de pilotes basés sur la version 2.6 du noyau Linux, le système d'exploitation le plus répandu dans le domaine. En plus de permettre la simulation des modules du noyau, SCoPE donne également possibilité de simuler le traitement et l'ordonnancement des interruptions.

En résumé, SCoPE est une librairie très intéressante pour modéliser un système temps réel et pour réaliser la cosimulation de systèmes hétérogènes MPSoC (Multiprocessor System-on-Chip).

2.2.3 SpaceStudio

SpaceStudio [22] est une plateforme ESL de coconception logicielle/matérielle qui permet de simuler différents niveaux d'abstraction (e.g. UTF, TF, LT, AT, etc.). Elle fournit à l'utilisateur une interface graphique qui lui permet de mettre facilement et rapidement en place la plateforme matérielle du système et de brancher l'application logicielle dessus puis de générer un prototype virtuel du système (VSP). En arrière de la partie graphique, le système est codé avec un SystemC étendu pour supporter des encapsulations et des fonctionnalités spécifiques à la plateforme. L'architecture du système peut donc être facilement analysée et modifiée pour réaliser une exploration architecturale afin d'identifier rapidement la configuration qui correspond le mieux aux besoins et aux spécificités du système.

Pour réaliser l'exploration des différentes configurations d'un système, celui-ci doit pouvoir être exécuté sur différents processeurs et même directement sur des accélérateurs matériels. Pour ce faire, l'application doit être composée de plusieurs tâches concurrentielles. Ainsi, chaque tâche peut être exécutée sur un processeur ou comme un composant matériel pur. Les communications entre les

différents blocs logiciels sont réalisées par des FIFOs générées automatiquement en SystemC : Une API fournit des fonctions de communications spécifiques à la plateforme SpaceStudio qui les interprète et génère tout le mécanisme de communication qui en découle, suivant sur quel support les tâches s'exécutent.

Pour bâtir la plateforme matérielle du système, l'utilisateur s'appuie sur une bibliothèque de composants préconçus sous le standard TLM-2.0. Ces composants sont répertoriés sous différents types : bus, mémoires, processeurs, horloges, etc. SpaceStudio permet, grâce à son interface, de sélectionner et d'apporter les configurations de bases aux différents composants souhaités pour les ajouter à la configuration de la plateforme.

L'outil SpaceStudio intègre des calculs de performance qui permettent de réaliser une simulation de la plateforme virtuelle. La simulation est effectuée à la fois pour la plateforme matérielle du système et pour l'implémentation logicielle afin d'extraire les différentes données de performance. En réalité, les composants matériels disponibles dans la librairie SpaceStudio sont déjà instrumentalisés pour collecter les données de simulations et les informations de temps des événements importants de l'exécution : changements de contexte, début et fin de transfert de données sur un bus, etc. Les tâches applicatives du système sont elles aussi automatiquement instrumentées de façon non intrusive et transparente pour ne pas modifier les temps de simulations. De plus, cette instrumentalisation n'implique aucun changement dans une tâche logicielle lorsque celle-ci s'exécute sur un processeur et que l'utilisateur souhaite changer l'architecture pour l'exécuter comme un composant matériel ou inversement.

Après la simulation du système, SpaceStudio analyse les différentes données récupérées. Le résultat des analyses est alors résumé et présenté graphiquement à l'utilisateur. Différentes métriques sur les performances du système sont aussi disponibles, comme l'utilisation des messages de queue au cours de la simulation, le taux de chargement d'un processeur, l'ordonnancement des tâches, ou encore les changements de contexte. Ces différentes métriques permettent à l'utilisateur de comparer différentes architectures matérielles et différents branchements de l'application logicielle sur la plateforme matérielle.

Enfin, cet outil permet de calculer une estimation des ressources matérielles nécessaires pour implémenter le système réalisé sur une carte FPGA. Il permet effectivement de réaliser une cosynthèse logicielle/matérielle en prenant en compte les différents types de ressources disponibles sur les FPGA cibles.

Une fois l'architecture validée par l'utilisateur grâce aux outils mis à sa disposition, SpaceStudio permet de faire appel à des outils de génération de code VHDL, comme les outils de synthèse HLS de Xilinx, qui permettent de réaliser une description RTL du système.

SpaceStudio est donc un outil qui permet de réaliser un VSP au niveau de l'électronique du système en assistant l'utilisateur lors de la création d'une plateforme virtuelle permettant de brancher une application logicielle sur une plateforme matérielle et de simuler l'ensemble de ce système. De nombreuses métriques permettent alors à l'utilisateur d'analyser différents aspects du système et de réaliser une exploration architecturale avant de générer une description RTL pouvant être implémentée sur une carte FPGA.

2.3 Flots de conception

Pour la réalisation d'un système embarqué, il est intéressant d'intégrer au flot de conception une description basée sur les modèles pour obtenir rapidement une représentation claire du système et une méthodologie basée sur l'ESL qui permet de réaliser des simulations comportementales du système avant de réaliser une description RTL de plus bas niveau d'abstraction.

Deux flots de conception qui utilisent les outils présentés plus tôt sont décrits ci-dessous.

2.3.1 Projet SoCKET

Le projet SoCKET [23] (SoC toolKit for critical Embedded sysTems) vise à élaborer un flot de coconception logiciel/matériel pour les systèmes embarqués qualifiés de critiques (aéronautique, aérospatiale, automobile, domaine de la santé). Les points clés du projet sont la maîtrise de la complexité grandissante de ces systèmes critiques en améliorant le temps de mise sur le marché et en réduisant les coûts de conception ainsi que la validation et la certification des systèmes embarqués critiques. Le flot de conception proposé par le projet SoCKET se décompose selon les étapes suivantes :

- Définition des spécificités du système : Des propriétés du système peuvent être déclarées notamment grâce au langage UML/MARTE.
- Architecture globale SoC : Cette architecture est définie par les architectes système s'aidant d'outils complémentaires.
- Prototype virtuel : Ce prototype est développé dès que la spécification fonctionnelle des IPs est disponible. Une description IP-XACT est utilisée pour garder la cohérence entre les spécifications, les modèles TLM et la partie logicielle.
- Validation : Cette étape permet de se faire une bonne idée sur la validité du système conçu. Des assertions ainsi que des injections de fautes peuvent être faites pour réaliser cette validation.

À partir de la description du niveau ESL, le flot de conception se concentre dans un premier temps sur l'exploration architecturale en utilisant le langage de modélisation SystemC puis utilise la technologie HLS pour un prototypage rapide du système. Grâce à cette technologie, l'utilisateur peut facilement générer une description RTL de sa plateforme virtuelle récupérant ainsi des informations sur les ressources matérielles nécessaires au système. Ce qui représente le but premier de ce flot de conception.

Ce projet met clairement en évidence un flot de conception à suivre pour réaliser des systèmes embarqués critiques. Il décrit toutes les étapes nécessaires. Cependant, aucune automatisation n'est présentée pour faire le lien entre les différents outils du flot qui est réalisé de façon manuelle.

2.3.2 AADL/AADS/SCoPE

Dans leur article [21], Varona-Gomez, R. et al., présentent un flot basé dans un premier temps sur la modélisation d'un système grâce au langage AADL. À partir d'AADL, le modèle va être traduit vers SCoPE qui sert d'ESL dans ce flot. Cette traduction entre ces deux étapes du flot de conception est réalisée par l'outil AADS.

Le projet se sert d'AADL comme langage basé sur les modèles pour son efficacité de représentation et d'analyse d'une architecture. L'évolutivité du langage ainsi que les nombreuses propriétés applicables à un thread et les analyses qui en découlent ont aiguillé ce choix.

Pour pouvoir simuler les performances du modèle, le flot utilise SCoPE. Les résultats des simulations et des analyses de performances sont ensuite utilisés pour guider l'utilisateur dans les modifications à apporter à son architecture.

Pour réaliser le lien entre le langage haut niveau basé sur les modèles et l'ESL, le projet introduit les outils AADS qui permettent d'extraire les informations nécessaires à SCoPE à partir du modèle AADL. Deux outils sont utilisés : AADS qui est codé en Java et qui se présente sous la forme d'un plug-in Eclipse et AADS-T qui est une version de AADS qui permet de traiter les modèles AADL qui utilisent l'annexe comportementale d'AADL.

AADS permet de faire la traduction à la fois de l'architecture d'un système temps réel, mais également de ses fonctionnalités en allant notamment chercher les contraintes de temps pour les différents threads.

AADS-T est une version AADS qui inclut l'annexe comportementale d'AADL et qui génère du code source compatible avec le modèle de calcul Ravenscar. Ce modèle permet de réaliser des analyses de cohérence avec les propriétés temps réels qui ont été indiqués et la simulation du système. L'avantage de ce modèle est qu'il peut facilement être implémenté sur un noyau temps réel de petite taille. L'inconvénient de cet outil est que douze propriétés temps réel doivent impérativement être renseignées dans le modèle pour pouvoir procéder aux différentes analyses.

Le projet met donc en place un flot complet de conception d'un système embarqué qui commence par une modélisation d'un système en AADL qui est ensuite traduit en langage SCoPE. Cependant, les outils de traduction nécessitent un modèle AADL très complet pour pouvoir être utilisés correctement.

CHAPITRE 3 CONCEPTS TECHNOLOGIQUES

Ce chapitre présente les différents concepts technologiques du flot de conception proposé dans ce mémoire. Ce flot se concentre sur la création du lien entre le langage AADL et le logiciel SpaceStudio qui joue ici le rôle de VSP. Le langage AADL nous sert à décrire des systèmes à très haut niveau tout en permettant de renseigner de nombreuses caractéristiques pour les composants. Quant au logiciel SpaceStudio, il nous sert de plateforme virtuelle afin de tester le comportement du système sur sa plateforme matérielle avant de générer une description RTL du système.

3.1 AADL

À la fin de l'année 1999, Bruce Lewis (US Army AMRDEC) a initié le comité SAE AADL [24], signifiant *Avionic Architectures Description Language*, afin de mettre en place une standardisation de représentation architecturale validée, à l'époque, par 10 compagnies aérospatiales (SAE ARD 5296). En 2001, le nom fut changé en *Architecture Analysis & Design Language* pour refléter le fait qu'aucune spécificité avionique n'était contenue dans le langage.

Depuis, AADL a été une technologie clé pour de nombreux projets industriels comme ESA ASSERT (2004-2008) pour piloter le développement de satellites, ou encore le projet TOPCASED (2005-2009) mené par Airbus afin de bâtir un outil industriel libre de droit pour réaliser de l'ingénierie basée sur les modèles pour les systèmes embarqués.

Aujourd'hui, AADL est utilisé en majeure partie dans l'industrie aérospatiale et notamment par des compagnies telles qu'Airbus ou encore par l'agence spatiale européenne (ESA) pour piloter le développement de projets.

Tel que discuté à la section 2.1.3, le langage AADL est un langage de description architecturale. Il permet de déclarer l'architecture logicielle ainsi que les composants de la plateforme matérielle sur laquelle va s'exécuter un système. Il est possible d'associer différentes propriétés à chaque composant qu'il soit logiciel ou matériel. De nombreux tests peuvent ensuite être effectués sur le modèle AADL suivant la complexité du modèle et les propriétés qui ont été renseignées. Il est par exemple possible, en ayant au préalable défini un flot de données dans le modèle AADL, de tester le temps maximum mis pour traverser ce flot en entier permettant ainsi d'assurer un temps de réponse maximum du système. Cette section va en premier lieu présenter les composants de l'AADL avant de présenter les propriétés, puis finalement

nous présentons OSATE2, un logiciel libre, qui est la principale plateforme de développement du langage.

3.1.1 Composants

L'AADL est un langage déclaratif qui permet de définir des composants suivant deux étapes : la première étape consiste à déclarer le composant comme une boite noire. On lui associe alors des ports d'entrée et de sortie. La seconde étape permet de déclarer une implémentation du composant en agissant sur la composition interne de celui-ci comme les sous-composants et leurs branchements. Ces deux étapes de déclaration des composants sont présentées à la figure 3-1.

Le langage propose un large éventail de composants que l'on peut dissocier en plusieurs types :

3.1.1.1 Les composants logiciels

Ces composants permettent de définir le comportement et l'architecture de la partie logicielle du système.

- **Process**

  Un processus permet de définir une zone mémoire où va s'exécuter le code logiciel associé à ses sous-composants. Il peut déclarer des threads et des groupes de threads comme sous-composants. Sur la figure 3-1, on peut voir la déclaration d'un processus (1) et notamment les connexions réalisées à l'intérieur de celui-ci. Les représentations graphiques boite noire (2) et boite blanche (3) de ce processus sont également visibles sur la figure.

```
process p                                          1
features
      input : in data port Data_type;
      output : out event port;
end p;

process implementation p.impl
subcomponents
      t1 : thread a;
      t2 : thread b;
connections
      c1 : port input -> t1.input;
      c2 : port t1.output -> t2.input;
      c3 : port t2.output -> output;
end p.impl;
```

1

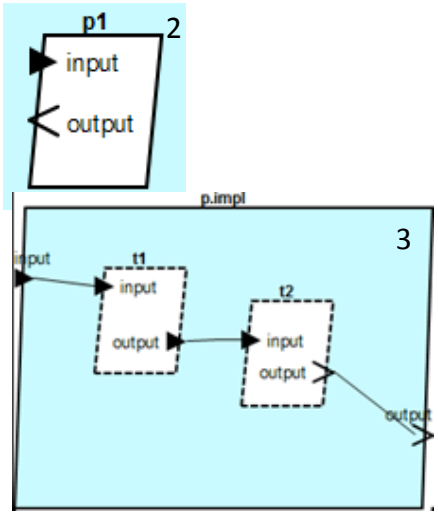

Figure 3-1 : Exemple de déclaration d'un processus

- **Thread et Thread group**

  Les threads ne sont autres que les files d'exécutions du logiciel. Ils partagent la mémoire de leur processus père avec les autres threads de ce processus. Les groupes de threads rassemblent plusieurs threads auxquels on peut assigner des propriétés communes. Les threads peuvent faire appel à des bouts de fonctions appelés *subprograms*.

- **Subprogram**

  Les sous-programmes font référence à des fonctions pouvant être codées dans plusieurs langages différents (seul le langage C va être utilisé dans ce projet). Ces sous-programmes peuvent être appelés par des threads où ils sont regroupés en séquence d'appels. La figure 3-2 montre comment un thread peut appeler un sous-programme et comment celui-ci fait référence à un code source grâce au mot clé *properties*.

```
subprogram test1
features
      in_parameter : in parameter Data_type;
      out_parameter : out parameter Data_type;
properties
      Source_Text => ("source.cpp");
      Source_Name => "function";
end test1;

thread implementation a.impl
calls
      loop : {code1 : subprogram test1;};
connections
      p1 : parameter input -> code1.in_parameter;
      p2 : parameter code1.out_parameter -> output;
end a.impl;
```

Figure 3-2 : Appel d'un sous-programme par un thread

- **Data**

  Certains ports de communications de composant sont référencés comme « data » ou « event data ». Il faut alors renseigner le type de donnée associé à ces ports. Ces données sont implémentées comme les autres composants AADL. Tel qu'illustré à la figure 3-3 où « Data_type » fait référence au type « long », elles peuvent faire référence à des types de données d'un langage.

```
data Data_type
properties
      Data_Model::Representation => ("long");
end Data_type;
```

Figure 3-3 : Déclaration d'un type de donnée

3.1.1.2 Les composants de la plateforme matérielle

Ces composants permettent de définir le support sur lequel va s'exécuter le logiciel

- **Bus**

  Les bus permettent aux différents composants matériels du système de communiquer entre eux. Un composant doit avoir un droit d'accès au bus pour être branché dessus. On peut voir sur la figure 3-4, la déclaration d'un bus ainsi qu'une requête d'accès présente à la déclaration d'un processeur.

```
processor ub
features
      socket : requires bus access AMBA;
end ub;

bus AMBA
end AMBA;
```

Figure 3-4 : Déclaration d'un bus

- **Processor**

  Le processeur est le composant matériel qui va exécuter le code logiciel. Plusieurs processeurs peuvent être déclarés dans un système. L'utilisateur doit alors renseigner sur quel processeur s'exécute chaque thread.

- **Memory**

  Les mémoires permettent de définir des zones mémoires accessibles par tous les threads du système.

3.1.1.3 Les composants hybrides

Ces composants ne peuvent, ni être classés comme logiciels, ni comme matériels, car ils sont présents sur les deux facettes du modèle

- **Device**[1]

  Les devices sont des composants en bordure du système. Ils sont représentés dans le modèle, mais ne font pas partie du système à développer. C'est une famille de composants, dont les capteurs et les actionneurs font partie, qui représente l'environnement d'exécution de notre système.

- **System**

  Le système global est déclaré comme un composant. Il crée des instances des composants dont il a besoin et les relie ensemble. Il permet également de lier les composants logiciels aux composants matériels. La figure 3-5 montre un exemple de déclaration d'un système en AADL (1) et sa représentation graphique (2).

[1] La traduction de device est *périphérique*, mais pour simplifier la lecture et faciliter le lien avec la terminologie AADL, nous resterons avec le terme *device*.

1

```
system implementation pba.speed
subcomponents
      speed_sensor : device sensor.speed;
      throttle : device actuator.speed;
      speed_control : process control.speed;
      interface_unit : device interface.pilot;
      RT_1GHz : processor Real_Time.one_GHz;
      Standard_Marine_Bus : bus Marine;
      Stand_Memory : memory RAM.Standard;
connections
      DC1 : port speed_sensor.sensor_data ->
                  speed_control.sensor_data;
      DC2 : port speed_control.command_data ->
                  throttle.cmd;
      DC3 : port interface_unit.set_speed ->
                  speed_control.set_speed;
      EC4 : port interface_unit.disengage ->
                  speed_control.disengage;
      BAC1 : bus access Standard_Marine_Bus <-> speed_sensor.BA1;
      BAC2 : bus access Standard_Marine_Bus <-> RT_1GHz.BA1;
      BAC3 : bus access Standard_Marine_Bus <-> throttle.BA1;
      BAC4 : bus access Standard_Marine_Bus <-> interface_unit.BA1;
      BAC5 : bus access Standard_Marine_Bus <-> Stand_Memory.BA1;
properties
      Allowed_Processor_Binding => (reference(RT_1GHz))
                  applies to speed_control.speed_control_laws;
      Allowed_Processor_Binding => (reference(RT_1GHz))
                  applies to speed_control.scale_speed_data;
      Actual_Memory_Binding => (reference(Stand_Memory))
                  applies to speed_control;

end pba.speed;
```

2

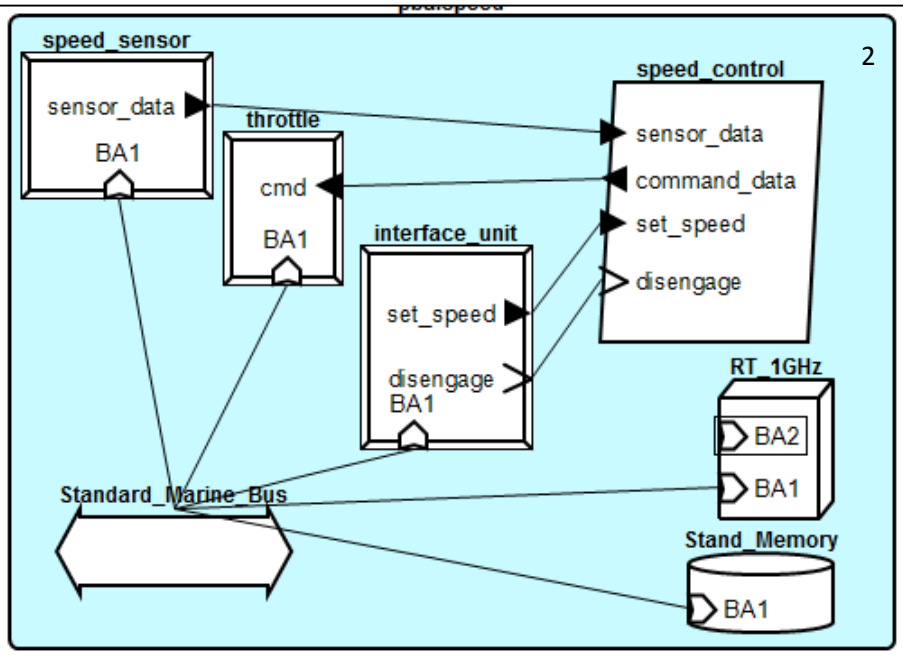

Figure 3-5 : Implémentation d'un système

3.1.2 Propriétés

L'un des grands intérêts du langage AADL est de pouvoir associer des propriétés aux différents composants d'un modèle. Ces propriétés peuvent se présenter sous différents types : les propriétés de type énuméré, de type entier, de type entier avec unité (e.g. 50 ms), etc. Le langage offre une grande panoplie de propriétés pour chaque type de composant utilisé, mais bien que la banque de propriétés disponibles soit importante, il est possible qu'un utilisateur ait besoin de propriétés plus spécifiques. Dans ce cas, il lui est possible de définir ses propres propriétés.

3.1.2.1 Déclarations

La déclaration des propriétés pour un composant peut se faire dans la déclaration de celui-ci, dans son implémentation ou dans les deux. Elle est précédée du mot clé « properties » comme on peut le voir sur la figure 3-6 où l'on indique dans sa déclaration que le thread b est périodique et dans son implémentation qu'il est exécuté toutes les 50 ms.

```
thread b
features
      input : in data port Data_type;
      output : out event port;
properties
      Dispatch_Protocol => Periodic;
end b;

thread implementation b.impl
properties
      period =>50 ms;
end b.impl;
```

Figure 3-6 : Déclaration de propriétés d'un thread

3.1.2.2 Property set

Tel qu'expliqué précédemment à la Section 2.1.3, si un utilisateur a besoin de propriétés spécifiques pour un projet, il peut créer un fichier de définition de propriété (« property set ») dans lequel il va définir de nouvelles propriétés ainsi que les différentes valeurs que peuvent prendre ces propriétés, si celles-ci sont de type énuméré. Il doit ensuite indiquer à quels types de composants ces propriétés sont destinées.

Pour utiliser un ensemble de propriétés défini par l'utilisateur, une description AADL doit indiquer qu'elle va y faire appel grâce à la syntaxe suivante :

```
with Nom_property_set;
```

Ensuite, l'utilisation d'une des propriétés de l'ensemble se fait de la manière suivante :

```
Nom_property_set::Nom_propriété => valeur;
```

3.1.3 OSATE2

OSATE 2.0 (Open-Source AADL Tool Environment) [25] est un outil libre basé sur IDE Eclipse et peut être ajouté comme module d'extension à celui-ci.

3.1.3.1 Développement

Cet outil nous permet d'éditer du code AADL et de réaliser une analyse de celui-ci en direct, permettant de faciliter la génération de code et la recherche de bugs. OSATE permet, entre autres, d'analyser la consistance d'un flot de donné qui doit posséder le même type de donnée.

Cet outil permet de générer un graphe pour faciliter la visualisation du système développé. Il est également possible de créer et d'éditer un modèle directement à partir de l'interface graphique pour des utilisateurs plus à l'aise avec ce type de développement.

3.1.3.2 Tests

Le langage AADL permet de réaliser facilement des tests à haut niveau. Ces tests sont très importants, car ils permettent de détecter très tôt des problèmes de conception faisant ainsi économiser beaucoup de temps aux équipes de programmation. L'outil OSATE 2 permet de réaliser bon nombre de ces tests préliminaires : on peut par exemple citer le projet CHEDDAR [26] qui permet de réaliser une analyse d'ordonnancement du système en fonction des propriétés renseignées dans le modèle.

3.2 SpaceStudio

SpaceStudio a été présenté dans la section 2.2.3 comme un outil développé par la compagnie SpaceCodesign [27] qui permet un codéveloppement de la partie comportemental d'un système (i.e. le logiciel) et de sa plateforme matérielle. Dans cette section, nous allons davantage mettre l'accent sur l'utilisation de SpaceStudio qui va nous permettre de valider les spécifications à la fois fonctionnelles et non fonctionnelles du système en créant un modèle exécutable systemC et de réaliser l'implémentation du système au niveau RTL en fin de projet.

3.2.1 Implémentation comportementale du système

La première étape de conception d'un système sur la plateforme consiste à implémenter la partie logicielle du système. Elle s'organise à travers deux types de composants : les modules et les périphériques que nous nommerons *devices* dans la suite de ce travail. Les modules sont des encapsulations aux threads du système alors que les devices sont, tout comme dans le langage AADL, des composants en bordure du système. Ces composants sont codés en SystemC qui permet de réaliser de nombreuses abstractions ESL dans des fonctions qui sont des encapsulations gérées par la plateforme. Ces encapsulations sont très pratiques, car elles permettent d'exécuter les modules aussi bien sur un processeur que comme un composant matériel sans avoir à modifier les codes de ces modules.

Ces modules doivent donc implémenter un constructeur spécifique à la plateforme qui va déclarer le thread qui sera exécuté par le module.

Pour échanger des données, ces modules doivent utiliser des fonctions spéciales qui abstraient des appels à différentes structures matérielles suivant si les modules s'exécutent sur un processeur ou en tant que composants matériels. Ces fonctions sont appelées *ModuleRead* et *ModuleWrite*. Elles prennent comme arguments : l'identifiant du module avec lequel la communication est établie, le comportement d'attente du module (la communication peut être bloquante ou non), l'adresse où recevoir le message ou à partir de laquelle l'envoyer et la taille du message.

3.2.2 Plateforme d'exécution

3.2.2.1 Création de la plateforme

SpaceStudio permet ensuite, grâce à une bibliothèque de périphériques matériels, de configurer une plateforme d'exécution que l'on appelle « configuration ». D'un côté, l'outil permet de sélectionner les modules et devices préalablement développés que l'utilisateur veut intégrer dans la configuration (figure 3-7), et de l'autre, il permet d'y intégrer les composants matériels nécessaires au système (figure 3-8).

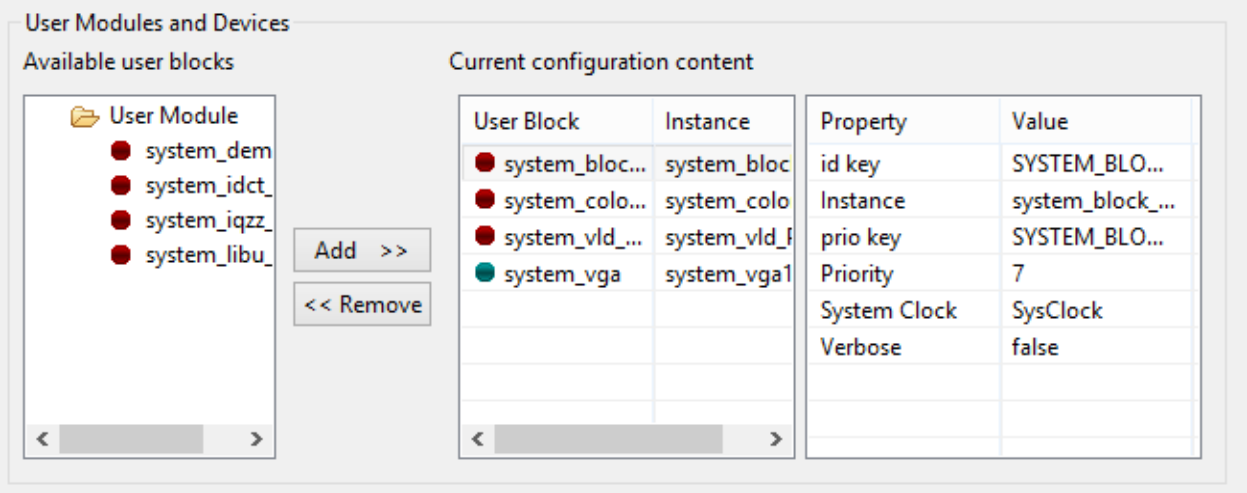

Figure 3-7 : Sélection des modules

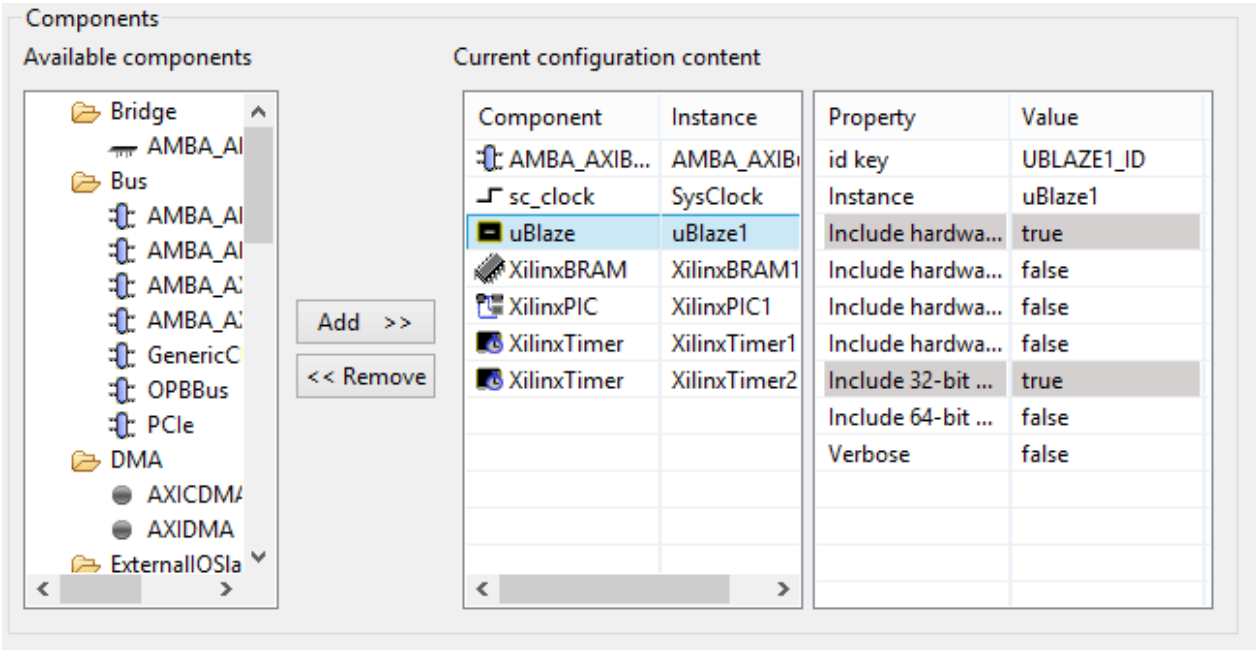

Figure 3-8 : Ajout de composants à la plateforme

3.2.2.2 Branchement de la plateforme

Une fois la plateforme peuplée de ses composants, l'utilisateur doit définir sur quel bus brancher les composants. C'est également à cette étape que l'on va choisir si un module est exécuté sur un processeur ou comme un composant matériel comme on peut le voir sur la figure 3-9 où les modules BLOCK_METRICS et DEMUX sont branchés sur le bus AMBA_AXIBus_LT1, et vont donc être

exécutés en matériel. Alors que les autres modules sont branchés au processeur uBlaze1 et seront donc exécutés sur ce dernier.

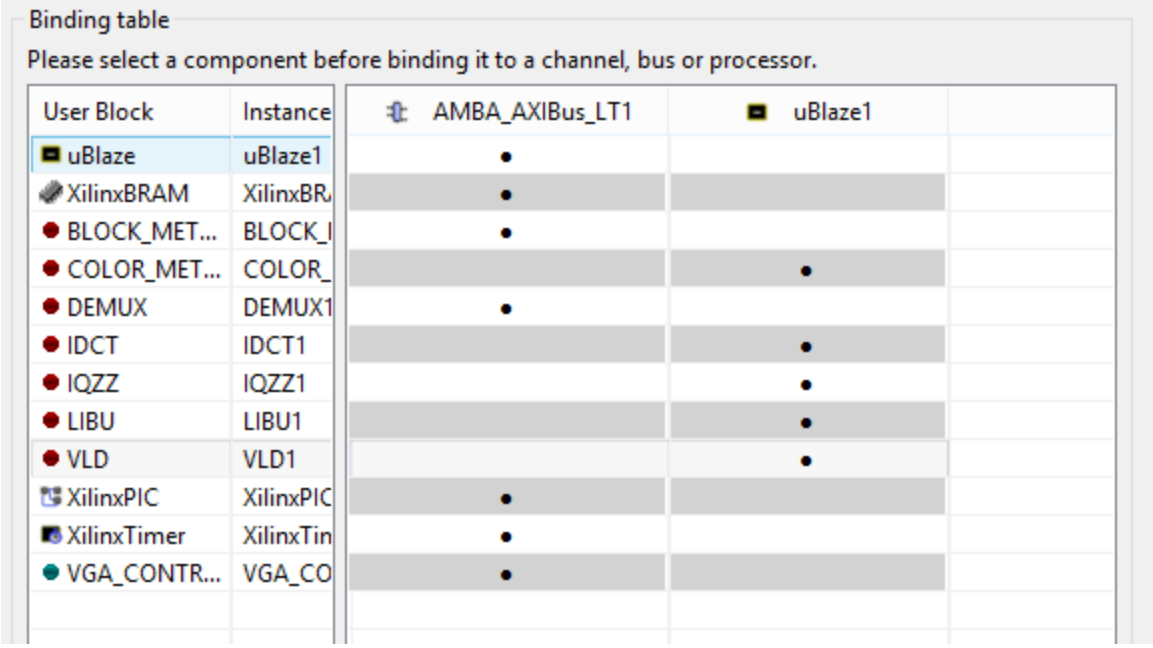

Figure 3-9 : Branchement de la plateforme

3.2.3 Test d'exécution

Une fois les modules codés et la plateforme en place, SpaceStudio peut générer un prototype virtuel du système. Avec ce prototype, il va être possible de procéder aux tests d'exécution du système.

3.2.3.1 Test fonctionnel

Grâce à ce test d'exécution, il est possible de vérifier que toutes les contraintes fonctionnelles du système sont satisfaites. Souvent, cette vérification fonctionnelle sera effectuée grâce à un banc de test qui aura été codé dans un device pour interagir avec le système.

3.2.3.2 Test non fonctionnel

L'outil SpaceStudio permet également de réaliser certaines analyses non fonctionnelles en marge du test fonctionnel. Il permet, entre autres, d'observer les accès mémoires qui ont eu lieu au cours de l'exécution, d'avoir accès à des statistiques sur l'utilisation des bus de communication ou encore d'analyser l'occupation d'un processeur par les différents threads qui tournent dessus comme sur la figure 3-10.

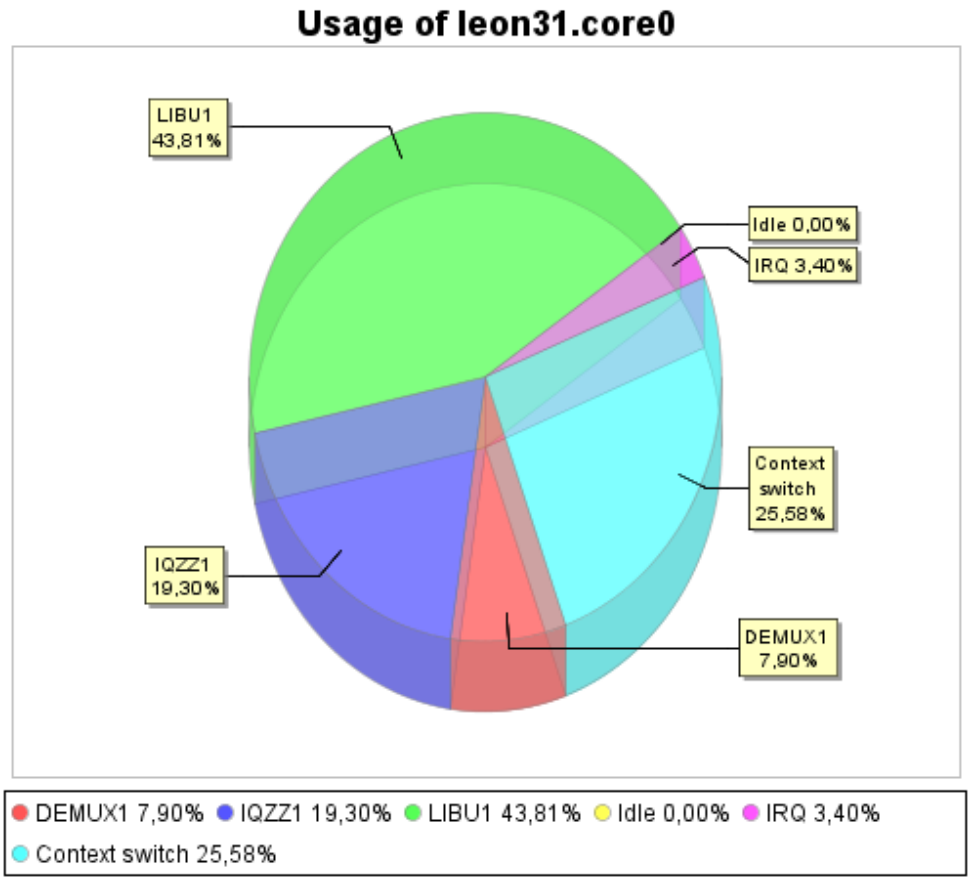

Figure 3-10 : Exemple de temps d'occupation d'un processeur

Ces évaluations de performances permettent d'analyser le système dans son ensemble (i.e. logiciel et matériel) pour valider un modèle haut niveau, et de se rendre compte de l'impact de certaines décisions prises sur l'architecture de celui-ci. Suite à ces premières analyses, il est possible d'aller modifier facilement l'architecture pour explorer d'autres configurations ou pour raffiner l'architecture actuelle.

Sur l'exemple de la figure 3-10, on pourrait essayer d'exécuter le module LIBU1 directement en matériel plutôt que de l'exécuter sur le processeur Leon3. Vu que ce module occupe une proportion importante du temps d'exécution, il peut devenir un candidat intéressant si on veut diminuer la charge de ce même processeur.

Ainsi grâce à cette exploration architecturale, l'utilisateur peut rapidement trouver la configuration la plus adaptée au système qu'il est en train de développer.

3.3 Ocarina

Ocarina [28] est un logiciel libre codé en ADA et disponible sur la plateforme de stockage « github ». Cet outil permet de réaliser une compilation du code AADL grâce à des analyses syntaxiques et lexicales qui forment la partie frontale (de l'anglais frontend) du logiciel. Ces analyses supportent la première et la seconde version d'AADL. À partir de là, Ocarina est capable de générer différents fichiers telle une description XML.

3.3.1 Création d'un arbre abstrait

Suite à l'analyse du code du modèle AADL, Ocarina génère un arbre abstrait qui regroupe toutes les informations du modèle dans une structure de données. C'est en parcourant cet arbre que l'outil est capable de générer facilement les fichiers demandés par l'utilisateur. Cet arbre est complètement générique, et est donc commun pour toutes les cibles proposées.

3.3.2 Cibles disponibles

Comme on l'a vu précédemment, Ocarina peut générer des fichiers pour différentes cibles à partir de l'arbre abstrait qu'il a créé. Les parties du logiciel permettant ces générations sont appelées « backend »[2]. Une vingtaine de backends sont actuellement mis à disposition par Ocarina comme la génération d'une représentation du modèle en XML ou encore d'un réseau de pétri.

[2] La meilleure traduction de backend est à notre avis *extrémité arrière*. Pour simplifier la lecture et faciliter le lien avec la terminologie AADL, nous resterons avec le terme *backend*.

CHAPITRE 4 GÉNÉRATION D'UNE SPÉCIFICATION EXÉCUTABLE À PARTIR DE AADL

Ce chapitre présente le principal développement de ce mémoire. Il s'agit d'un outil nommé AADL2Space qui se présente sous la forme d'un script Python. Ce script lance une analyse du code AADL par l'intermédiaire du logiciel Ocarina présenté précédemment. Grâce à ce logiciel, un script python faisant appel à l'API de création de projets de SpaceStudio est généré. Ce second script permet donc de créer un projet SpaceStudio et de mettre en place l'architecture complète du système (i.e. l'architecture logicielle ainsi que la plateforme matérielle). Dans un premier temps, il déclare les différents modules logiciels et leur associe des codes sources. Puis dans un second temps, il instancie tous les composants matériels et réalise le branchement des modules dessus. Les codes sources associés aux modules logiciels sont également créés grâce à Ocarina.

4.1 Ocarina

La majeure partie du projet consiste à ajouter une option à cet outil permettant de générer le script python de création de projet SpaceStudio ainsi que les codes sources des modules logiciels.

La modification apportée à Ocarina dans le cadre du projet consiste à ajouter notre propre backend à l'outil que l'on a appelé « aadl_spacestudio ». En plus de coder ce backend, il a fallu le brancher correctement au reste du logiciel.

4.1.1 Parcours de l'arbre abstrait et récupération d'informations

L'arbre abstrait généré par Ocarina est rempli d'informations utiles, et son parcourt nous permet de les récupérer. Ces informations sont de différents types.

4.1.1.1 Propriétés

L'un des principaux enjeux de ce projet est de pouvoir récupérer des propriétés définies pour des composants dans le modèle AADL pour les transposer dans notre projet SpaceStudio. Pour ce faire, on réalise un passage dans l'arbre abstrait en listant dans une table de hachage toutes les propriétés d'un composant. Ensuite, pour chaque type de composant on va chercher dans la table de hachage les propriétés qui nous intéressent pour les ajouter à la structure de notre composant.

4.1.1.1.1 Property set

Comme cela a été dit dans la section 3.1.2, AADL possède de nombreuses propriétés pouvant être assignées aux composants. Mais pour que la description du système puisse être portée vers un projet SpaceStudio, des propriétés spécifiques sont requises par la plateforme de codesign. C'est pourquoi un fichier « property set » a été écrit (disponible dans l'annexe A) permettant de définir ces différentes propriétés.

4.1.1.1.2 Utilisation des propriétés

Comme il a été discuté plus tôt, toutes les propriétés renseignées dans le modèle AADL ne nous intéressent pas forcément. De plus, les propriétés intéressantes ne s'utilisent pas toute de la même façon : certaines seront utilisées durant la création du projet SpaceStudio lors de l'implémentation des composants. D'autres comme les propriétés temps réel pour les threads seront implémenté dans les codes sources de ces composants. Et enfin, il y a les propriétés de mappage des threads sur les processeurs qui sont décrites dans la prochaine section et sont utilisées lors du branchement des composants.

4.1.1.2 Branchements de la plateforme matérielle

Il existe deux types de branchement dans un modèle AADL.

4.1.1.2.1 Mappage du logiciel sur les processeurs et les mémoires

Ces propriétés sont spéciales, car elles sont déclarées dans le système AADL (i.e. la plus grosse granularité du modèle AADL) indiquant qu'un groupe de processus, un processus, un groupe de thread, ou un thread doit s'exécuter sur tel ou tel processeur et/ou a besoin d'un accès à telle ou telle mémoire. Cependant, lors de la compilation du modèle, ces propriétés sont stockées dans l'arbre abstrait comme étant des propriétés du composant ciblé (groupe de processus, etc.).

Si ces propriétés sont appliquées à un processus alors celles-ci doivent être appliquées à tous ses threads. Cela pose problème, car ces propriétés ne sont pas propagées aux sous-composants. Or les seuls composants logiciels utilisés pour créer un projet SpaceStudio sont les threads. Heureusement, le parcours de l'arbre abstrait se fait de façon récursive (i.e. la visite d'un composant appelle la visite de ses sous-composants) nous permettant ainsi de propager ce type de propriétés pour l'appliquer correctement aux threads.

```
system implementation integration.ubimpl
subcomponents
      demux : process mjpeg::software::Demuxp.impl;
      vld : process mjpeg::software::VLDp.impl;
      iqzz : process mjpeg::software::IQZZp.impl;
      idct : process mjpeg::software::IDCTp.impl;
      libu : process mjpeg::software::LIBUp.impl;
      proc : processor mjpeg::platform::ub.impl;
      busAmba : bus mjpeg::platform::AMBA;
      mem : memory mjpeg::platform::BRAM;
properties
      Actual_Processor_Binding => (reference (proc)) applies to demux;
      Actual_Processor_Binding => (reference (proc)) applies to idct;
      Actual_Processor_Binding => (reference (proc)) applies to iqzz;
      Actual_Processor_Binding => (reference (proc)) applies to libu;
      Actual_Memory_Binding => (reference (mem)) applies to demux;
end integration.ubimpl;
```

Figure 4-1 : Exemple de mappage sur un processeur et une mémoire

4.1.1.2.2 Demandes d'accès à un bus

Le deuxième type de branchement à la plateforme matérielle se fait par l'intermédiaire de demandes d'accès. L'accès à un bus se déroule en deux étapes : dans un premier temps un composant requiert l'accès à un type de bus de la façon suivante :

```
nom_requete : requires bus access type_bus;
```

Cette ligne est à écrire dans les caractéristiques (features) du composant avec la déclaration de ses ports. Le champ type_bus est à remplacer par le nom d'un bus déclaré dans le modèle.

Dans un second temps, l'accès au composant sera confirmé dans la déclaration des connexions du système telle qu'illustrée à la figure 4-2 où le bus appelé Standard_Marine_Bus (2) est connecté aux différents composants matériels par l'intermédiaire des requêtes d'accès à ce bus faites dans les déclarations des composants (1).

```
device interface
features
      set_speed : out data port set_speed_value;
      disengage : out event port;
      BA1 : requires bus access Marine.interface;
end interface;

system implementation pba.speed
subcomponents
      speed_sensor : device sensor.speed;
      throttle : device actuator.speed;
      interface_unit : device interface.pilot;
      RT_1GHz : processor Real_Time.one_GHz;
      Standard_Marine_Bus : bus Marine;
      Stand_Memory : memory RAM.Standard;
connections
      BAC1 : bus access Standard_Marine_Bus <-> speed_sensor.BA1;
      BAC2 : bus access Standard_Marine_Bus <-> RT_1GHz.BA1;
      BAC3 : bus access Standard_Marine_Bus <-> throttle.BA1;
      BAC4 : bus access Standard_Marine_Bus <-> interface_unit.BA1;
      BAC5 : bus access Standard_Marine_Bus <-> Stand_Memory.BA1;
properties
      Allowed_Processor_Binding => (reference(RT_1GHz))
                  applies to speed_control.speed_control_laws;
      Allowed_Processor_Binding => (reference(RT_1GHz))
                  applies to speed_control.scale_speed_data;
      Actual_Memory_Binding => (reference(Stand_Memory))
                  applies to speed_control;
end pba.speed;
```

Figure 4-2 : Déclarations des accès à un bus

Ce sont ces connexions que l'on va analyser lors de notre parcours de l'arbre abstrait nous permettant d'enregistrer à quel bus ou quelle mémoire brancher nos composants.

Il est à noter que la présence d'au moins un bus dans un système est obligatoire pour un projet SpaceStudio. Le programme renverra un message d'erreur si un composant matériel ne possède pas d'accès à un bus.

4.1.1.3 Ports et connexions du logiciel

Le dernier type d'information qui nous intéresse définit l'architecture logicielle de notre système et donc les ports de communication des threads et leurs connexions. Le but est de savoir avec quel thread ou device communique le port d'un composant. Ce mappage des connexions demande un certain travail et sera utile dans la section 4.3.1.2.

Pour relier deux threads de deux processus différents, une communication doit suivre le chemin montré par la figure 4-3.

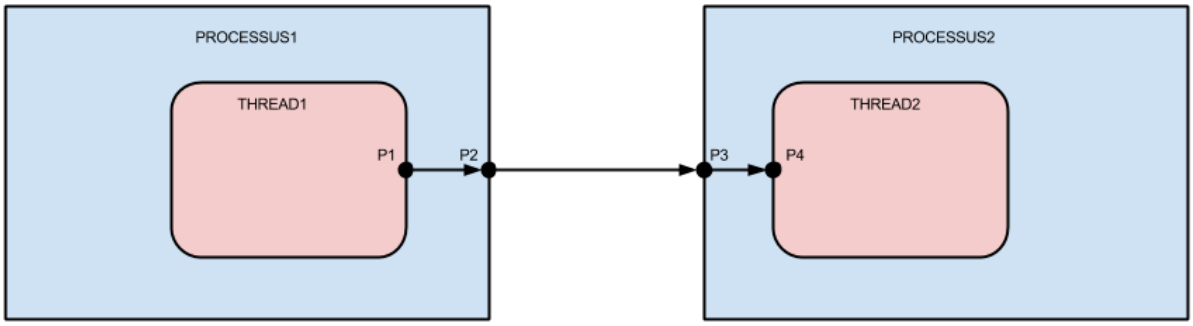

Figure 4-3 : Communication entre deux threads

La difficulté provient du fait que le ce chemin transite à travers plusieurs niveaux de granularité, ce qui rend difficile le mappage de ces connexions puisque le parcourt de l'arbre se fait de façon récursive. Pour réussir ce mappage, il a fallu utiliser deux tables de hachage permettant de stocker le début et la fin d'une chaine de connexion et de venir y coller des connexions élémentaires (i.e. entre deux ports adjacents comme le port P1 et le port P2 dans la figure 4-3).

Dans l'exemple de la figure 4-3, les connexions élémentaires vont apparaitre dans l'ordre suivant lors du parcours de l'arbre abstrait : P2 → P3, P1 → P3 puis P3 → P4.

On va donc, dans un premier temps, enregistrer la connexion P2 ➔ P3 dans nos tables de hachage, puis la connexion P1 ➔ P3 et enfin la connexion finale P1 ➔ P4.

4.1.2 Génération des fichiers pour SpaceStudio

Notre backend ne génère pas directement un projet SpaceStudio, mais un script python qui lui s'en chargera. Ce script intègre toutes les informations que l'on souhaite transmettre à la plateforme SpaceStudio.

En plus de ce script, le backend génère également les codes sources pour les modules logiciels du projet. Ces codes sources ne sont alors que des squelettes de code et ne sont donc pas fonctionnels. Cependant, ils permettent à l'utilisateur de rapidement coder le comportement logiciel du système une fois le projet ouvert sur SpaceStudio.

4.2 Création du script

Le script que l'on crée utilise l'API de création de projets de SpaceStudio et est destiné à être exécuté par le lanceur du logiciel grâce à la ligne de commande :

SpaceStudioLauncher.bat -script FULL_PATH_TO_SCRIPT

Ce script définit dans un premier temps, dans des tables, tous les composants de la plateforme ainsi que les modules et devices du système. On indique également dans ces tables différentes informations sur les composants et notamment le sous-type de composant comme indiqué au tableau 1. Ces sous-types sont des propriétés définies dans le « property set » de AADL décrit plus tôt et permettent, lors de l'exécution du script, d'instancier le bon type de composant (e.g. un processeur uBlaze).

Tableau 1 : Correspondance des composants entre AADL et SpaceStudio

Composants AADL	**Composants SpaceStudio / (Examples des sous types de composants)**
Bus	Bus / (AMBA_AHBBus, AMBA_APBBus, AMBA_AAXIBus_LT, AMBA_AXIBus, OPBBus)
Processor	Processor / (Xilinx MicroBlaze, Gaisler LEON3, ARM, MIPS, Freescale, Intel)
Memory	Memory / (GaislerOnChipRam, GaislerOnCHipRom, GaislerRam, LMBRAM, BRAM, DDR)
Thread	Module
Device	Device

Ensuite, ce même script va créer à proprement parlé le projet et le « peupler » avec les différents modules logiciels et les devices en leur associant à chacun, un fichier .cpp et un .h. Ces fichiers sont les codes sources qui ont étés générés par notre backend. Pour finir, il va instancier chaque composant dans le projet et les lier les uns aux autres suivant le modèle AADL.

```
project = create_project(PROJECT_NAME, temp_dir)
populate_project(project, MODULE_NAMES, DEVICE_NAMES, temp_dir + '\\' + PROJECT_NAME)
create_designs(project, PROJECT_NAME, MODULE_NAMES, DEVICE_NAMES, PROCESSOR_NAMES, BUS_NAMES,
MEMORY_NAMES)
```

Figure 4-4 : Trois étapes de création de projets du script python

4.2.1 Listing des composants

Pour pouvoir instancier correctement les composants du système, le script Python a besoin de connaitre les propriétés de ces composants. On crée donc, pour chaque type de composant, une table comportant les informations nécessaires. Ces tables sont explicitées dans l'annexe B.

4.2.2 Création du projet

La première étape du script est la création du projet. Cette étape consiste à fournir à l'API de SpaceStudio le nom du projet.

```
def create_project(project_name, base_dir):
	return projectEngine.createProject(project_name, base_dir)
```

Figure 4-5 : Création du projet

4.2.3 Instanciation des modules et devices

La prochaine étape du script consiste à peupler le projet ce qui signifie déclarer les modules qui vont faire partie du projet et leur assigner des codes sources. Ces codes sources ont été générés précédemment par notre outil. Cette génération automatique sera présentée à la section 4.3.

```
def populate_project(project, module_names, device_names, base_dir):
	path = base_dir + '\import\src'
	os.mkdir(path)
	shutil.copyfile('C:\workspace\SpaceStudioProject\deployement.h', path + '\deployement.h')
	shutil.copyfile('C:\workspace\SpaceStudioProject\deadline.h', path + '\deadline.h')
	shutil.copyfile('C:\workspace\SpaceStudioProject\deadline.cpp', path + '\deadline.cpp')
	for module_name in module_names:
		name = module_name[0]
		module = project.createModule(name)
		resource_dir = os.path.join(os.environ['SPACE_CODESIGN_ENV'],
'C:\workspace\SpaceStudioProject')
		shutil.copyfile(os.path.join(resource_dir, name + '.h'), module.getHeaderPath())
		shutil.copyfile(os.path.join(resource_dir, name + '.cpp'), module.getSourcePath())
	for device_name in device_names:
		name = device_name[0]
		device = project.createDevice(name, False)
```

Figure 4-6 : Peuplement du projet

4.2.4 Instanciation de la plateforme

La dernière étape consiste à créer tous les composants de la plateforme matérielle en parcourant les tables qui ont été présentées ci-dessus. Ces composants sont ensuite branchés correctement sur les bus de données susnommés dans les tables.

Une fois la plateforme matérielle correctement créée et branchée, les modules et les devices sont également branchés dessus. Les devices sont donc branchés au bus indiqué dans leur table. Les modules sont eux connectés sur le processeur donné dans leur table. Comme il est dit dans la section 4.2.1.1, si les modules n'ont pas de processeur associé, il est écrit 'hardware' dans leur table et seront alors branchés directement sur un bus pour être exécutés comme des modules matériels (plus précisément comme coprocesseurs).

```
def create_designs(project, name, module_names, device_names, processor_names, bus_names, memory_names):
        Design_name = name + '.design'
        bus = {}
        processor = {}
        design = project.createArchitecturalDesign(Design_name)
        for bus_name in bus_names :
                bus[bus_name[0]] = design.createComponentInstance('Bus',bus_name[1])
        Timer = design.createComponentInstance('Timer', 'XilinxTimer')
        Timer.connectTo(bus[bus_names[0][0]])
        for memory_name in memory_names :
                memory = design.createComponentInstance('Memory', memory_name[1])
                memory.connectTo(bus[memory_name[2]])
        for processor_name in processor_names :
                processor[processor_name[0]] = design.createProcessorInstance(bus[processor_name[2]],
processor_name[1])
        for module_name in module_names :
                module = design.createModuleInstance(module_name[0])
                if not module_name[1] == 'Hardware' :
                        module.mapTo(processor[module_name[1]])
                else :
                        module.mapTo(bus[bus_names[0][0]])
        for device_name in device_names:
                device = design.createDeviceInstance(device_name[0])
                device.connectTo(bus[device_name[1]])
        design.save()
```

Figure 4-7 : Création de la plateforme matérielle

4.3 Codes sources des modules

Les modules sont les composants logiciels de notre système et sont donc associés à un fichier .cpp et un fichier .h pour pouvoir exécuter notre système et faire toutes les vérifications fonctionnelles nécessaires.

Notre outil AADL2Space permet de générer automatiquement à partir du modèle AADL, ces fichiers .cpp et .h pour chaque module qui va servir de codes sources.

Dans un modèle AADL, un thread (alter ego du module SpaceStudio) peut être ou non associé à un code C. Si aucun code n'est associé à un thread, seul un squelette de code source sera alors créé permettant ainsi à l'utilisateur de coder facilement et rapidement son module à partir de la plateforme SpaceStudio en remplissant les espaces désignés.

Dans le cas contraire, il faut intégrer le code fourni dans le modèle AADL à notre code source destiné au module.

4.3.1 Environnement logiciel

L'un des grands intérêts de la modélisation en module faite par SpaceStudio est la généricité de ses fonctions de communications. En effet, du point de vue du code d'un module pour communiquer avec un autre module, on utilisera les fonctions ModuleRead et ModuleWrite. D'autre part, la communication entre un module et un device, sur SpaceStudio, se fait avec les fonctions DeviceRead et DeviceWrite.

Une des caractéristiques intéressantes de la plateforme SpaceStudio est d'offrir une description unique à un module. Plus précisément, dans ce module les appels aux primitives de communication sont les mêmes que le module soit exécuté comme module logiciel sur un processeur ou qu'il soit exécuté comme module matériel directement sur la plateforme. Selon le résultat de mappage (logiciel ou matériel), les primitives de communication procureront le comportement de communication approprié.

Le problème est que ces fonctions, comme présentées dans la section 3.2.1, utilisent comme argument l'identifiant du module avec lequel la communication est établie. Or dans un modèle AADL, un thread communique par l'intermédiaire de ses ports, mais n'a pas la visibilité de la globalité du système et ne connait donc pas l'identifiant du thread à l'autre bout de la communication. Pour la même raison, un thread ne sait pas s'il doit communiquer avec un autre thread (on aurait alors une communication module à module) ou avec un device (on aurait alors une communication de module à device).

Pour pallier ce problème, il a fallu rendre ces fonctions de lecture et d'écriture encore plus génériques. Un fichier « communication.h » a donc été créé permettant de définir une fonction read et une fonction write qui prennent comme argument le port sur lequel la lecture ou l'écriture doit être effectuée.

Un fichier « deployement.h » est également créé. Ce fichier permet de stocker tous les noms des ports de communication du système et d'associer à chacun de ces ports, l'identifiant du module ou du device avec lequel celui-ci va communiquer.

Ces deux fichiers seront générés pour chaque projet en même temps que les codes sources des différents modules. Les détails de ces deux fichiers sont disponibles aux annexes C et D.

4.3.2 Squelette

Les squelettes de code qui sont créés par notre backend restent très simples, mais permettent à l'utilisateur de s'affranchir de plusieurs particularités de SpaceStudio, par exemple, toutes les inclusions nécessaires au fonctionnement des modules ou encore tous les constructeurs des modules tel qu'illustré à la figure 4-8.

```
////////////////////////////////////////////////
///////////////   Constructor   ////////////////
////////////////////////////////////////////////
system_idct_P4::system_idct_P4(sc_module_name zName, double dClockPeriod,
sc_time_unit ClockPeriodUnit, unsigned char ucID, unsigned char ucPriority, bool
bVerbose)
: SpaceBaseModule(zName, dClockPeriod, ClockPeriodUnit, ucID, ucPriority,
bVerbose)
{
      SC_THREAD(thread);
}
```

Figure 4-8 : Constructeur d'un module

Ce constructeur est identique pour chaque module et sera appelé lors de l'exécution du projet. Le fichier qui effectue les appels aux différents constructeurs des modules est généré automatiquement lors de la compilation du projet.

En plus du constructeur du module, une fonction « thread » est également générée automatiquement. Cette fonction va organiser l'exécution du thread à l'intérieur d'une boucle infinie. À l'intérieur de cette boucle seront vérifiées les propriétés temps réel qui ont été assignées au module. C'est donc à l'intérieur de cette boucle que sera exécuté le code à proprement parler du thread.

```
while(1)
{
	DeviceWrite(TIMER1_ID, offset, &initValue);

	//Execution between 10 and 40 ms

	operation();

	DeviceRead(TIMER1_ID, offset, &timerValue);
	if (timerValue < 50 ms)
		wait(50 ms - timerValue); // Wait minimum latency
	else
		sc_stop(); //Miss Deadline
}
```

Figure 4-9 : Exemple de boucle infinie d'un thread

La figure 4-9 montre un exemple de boucle pour un thread périodique s'exécutant entre 10 et 40 ms à une période de 50 ms.

Un fichier .h est également créé pour déclarer les entêtes des fonctions présentées ci-dessus et pour mettre en place la structure de classe des modules.

4.3.3 Intégration d'un code C

On a vu dans la section 3.1.1.1 qu'il est également possible d'assigner des fonctions codées en C à un thread, en y branchant un sous-programme. Cette assignation de code est néanmoins soumise à quelques conditions qui sont explicitées dans la section suivante.

L'intégration de ces codes C aux squelettes se fait en deux temps : dans un premier temps, notre backend va créer un fichier texte qui va répertorier les associations entre les modules et les fonctions en C. Dans un second temps, on va exécuter un script Python qui va parcourir le fichier texte pour intégrer correctement les fonctions aux codes des modules.

4.3.3.1 Mappage des fonctions

Le fichier texte créé par notre backend s'appelle « map_function.txt ». Les noms de tous les modules devant intégrer une fonction C sont inscrits dans ce fichier, ainsi que les noms de ces fonctions et le fichier source où se trouve cette fonction. Il s'organise de la manière suivante :

```
MODULE_1
      before mod1_funct1 source_mod1.cpp
      loop mod1_funct2 source_mod1.cpp
MODULE_2
      loop mod2_funct source_mod2.cpp
…
MODULE_n
      loop modn_funct source_modn.cpp
```

4.3.3.2 Script d'intégration

Un script python a donc été mis au point pour pouvoir intégrer les différentes fonctions C aux codes des modules. Il se décompose en 3 étapes.

4.3.3.2.1 Récupération des fichiers sources

Dans cette première étape, le script va parcourir le fichier « map_function » pour récupérer les noms des modules et ceux des fichiers sources correspondants.

4.3.3.2.2 Modification du fichier NOM_MODULE.cpp

Le nom du code source du module (fichier 1) et celui du fichier source (fichier 2) sont fournis par la première étape. À partir de là, on parcourt ces fichiers pour réaliser les actions suivantes :

- Intégration des inclusions et définitions du fichier 2 avec celles du fichier 1.
- Écriture de l'appel de la fonction C au milieu de la boucle infinie du thread du fichier 1.
- Récupération des corps des fonctions du fichier 2 pour les importer dans le fichier 1.
- Modification des entêtes de ces fonctions pour y introduire le nom de la classe (i.e. le nom du module). Par exemple, la fonction `void do(int i)` va devenir `void NOM_MODULE::do(int i).`

4.3.3.2.3 Modification du fichier NOM_MODULE.h

Là encore, le fichier NOM_MODULE.h (fichier 3) et le fichier source .h (fichier 4) sont donnés par la première étape. Pour ces fichiers, le script doit réaliser les actions suivantes :

- Intégration des inclusions et définitions du fichier 2 avec celles du fichier 1.

- Récupération des entêtes de toutes les fonctions du fichier 4 pour les importer dans la partie « private » de la classe.

4.3.4 Contraintes

Afin de pouvoir assigner du code C à un modèle AADL, l'utilisateur doit respecter quelques règles, sans lesquelles, les scripts Python ne seront pas capables de l'intégrer au code source des différents modules SpaceStudio.

Ces règles ont donc été listées sous la forme des différentes contraintes suivantes :

- L'utilisateur ne doit déclarer au maximum qu'un seul paramètre de sortie pour un sous-programme.
- L'utilisateur a le choix d'exécuter un sous-programme avant la boucle infinie du thread ou à l'intérieur. Dans tous les cas, il doit indiquer dans le modèle AADL à quel endroit du thread va s'exécuter le sous-programme à l'aide des mots clés « before » (pour que le sous-programme s'exécute avant la boucle) et « loop » (pour que le sous-programme s'exécute dans la boucle) comme à la figure 4-10.

```
calls
      loop : {code : subprogram IQZZ_Function;};
      before : {code1 : subprogram IQZZ_Before;};
```

Figure 4-10 : Exemple d'appels de sous-programmes

- Le modèle AADL permet de renseigner des paramètres d'entrée et un paramètre de sortie pour un sous-programme. L'envoi et la réception de ces paramètres sont générés automatiquement par l'outil. Cependant, il se peut qu'à l'intérieur même d'une fonction, un module ait besoin d'effectuer des communications avec les autres composants du système. L'utilisateur a donc la responsabilité d'effectuer ces communications avec les fonctions Read et Write décrites dans la section 4.3.1.1.

4.4 Validation

Pour valider l'outil qui a été conçu, plusieurs configurations particulières d'un modèle AADL ont été testées. On a choisi un modèle déjà existant et disponible sur Internet (https://wiki.sei.cmu.edu/aadl/index.php/SpeedRegulation) pour montrer que l'outil développé s'adapte bien à un modèle AADL non conçu spécifiquement pour être porté sur la plateforme SpaceStudio (modèle AADL disponible en annexe). Plusieurs cas particuliers sont donc observés.

4.4.1 Type de données non renseigné

Le modèle n'étant pas conçu pour être transposé en projet SpaceStudio, les types de données avaient de fortes chances de ne pas convenir au requis de notre outil, et effectivement comme on peut voir sur la figure 4-11, un message d'erreur est envoyé.

```
+======================== OCARINA BUG DETECTED =======================+
| Detected exception: SYSTEM.ASSERTIONS.ASSERT_FAILURE                 |
| Error: No C type found for picture data with Data_Model::Representation property |
| Please refer to the User's Guide for more details.                   |
+======================================================================+
```

Figure 4-11 : Erreur sur un type de données

4.4.2 Bus absent

Un autre test consiste à ne pas implémenter de bus dans le modèle. Il a été mentionné dans la section 4.4.2 que si aucun bus n'est implémenté, l'outil retourne un message d'erreur. On peut voir également à la figure 4-12 que si un composant n'est pas relié à un bus, un message d'erreur est également envoyé.

```
+======================== OCARINA BUG DETECTED =======================+
| Detected exception: SYSTEM.ASSERTIONS.ASSERT_FAILURE                 |
| Error: ecu1 is not bound to a bus                                    |
| Please refer to the User's Guide for more details.                   |
+======================================================================+
```

Figure 4-12 : Erreur de branchement d'un composant

4.4.3 Composants sans type

Comme le modèle AADL n'avait pas pour but d'être porté sur SpaceStudio, les propriétés de sous-type des composants ne sont pas renseignées et on peut voir que l'outil leur assigne quand même un type par défaut.

```
PROJECT_NAME = 'integration.implementation2'
BUS_NAMES = [('bus1','AMBA_AXIBus'), ('bus2','AMBA_AXIBus')]
MEMORY_NAMES = []
# processor (processor_name, processor_type, binding_bus, scheduling_protocol)
PROCESSOR_NAMES = [('ecu1','uBlaze','bus1','round_robin'), ('ecu2','uBlaze','bus2','round_robin'),
('ecu3','uBlaze','bus1','round_robin')]
```

Figure 4-13 : Types par défaut assignés aux composants

4.4.4 Plusieurs bus

Si l'on répartit le branchement des composants sur plusieurs bus, on peut voir à la figure 4-13 que cette répartition est bien répercutée dans le script Python et a posteriori dans la configuration SpaceStudio tel qu'illustré aux figures 4-14 et 4-15.

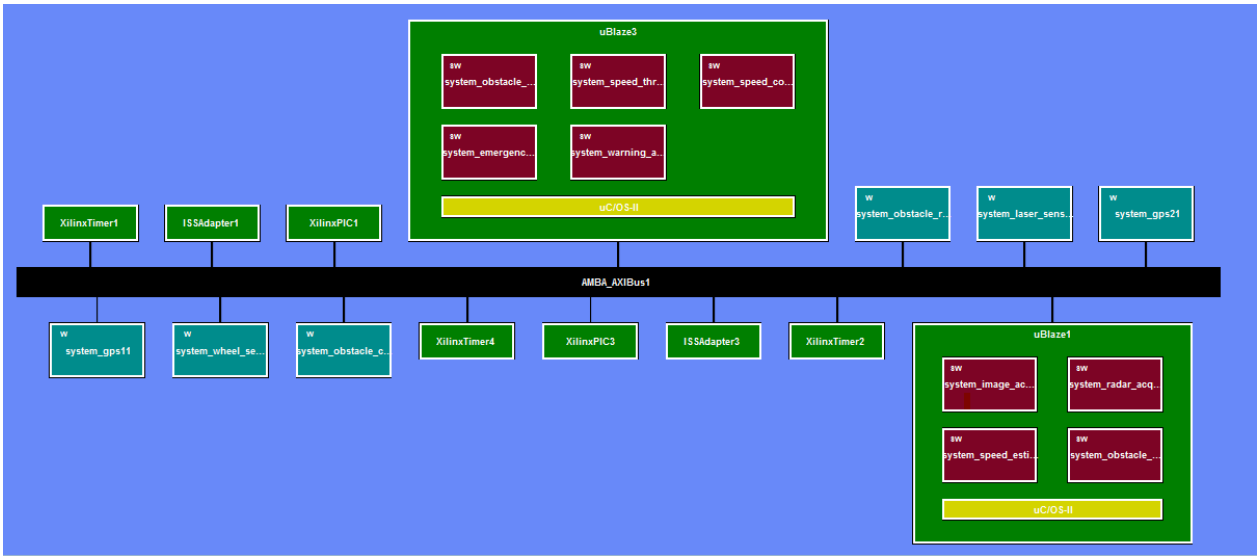

Figure 4-14 : Premier bus de la configuration

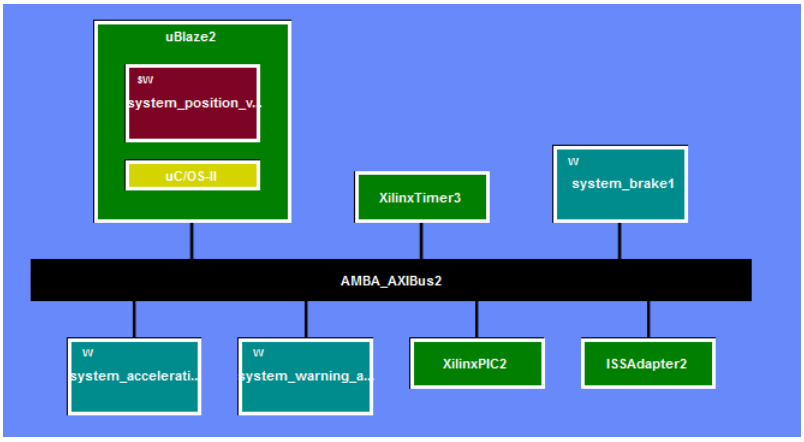

Figure 4-15 : Second bus de la configuration

4.4.5 Plusieurs processeurs

De la même manière, une répartition des modules sur plusieurs processeurs est possible.

```
MODULE_NAMES = [('system_image_acquisition_thr_acq','ecu1',18,'Periodic',50),
('system_radar_acquisition_thr','ecu1',18,'Periodic',10),
('system_speed_estimate_thr','ecu1',18,'Periodic',8),
('system_position_voter_thr','ecu2',18,'Periodic',12),
('system_obstacle_detection_thr','ecu1',18,'Periodic',100),
('system_obstacle_distance_evaluation_thr','ecu3',18,'Periodic',10),
('system_speed_threshold_calculation_thr','ecu3',18,'Periodic',4),
('system_speed_controller_thr','ecu3',18,'Periodic',5),
('system_emergency_detection_thr','ecu3',18,'Periodic',4),
('system_warning_activation_thr','ecu3',18,'Periodic',2)]
```

Figure 4-16 : Branchement des modules sur plusieurs processeurs

CHAPITRE 5 ÉTUDE DE CAS

Ce chapitre présente une étude de cas suivant le flot de conception proposé à travers une application de traitement vidéo MJPEG faisant partie d'un système de vidéo en vignette[3] qui ne nécessite pas une grande vitesse de lecture. On utilise le flot de conception présenté dans ce travail et qui comporte les étapes suivantes: description du modèle en AADL, traduction de AADL à SpaceStudio via AADL2Space, test fonctionnel et non fonctionnel du projet et raffinement de l'architecture.

5.1.1 Description de l'application

L'application qui nous sert d'exemple est fournie par la compagnie SpaceCodesign. Celle-ci est une application multimédia MJPEG (Motion JPEG) pour le décodage d'un flux vidéo en images JPEG. La figure 5-1 présente le diagramme fonctionnel de cette application.

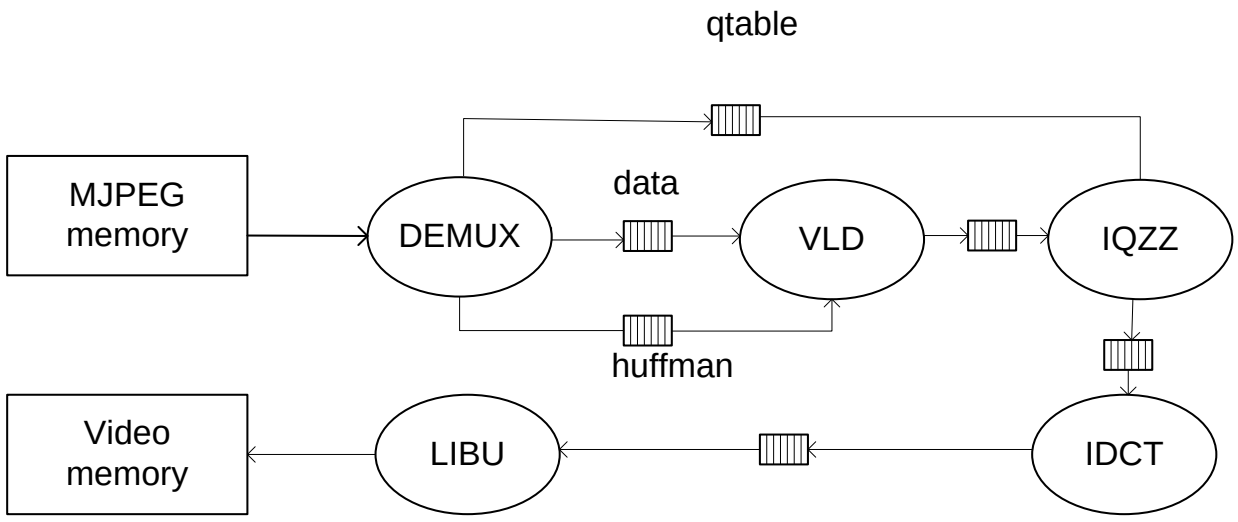

Figure 5-1: Diagramme fonctionnel de l'application

[3] De l'anglais thumbnail

Les fonctionnalités des différents blocs sont décrites ci-dessous :

- MJPEG memory : cette mémoire contient le fichier MJPEG qui doit être décodé ainsi que les marqueurs qui seront transmis au module demux contenant l'information nécessaire pour décoder correctement le fichier.
- DEMUX : ce module s'occupe de lire le contenu de la mémoire puis analyse les marqueurs qui délimitent les sections du fichier à décoder avant d'envoyer ces informations à tous les modules du système. Ensuite, ce module a pour objectif le démultiplexage du flux de données pour envoyer différentes tables aux autres modules : la table de dé-quantification (*qtable*) au module IQZZ, la table de Huffman et les données du fichier au module VLD.
- VLD : ce module se charge de la décompression du flux compressé. Il utilise les tables de Huffman pour décoder les données puis envoie le résultat du décodage au bloc IQZZ.
- IQZZ : ce module s'occupe de la dé-quantification. Ensuite, les résultats obtenus sont transférés au bloc IDCT.
- IDCT : ce module applique une transformée cosinus inverse sur les données reçues et envoie le résultat au module LIBU.
- LIBU : ce module construit des lignes complètes d'images à partir des données reçues de l'IDCT et envoie ensuite les pixels vers la mémoire vidéo.

5.1.2 Modèle AADL

La première étape de notre flot de conception a donc été de modéliser la description du système qui a été faite précédemment à l'aide du langage AADL. La figure 5-2 montre une vue graphique du modèle qui a été réalisé (le modèle complet est disponible en annexe).

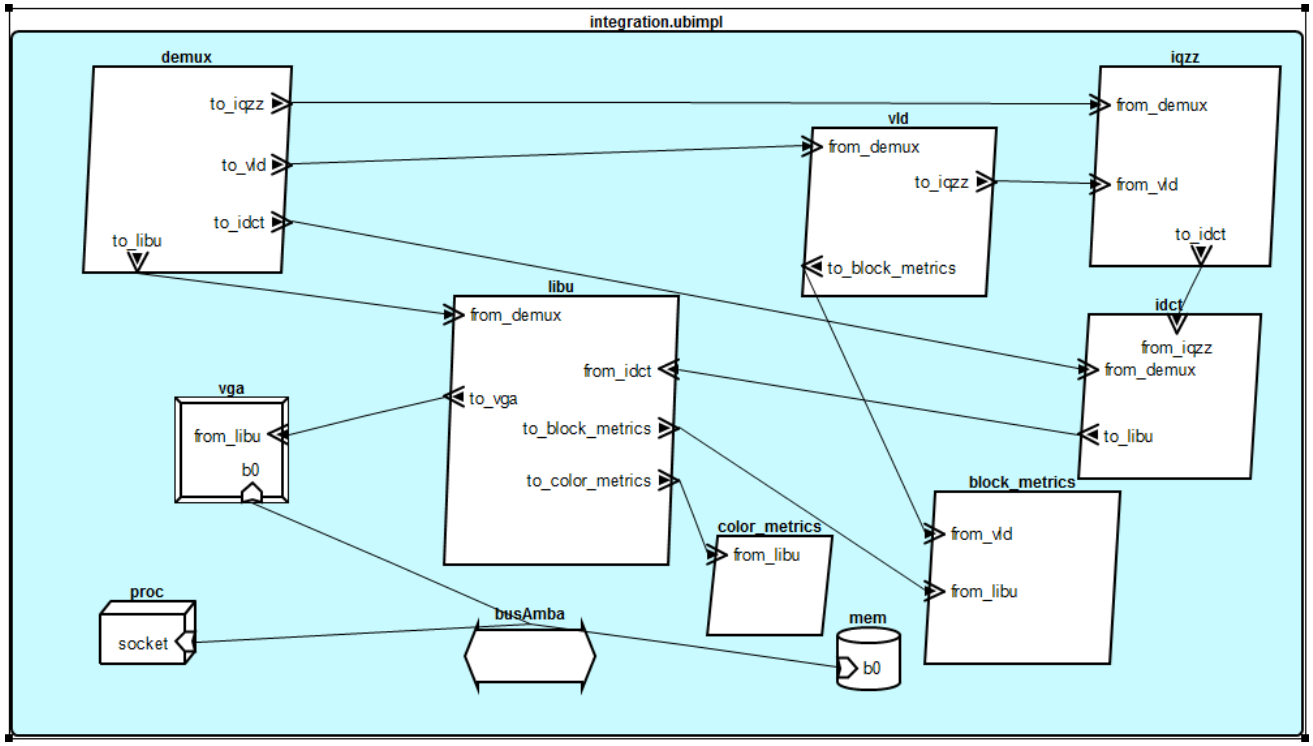

Figure 5-2 : Modèle AADL de l'application MJPEG

Deux modules ont été ajoutés au diagramme fonctionnel : un module block_metrics et un autre color_metrics. Ces deux modules servent à afficher des informations sur le flux de données durant l'exécution.

À chacun des threads du modèle est associé un sous-programme qu'il faudra donc intégrer aux codes sources des modules.

Un processeur « uBlaze » et un processeur « ARM Cortex A9 » (abrégé par A9 dans la suite) sont utilisés ainsi qu'un bus « AMBA_AXIBus » et une mémoire « XilinxBRAM »

Pour cet exemple, on a choisi de faire s'exécuter les threads vld, idct, et iqzz sur le processeur A9 et le thread sur le processeur uBlaze alors que les autres threads seront exécutés comme des modules matériels.

```
properties
      Actual_Processor_Binding => (reference (uBlaze)) applies to demux;
      Actual_Processor_Binding => (reference (a9)) applies to idct;
      Actual_Processor_Binding => (reference (a9)) applies to iqzz;
      Actual_Processor_Binding => (reference (a9)) applies to vld;
      Actual_Memory_Binding => (reference (mem)) applies to demux;
```

Figure 5-3 : Mappage des threads sur la plateforme matérielle

5.1.3 Utilisation de l'outil créé

Lors de l'exécution de l'outil, on génère les fichiers suivants : le script python de création de fichiers, le fichier communication.h et le fichier deployement.h pour les communications logicielles entre les modules, les squelettes de codes sources (.cpp et .h) des modules du système, et le fichier map_function.txt.

5.1.3.1 Script Python

Une grande partie du script Python généré est identique pour tous les modèles AADL et a été présentée dans la section 4.2. On peut voir sur la figure 5-4, la partie spécifique à ce modèle.

```
PROJECT_NAME = 'integration.DemuxLibuUb_IdctA9'
BUS_NAMES = [('busAmba','AMBA_APBBUS')]
MEMORY_NAMES = [('BRAM','XilinxBRAM','busAmba')]

# processor (processor_name, processor_type, binding_bus, scheduling_protocol)
PROCESSOR_NAMES = [('proc1','armCortexA9','busAmba','round_robin')]#,
('proc2','uBlaze','busAmba','round_robin')]

# module (module_name, binding_processor, priority, dispatch_protocol, period)
MODULE_NAMES = [('system_demux_P1','proc1',18,'Periodic',50),
('system_vld_P2','Hardware',18,'Periodic',50), ('system_iqzz_P3','Hardware',18,'Periodic',50),
('system_idct_P4','proc1',18,'Periodic',50), ('system_libu_P5','proc1',18,'Periodic',50),
('system_block_metrics_P6','Hardware',18,'Periodic',50),
('system_color_metrics_P7','Hardware',18,'Periodic',50)]

DEVICE_NAMES = [('system_vga','busAmba')]
```

Figure 5-4 : Script python généré

On remarque ici les composants matériels du système qui indiquent qu'ils vont venir se brancher au bus « busAmba ». On remarque également les modules iqzz, demux et idct qui indiquent le processeur « proc » comme plateforme d'exécution. Il est à noter que la table MODULE_NAMES est déclarée sur une seule ligne dans le script python, mais a été ramenée sur quatre lignes ici pour plus de clarté.

5.1.3.2 Communication.h

Ce fichier est identique pour tous les modèles AADL, car il instancie des fonctions complètement génériques. Ce fichier a été décrit dans la section 4.3.1.

5.1.3.3 Deployement.h

La figure 5-5 représente une partie tronquée du fichier deployement.h (le fichier complet est disponible en annexe), mais elle permet de constater que les ports du système sont associés chacun à un module ou device déclaré dans la structure « enum ». On constate également que le tableau contient bien un 0 dans la case correspondant au VGA qui est bien un device et non un module.

```
enum MODULES_DEVICES {
system_demux_P1,
system_vld_P2,
system_iqzz_P3,
system_idct_P4,
system_libu_P5,
system_block_metrics_P6,
system_color_metrics_P7,
system_vga
};

#define system_demux_P1_to_vld system_vld_P2
#define system_demux_P1_to_iqzz system_iqzz_P3
#define system_demux_P1_to_idct system_idct_P4
#define system_demux_P1_to_libu system_libu_P5
#define system_vld_P2_from_demux system_demux_P1
#define system_vld_P2_to_iqzz system_iqzz_P3
#define system_vga_from_libu system_libu_P5

bool isModule[8] = {1, 1, 1, 1, 1, 1, 1, 0};
```

Figure 5-5 : Fichier deployement.h généré

5.1.3.4 Fichier map_function.txt

La figure 5-6 montre qu'une fonction C++ a été associée à chaque thread du modèle AADL. On peut voir également que le thread iqzz aura une fonction à exécuter avant de rentrer dans sa boucle infinie.

```
system_demux_P1
	loop demux_funct DEMUX.cpp
system_vld_P2
	loop vld_funct VLD.cpp
system_iqzz_P3
	before before_iqzz IQZZ.cpp
	loop iqzz_funct IQZZ.cpp
system_idct_P4
	loop idct_funct IDCT.cpp
system_libu_P5
	loop libu_funct LIBU.cpp
system_block_metrics_P6
	loop metrics BLOCK_METRICS.cpp
system_color_metrics_P7
	loop color COLOR_METRICS.cpp
```

Figure 5-6 : Fichier map_function.txt généré

5.1.4 Intégration des codes C++ aux modules

Une fois que notre backend a fini de générer les fichiers, on lance le script présenté à la section 4.3.3. pour intégrer toutes les fonctions C++ indiquées dans le fichier map_function.txt.

```
void system_iqzz_P3::thread(void) // Can be either SW or HW
{
	unsigned int initValue;
	unsigned int timerValue;
	unsigned int nbQuant2;
	unsigned int nbQuant;
	unsigned int uiCommand;

	Read(system_iqzz_P3_from_demux, &nbQuant2);
	nbQuant = before_iqzz(nbQuant2);

	while(1)
	{
		DeviceWrite(XILINXTIMER1_ID, SPACE_NON_BLOCKING, &initValue);
		Read(system_iqzz_P3_from_demux, SPACE_WAIT_FOREVER, &uiCommand);

	iqzz_funct(uiCommand, nbQuant);
```

Figure 5-7 : Fonction thread du module iqzz

5.1.5 Création du projet SpaceStudio

Une fois qu'on a généré tous les fichiers nécessaires, on lance l'exécution du script pour créer le projet SpaceStudio. On peut ensuite ouvrir le projet sur la plateforme SpaceStudio qui nous informe qu'une configuration est disponible. La figure 5-9 montre le cheminent de l'information effectué pour implémenter un processeur sur la plateforme, alors que la figure 5-8 montre celui réalisé pour y brancher un thread lors de l'implémentation sur SpaceStudio.

AADL

```
thread Demux
features
	to_vld : out event data port Uint_data;
	to_iqzz : out event data port Uint_data;
	to_idct : out event data port Uint_data;
	to_libu : out event data port Uint_data;
end Demux;
thread implementation Demux.impl
calls
	loop : {code : subprogram Demux_Function;};
properties
	Period => 50ms
	Dispatch_Protocol => Periodic;
end Demux.impl;
```

Python

```
MODULE_DEMUX = ('system_demux_P1','proc',18,'Periodic',50)
module = project.createModule(name)
shutil.copyfile(os.path.join(resource_dir, name + '.h'), module.getHeaderPath())
shutil.copyfile(os.path.join(resource_dir, name + '.cpp'), module.getSourcePath())
moduleInstance = design.createModuleInstance(MODULE_DEMUX[0])
moduleInstance.mapTo(processor[MODULE_DEMUX[1]])
```

SpaceStudio

```
void system_demux_P1::thread(void)
{
	Double initValue;
	Double timerValue;
	while(1)
	{
		DeviceWrite(XILINXTIMER1_ID, offset, &initValue);

		Demux_Function();

		DeviceRead(XILINXTIMER1_ID, offset, &timerValue);
		if (timerValue <50)
			wait(50 - timerValue);
		else
			sc_stop() ;
	}
}
```

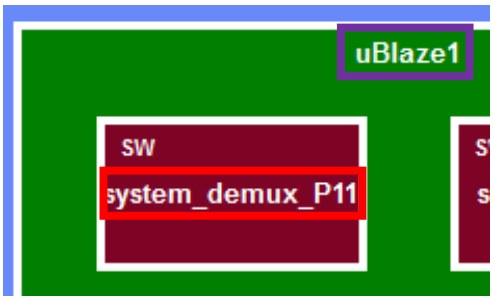

Figure 5-8 : Branchement du thread DEMUX

AADL

```
processor ub
features
       socket : requires bus access AMBA;
end ub;
processor implementation ub.impl
properties
       SpaceStudio::Subtype_component => uBlaze;
end ub.impl;
```

Python

```
PROCESSOR = ('proc', 'uBlaze' 'busAmba')
processor[PROCESSOR[0]] = design.createProcessorInstance(bus[processor_name[2]], processor_name[1])
```

SpaceStudio

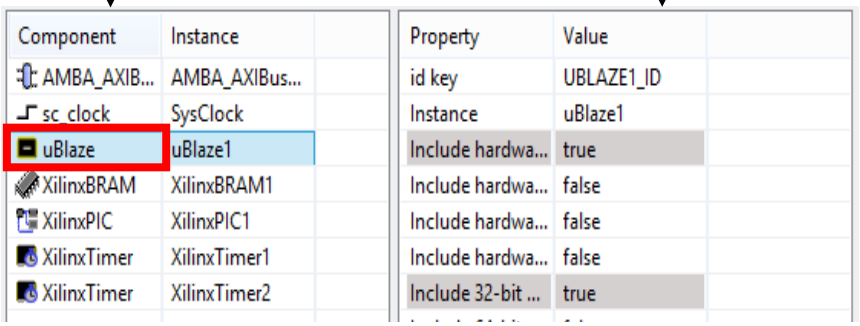

Figure 5-9 : Branchement d'un processeur MicroBlaze

La figure 5-10 est une représentation graphique de la configuration. On peut vérifier que tous les composants ont été branchés correctement.

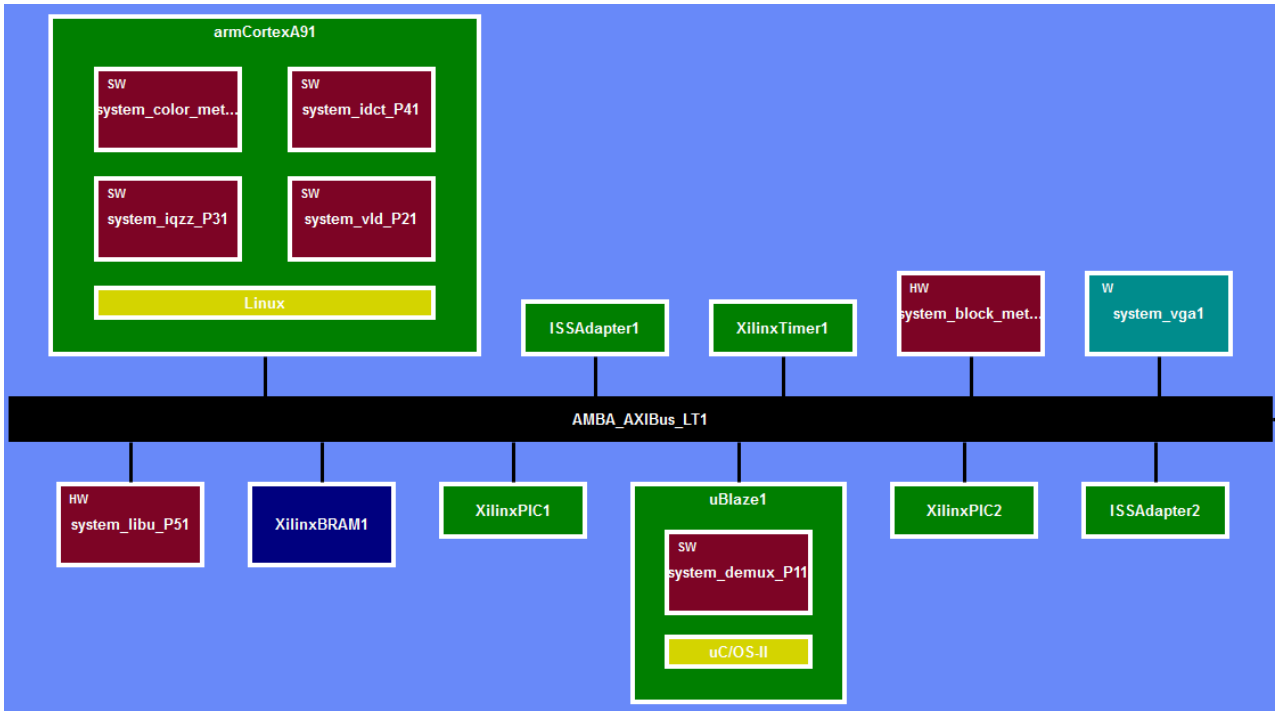

Figure 5-10 : Configuration du projet SpaceStudio

Il reste ensuite à écrire le code du device VGA, car malgré le fait que le langage AADL permette d'assigner du code source à un device, nous avons fait le choix de nous concentrer sur la génération automatique des modules et de ne pas implémenter celle des devices. Ici, le code du VGA a été récupéré dans les fichiers mis à disposition par la compagnie SpaceCodesign.

5.1.6 Exécution et analyse de la configuration

Une fois que le projet a été entièrement porté sur SpaceStudio, la plateforme permet de réaliser une cosimulation de l'application logicielle et de la plateforme matérielle. Les détails de cette simulation sont donnés dans le chapitre suivant.

5.1.7 Synthèse

Pour compléter le flot de conception, il faut réaliser la synthèse du système au niveau RTL pour pouvoir l'implémenter sur carte. Cette synthèse permet également de se renseigner sur les ressources matérielles dont a besoin le système pour compléter les métriques utiles à l'exploration architecturale explicitée dans le chapitre suivant.

Pour réaliser cette synthèse, la plateforme SpaceStudio permet d'appeler les synthétiseurs de différents fabricants :

- HLS Vivado pour Xilinx, et
- Calypto Catapult.

Notez qu'il est également possible de générer un SystemC natif.

CHAPITRE 6 RÉSULTATS

6.1.1 Analyse de l'architecture

Une fois le projet compilé, celui-ci peut être exécuté. Avec la première configuration provenant du flot, les résultats de la cosimulation indiquent une vitesse de décodage vidéo de 2,5 fps (images par seconde).

Pour observer les ressources matérielles nécessaires au système, on le synthétise avec comme cible, une carte FPGA Xilinx Zynq-7000. La configuration utilise environ 30% des ressources matérielles de la carte.

Cette configuration illustre une architecture qui demande peu de ressources matérielles, mais dont le temps d'exécution indique une performance très basse en termes d'images par seconde. Elle pourrait convenir à une classe d'application (système de vision pour un robot qui se déplace à très faible vitesse) mais pour d'autres types d'applications, on souhaiterait avoir de meilleures performances.

On peut alors voir sur la figure 6-1 que le processeur ARM cortex A9 est utilisé à environ 66% avec le module vld qui occupe le plus les deux cœurs (presque 37% d'occupation).

Une piste d'amélioration consiste alors à essayer de faire s'exécuter le module vld en tant que composant matériel.

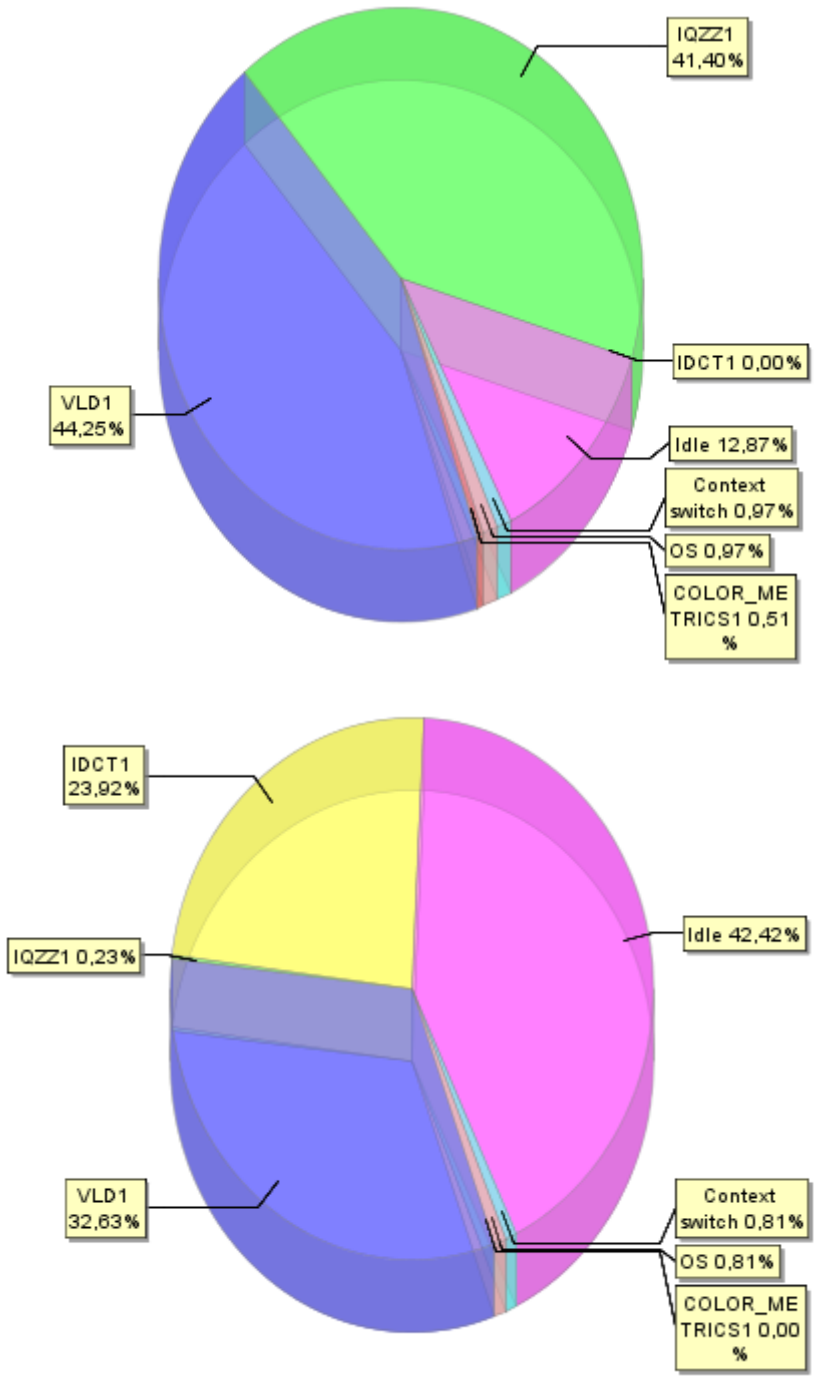

Figure 6-1 : Taux d'utilisation des cœurs du processeur A9

6.1.2 Raffinement de l'architecture

Pour la suite de ce travail nous avons établi les contraintes suivantes en nous appuyant sur les besoins de l'application de vidéo en vignette :

- il a été décidé d'utiliser une combinaison de deux processeurs : un processeur ARM cortex A9 et un uBlaze,
- Pour les besoins du reste de l'application vidéo, le taux d'occupation du processeur dual-core ARM cortex A9 par le décodage MJPEG ne doit pas dépasser 10%,
- la vitesse de décompression doit être comprise entre 4 et 6 fps pour rester dans la gamme de l'application, et
- le système ne doit pas utiliser plus de 40% des ressources matérielles.

L'amélioration d'une contrainte entraine la détérioration des autres et l'exploration architecturale permet de trouver un équilibre de ces contraintes et d'identifier l'architecture la plus adaptée. Les différentes architectures qui ont été essayées sont présentées dans la table 3.

Tableau 2 : Cinq architectures HW/SW vérifiées

Architectures	**Branchements logiciels**		**Branchements matériels**
	ARM	**uBlaze**	
1	IDCT/IQZZ/VLD	DEMUX	LIBU
2	IDCT	DEMUX	IQZZ/VLD/LIBU
3	IQZZ	DEMUX	IDCT/VLD/LIBU
4	VLD	DEMUX	IDCT/IQZZ/LIBU
5	IDCT	DEMUX/LIBU	IQZZ/VLD

Les tableaux 4 et 5 présentent les performances des différentes configurations ainsi que les ressources matérielles qu'elles nécessitent :

Tableau 3 : Performances des cinq architectures

Arch.	FPS	Chargement maximum sur le processeur ARM (%)
1	2,5	66
2	17,54	48
3	17,4	49
4	4,7	50
5	4,2	10

Tableau 4 : Ressources matérielles des cinq architectures

Arch.	LUT (%)	FF (%)	RAM (%)	DSP (%)
1	31	27	37	21
2	45	34	47	22
3	46	33	47	27
4	34	30	43	47
5	22	13	45	8

On peut voir que la configuration 1 ne satisfait pas la contrainte de vitesse de décodage puisqu'elle permet de décoder une vidéo à une vitesse de 2,5 fps. Cette configuration est donc abandonnée. On peut également éliminer les architectures 2, 3 et 4, car elles impliquent une trop grande occupation du processeur ARM cortex A9. La configuration retenue est donc la configuration 5.

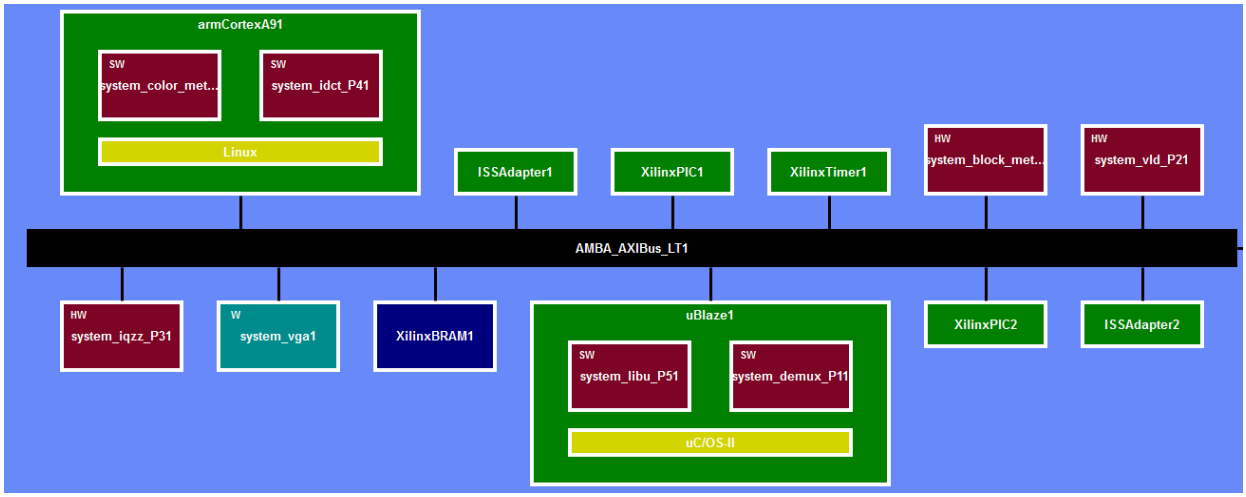

Figure 6-2 : Configuration finale

CHAPITRE 7 CONCLUSION ET PERSPECTIVES

Dans le cadre de ce mémoire, un flot de conception pour systèmes embarqués a été présenté à travers ses différentes étapes qui ont été mises en perspective par la présentation de plusieurs outils utilisés dans la conception de systèmes embarqués. Ce flot a ensuite été illustré grâce à un exemple d'application de décodage vidéo MJPEG.

Une revue de littérature a été élaborée et permet de mettre en avant divers outils inhérents à la conception de systèmes embarqués ainsi que quelques exemples de flot de conception qui les utilisent. On y retrouve différents langages de description utilisant les méthodologies MBE qui donnent rapidement, dans un flot de conception, une vision globale du projet permettant de faciliter les échanges clients-designers. Cet état de l'art présente également plusieurs langages ESL très proches d'une implémentation sur carte et qui, grâce à des principes d'encapsulations, permettent de simuler l'exécution du système et d'obtenir un comportement très proche de celui obtenu sur le système réel.

L'objectif principal qui consistait à combler le gap entre le langage de modélisation de haut niveau d'abstraction AADL et l'environnement d'exécution a été atteint. Il s'agissait de proposer une nouvelle méthodologie pour la conception de systèmes embarqués. La méthode proposée permet une vérification fonctionnelle d'un système et une validation de son architecture. En partant d'un modèle haut niveau décrit grâce au langage AADL, on utilise l'outil qui a été développé durant ce projet pour générer un script Python destiné à être exécuté sur la plateforme de conception conjointe logicielle/matérielle SpaceStudio. Une fois le projet porté sur la plateforme SpaceStudio, une étape de vérification fonctionnelle du système ainsi qu'un raffinement de son architecture peut alors débuter. Une fois tous les ajustements d'architecture effectués et les contraintes spécifiques aux systèmes embarqués fixées par l'utilisateur, le logiciel permet une génération de description RTL. Le flot de conception permet ainsi de débuter avec un langage très haut niveau pour arriver à une description de circuit tout en validant des aspects fonctionnels et non fonctionnels durant toute la conception du système.

L'outil développé permet une très bonne conversion pour un modèle AADL à condition que celui-ci soit un tant soit peu élaboré dans l'optique d'effectuer une validation architecturale. C'est-à-dire que le modèle doit se servir du *property set* explicité dans ce mémoire notamment lors de la déclaration des composants matériels du modèle qui doivent indiquer de quel type ils sont. Le travail qui a été effectué sur l'analyse des propriétés des différents composants de l'architecture d'un modèle permet désormais d'étoffer facilement le panel de propriétés déployable sur SpaceStudio.

Tous les branchements effectués entre les threads et les devices du modèle AADL sont également enregistrés du composant émetteur jusqu'au récepteur afin de mapper les communications inter modules sur la plateforme SpaceStudio.

De plus, l'outil permet de générer les codes sources pour les modules logiciels de SpaceStudio. Un squelette de code est fourni à l'utilisateur qui n'a plus qu'à le compléter une fois sur la plateforme SpaceStudio. Si du code C ou C++ est associé à un thread dans le modèle AADL alors l'outil permet de l'intégré au code source du module correspondant. Néanmoins, le code C doit répondre à plusieurs contraintes comme utiliser les fonctions de communication de SpaceStudio.

La génération d'un rapport des différentes données de simulations et d'implémentation se fait dans un premier temps par la vérification des contraintes temps réel des différents threads du système puis en récupérant les différentes métriques liées à l'implémentation du système sur carte. Ce rapport permet ensuite au concepteur de voir si le système répond aux contraintes données ou de voir comment le modifier pour y répondre.

La mise en application du flot de conception proposé a été réalisée par l'intermédiaire d'une application de décodage vidéo MJPEG. Plusieurs configurations du modèle AADL ont pu être portées vers la plateforme SpaceStudio qui a permis d'analyser plusieurs données résultantes des simulations. L'exemple du modèle AADL d'un système de régulation de vitesse récupéré sur Internet a permis de montrer les modifications à apporter au modèle pour rapidement l'adapter à l'outil développé dans le but de générer un projet sur la plateforme SpaceStudio à partir de celui-ci.

Les travaux effectués durant ce projet ont mené à l'écriture d'un article de conférence présenté lors du symposium Rapid System Prototyping à Amsterdam en octobre 2015. Un article de revue dans un journal est aussi en préparation.

À la suite de ce projet, plusieurs travaux futurs peuvent être envisagés afin d'en poursuivre les contributions. Les problèmes de compatibilités entre les fonctions génériques *read* et *write* désirées et la plateforme SpaceStudio causent des erreurs lors de la compilation du projet et représente un axe important d'amélioration à apporter. La qualité et la complétude du rapport transmis au designer AADL après la simulation et l'implémentation sont également à améliorer. Une autre piste pour des travaux futurs repose sur le fait d'effectuer une génération d'un modèle AADL à partir d'un projet SpaceStudio (i.e. opération inverse à celle réalisée par l'outil développé dans ce mémoire).

Un tel outil pourrait permettre, une fois l'exploration architecturale effectuée sous SpaceStudio, de revenir au niveau du modèle AADL pour effectuer toutes les validations de systèmes offerts par le langage. Ce fonctionnement correspond à la philosophie du langage AADL qui consiste à réaliser différentes analyses sur un modèle et d'avoir un retour afin d'étoffer celui-ci.

En conclusion, ce mémoire doit avoir contribué à apporter différentes pistes de réflexion sur différents outils de conception de systèmes embarqués et à apporter une solution au besoin d'un flot de conception rapide.

BIBLIOGRAPHIE

[1] D. C. Schmidt, "Guest editor's introduction: Model-driven engineering," *Computer*, vol. 39, pp. 0025-31, 2006.

[2] P. Wilson and H. A. Mantooth, *Model-based Engineering for Complex Electronic Systems: Techniques, Methods and Applications*: Newnes, 2013.

[3] B. Vogel-Heuser, D. Schütz, T. Frank, and C. Legat, "Model-driven engineering of Manufacturing Automation Software Projects – A SysML-based approach," *Mechatronics*, vol. 24, pp. 883-897, 10// 2014.

[4] M. Benyoussef, J.-F. Boland, G. Nicolescu, G. Bois, and J. Hugues, "Design Space Exploration: Bridging the Gap Between High-Level Models and Virtual Prototype," presented at the Embedded Real Time Software and Systems (ERTS2), 2014.

[5] P. H. Feiler and D. P. Gluch, *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*: Addison-Wesley, 2012.

[6] G. Martin, B. Bailey, and A. Piziali, *ESL design and verification: a prescription for electronic system level methodology*: Morgan Kaufmann, 2010.

[7] *https://wiki.sei.cmu.edu/aadl/index.php/Models_examples*.

[8] S. Bohner, R. Ravichandar, and J. Arthur, "Model-based engineering for change-tolerant systems," *Innovations in Systems and Software Engineering*, vol. 3, pp. 237-57, 12/ 2007.

[9] F. Herrera, H. Posadas, P. Penil, E. Villar, F. Ferrero, R. Valencia, *et al.*, "The COMPLEX methodology for UML/MARTE modeling and design space exploration of embedded systems," *Journal of Systems Architecture*, vol. 60, pp. 55-78, 01/ 2014.

[10] M. Sabetzadeh, S. Nejati, L. Briand, and A.-H. E. Mills, "Using SysML for modeling of safety-critical software-hardware interfaces: Guidelines and industry experience," in *13th IEEE International Symposium on High Assurance Systems Engineering, HASE 2011, November 10, 2011 - November 12, 2011*, Boca Raton, FL, United states, 2011, pp. 193-201.

[11] R. Varona-Gómez and E. Villar, "AADL simulation and performance analysis in SystemC," in *Engineering of Complex Computer Systems, 2009 14th IEEE International Conference on*, 2009, pp. 323-328.

[12] W. Geng, Z. Xing-she, D. Yun-wei, and Z. Hong-bing, "Studying on AADL-based architecture abstraction of embedded software," in *2009 International Conference on Scalable Computing and Communications; Eighth International Conference on Embedded Computing. SCALCOM-EMBEDDEDCOM 2009, 25-27 Sept. 2009*, Piscataway, NJ, USA, 2009, pp. 14-19.

[13] V. Gaudel, F. Singhoff, A. Plantec, S. Rubini, P. Dissaux, and J. Legrand, "An Ada design pattern recognition tool for AADL performance analysis," in *2011 Annual International Conference of the Association for Computing Machinery's Special Interest Group on the Ada Programming Language, SIGAda 2011, November 6, 2011 - November 10, 2011*, Denver, CO, United states, 2011, pp. 61-68.

[14] S. Rubini, F. Singhoff, and J. Hugues, "Modeling and verification of memory architectures with AADL and REAL," in *16th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS 2011, April 27, 2011 - April 29, 2011*, Las Vegas, NV, United states, 2011, pp. 338-343.

[15] O. Gilles and J. Hugues, "Expressing and Enforcing User-Defined Constraints of AADL Models," in *2010 15th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2010), 22-26 March 2010*, Los Alamitos, CA, USA, 2010, pp. 337-42.

[16] M. M. Fernandez, "Using AADL to enable MBSE for NASA space mission operations," in *13th International Conference on Space Operations, SpaceOps 2014, May 5, 2014 - May 9, 2014*, Pasadena, CA, United states, 2014.

[17] W. Cesário, A. Baghdadi, L. Gauthier, D. Lyonnard, G. Nicolescu, Y. Paviot, *et al.*, "Component-based design approach for multicore SoCs," in *Proceedings of the 39th annual Design Automation Conference*, 2002, pp. 789-794.

[18] A. C. H. Ng, J. W. Weijers, M. Glassee, T. Schuster, B. Bougard, and L. Van Der Perre, "ESL design and HW/SW co-verification of high-end software defined radio platforms," in *CODES+ISSS 2007: 5th International Conference on Hardware/Software Codesign and System Synthesis, September 30, 2007 - October 3, 2007*, Salzburg, Austria, 2007, pp. 191-196.

[19] M. Pockrandt, P. Herber, and S. Glesner, "Model checking a SystemC/TLM design of the AMBA AHB protocol," in *2011 9th IEEE Symposium on Embedded Systems for Real-Time Multimedia, 13-14 Oct. 2011*, Piscataway, NJ, USA, 2011, pp. 66-75.

[20] Y. W. Hau and M. Khalil-Hani, "SystemC-based HW/SW co-simulation platform for system-on-chip (SoC) design space exploration," *International Journal of Information and Communication Technology*, vol. 2, pp. 108-19, / 2009.

[21] R. Varona-Gómez, E. Villar, A. I. Rodríguez, F. Ferrero, and E. Alaña, "Architectural Optimization & Design of Embedded Systems based on AADL Performance Analysis," *American Journal of Computer Architecture*, vol. 1, pp. 21-36, 2012.

[22] L. Moss, H. Guerard, G. Dare, and G. Bois, "An ESL methodology for rapid creation of embedded aerospace systems using hardware-software co-design on virtual platforms," in *SAE 2012 Aerospace Electronics and Avionics Systems Conference, AEAS 2012, October 30, 2012 - November 1, 2012*, Phoenix, AZ, United states, 2012.

[23] V. Lefftz and J. Lachaize, "SoCKET: a HW/SW co-design flow: presentation feedbacks from space application domain," in *DASIA 2012. Data Systems In Aerospace, 14-16 May 2012*, Noordwijk, Netherlands, 2012, pp. 261-8.

[24] A. c. Wiki. (2010). *The Story of AADL*. Available: https://wiki.sei.cmu.edu/aadl/index.php/The_Story_of_AADL

[25] S. A. Team, "An extensible open source AADL tool environment (OSATE). Software Engineering Institute, CMU, 2006," ed.

[26] M. Kerboeuf, A. Plantec, F. Singhoff, A. Schach, and P. Dissaux, "Comparison of six ways to extend the scope of Cheddar to AADL v2 with Osate," in *Engineering of Complex Computer Systems (ICECCS), 2010 15th IEEE International Conference on*, 2010, pp. 367-372.

[27] L. Moss, H. Guérard, G. Dare, and G. Bois, "Rapid Design Exploration on an ESL Framework featuring Hardware-Software Codesign for ARM Processor-based FPGA's," *Space*, vol. 1, 2012.

[28] J. Hugues, B. Zalila, L. Pautet, and F. Kordon, "From the prototype to the final embedded system using the Ocarina AADL tool suite," *ACM Trans. Embed. Comput. Syst.*, vol. 7, pp. 1-25, 2008.

ANNEXES

ANNEXE A - PROPERTY SET

```
property set SpaceStudio is
with Memory_Properties;

----------------------TYPES OF PROPERTIES-------------------------
Size : type aadlinteger 0 Bytes .. Max_Memory_Size units Size_Units;
component_type : type enumeration ( Module, UserDevice, SSMemory, PIC,
SSBus, ISS,
Bridges, ExternalIOSlave, Serial, Signal,
Timer, Adapter
) ;
subtype : type enumeration (STANDARD, RTL, UserDeviceSlave, MASTERSLAVE,
GaislerDDR,
GaislerOnChipRAM, GaislerOnCHipROM, GaislerRAM, XilinxLMBRAM, XilinxBRAM,
APBBus, AMBA_AXIBus, AMBA_AXIBus_LT, AMBA_AHBBus, LMBBus, OPBBus, PLBBus,
TFChannel,
UTFChannel, leon3, uBlaze, armCortexA9, ISSAdapter
);
config_type : type enumeration (Simtek, Elix);

------------------POPERTIES RELATED TO PROJECT--------------------
name : aadlstring applies to (package, system, abstract);
ConfigName : aadlstring applies to (package, system, abstract);
level : SpaceStudio::config_type applies to (package, system, abstract);--
Need to be updated
SpaceStudioVersion : aadlinteger applies to (package, system, abstract);

-----------------PROPERTIES RELATED TO COMPONENTS-----------------
Type_component : SpaceStudio::component_type
applies to (all);
Subtype_component : SpaceStudio::subtype
applies to (all);
verbose : aadlboolean
applies to (all);
ulBaseAddress : aadlstring
applies to (all);
ulHighAddress : aadlstring
applies to (all);
ulMaxComponentSize : Size
applies to (all);
ulMinComponentSize : Size
applies to (all);
isFixRange : type aadlboolean;
ulSize : Size
applies to (memory);
```

```
end SpaceStudio;
```

ANNEXE B – TABLES PYTHON DES COMPOSANTS DU SYSTEME

1 Modules

Les informations nécessaires à lister pour un module incluent son nom, le nom du processeur sur lequel il doit être exécuté et la priorité de ce module. Ces informations sont listées de la façon suivante :

```
MODULE_NAMES = [('nom_du_module,'processeur_d_execution', priorité)]
```

Si aucun processeur n'est assigné au module pour son exécution dans le modèle AADL, ce champ sera renseigné par 'hardware'. La priorité d'un module est de base mise à 18 si celle-ci n'a pas été précisée dans la spécification AADL. Si le module doit s'exécuter en matériel, alors sa priorité ne sera pas prise en compte.

2 Devices

Pour un device, seuls son nom et celui du bus auquel il doit être branché pour communiquer avec le système sont à renseigner :

```
DEVICE_NAMES = [(nom_du_device','bus_de_branchement')]
```

Si aucun bus n'est assigné au device dans le modèle AADL, le backend s'interrompra et enverra un message d'erreur à l'utilisateur.

3 Processeurs

Quant au processeur, il a besoin de plus d'informations. Pour l'intégrer correctement au système, le script a besoin de connaitre son nom, le type de processeur dont il s'agit (voir Tableau 1), le bus de données auquel il est censé être branché et enfin son protocole d'ordonnancement :

```
PROCESSOR_NAMES = [('nom_du_processeur','type', 'bus','protocole_d_ordonancement')]
```

Encore une fois, le backend retournera un message d'erreur si le processeur n'a pas été branché sur un bus dans le modèle AADL. Le uBlaze est le type de processeur renseigné de base, et l'algorithme du tourniquet (de l'anglais *round-robin*) représente le protocole d'ordonnancement de base de l'OS.

4 Bus

Les bus sont les composants les plus basiques et il est seulement nécessaire de connaitre leur nom et leur type :

```
BUS_NAMES = [('nom_du_bus','type')]
```

Le type de base d'un bus à défaut d'être renseigné sera « AMBA_AXIBus ».

5 Mémoires

Tout comme les processeurs, le bus sur lequel la mémoire est branchée doit être indiqué ainsi que le nom de la mémoire et son type :

```
MEMORY_NAMES = [('nom_de_la_memoire','type','bus')]
```

Le type de base d'une mémoire est « XilinxBRAM ».

ANNEXE C - COMMUNICATION.H

Ce fichier comprend donc les déclarations des fonctions read et write génériques. Comme on peut le voir à la figure 8-1, ces fonctions font appel aux fonctions de communication de la plateforme SpaceStudio pour lesquels l'identifiant de la cible ou de la source de la communication a été récupéré grâce au fichier « deployement.h ».

```
template<typename Type> void Read(MODULES_DEVICES source, Type *data, int size=1, int
offset=0)
{
	if (isModule[source])
	{
		ModuleRead(source, SPACE_BLOCKING, data, size);
	}
	else
	{
		DeviceRead(source, offset, SPACE_BLOCKING, data, size);
	}
}

template<typename Type> void Write(MODULES_DEVICES dest, int offset=0, Type *data, int
size=1, int offset=0)
{
	if (isModule[dest])
	{
		ModuleWrite(dest, SPACE_BLOCKING, data, size);
	}
	else
	{
		DeviceWrite(dest, offset, SPACE_BLOCKING, data, size);
	}
}
```

Figure 7-1 : Déclaration des fonctions Read et Write

On utilise un template pour définir ces fonctions, car le type de donnée utilisé est défini par le modèle AADL puisqu'à chaque port de communication est associé un type de donnée.

ANNEXE D - DEPLOYMENT.H

Ce fichier se décompose en trois parties pour permettre aux fonctions de communication d'être entièrement génériques :

- La première partie consiste à déclarer tous les noms de modules et de devices du système comme un type énuméré de la façon suivante :

```
enum MODULES_DEVICES {
Nom_Module_1,
Nom_Module_2,
…
Nom_Device_1
};
```

- Ensuite grâce à des « #define », on associe à chaque nom de port du système une valeur du type énuméré défini précédemment correspondant à la cible ou à la source de la communication liée à ce port :

```
#define Nom_Module_1_Port1 Nom_Module_2
#define Nom_Module_1_Port1 Nom_Device_1
#define Nom_Module_2_Port2 Nom_Device_1
…
#define Nom_Device_1_Port1 Nom_Module_1
```

- Enfin, on créer un tableau de booléens que l'on appelle « isModule ». La taille de ce tableau est identique au nombre cumulé de module et de device. Les noms des modules et devices du système ayant été déclarés dans un type énuméré, ils peuvent servir d'indice pour accéder aux différentes cellules du tableau. Ainsi chaque cellule représente un des modules ou devices du système et est remplie avec un 1 si la case représente un module, et avec un 0 si elle représente un device. :

```
bool isModule[n] = {True, True, …, False};
```

ANNEXE E- MODELE AADL DE L'APPLICATION DE TRAITEMENT VIDEO

```
package mjpeg::software
public
with Data_Model;
with mjpeg::platform;


      data Uint_data
--    properties
--          Data_Model::Representation => "unsigned int";
      end Uint_data;

      data SAshort_data
--    properties
--          Data_Model::Representation => "SPACE_ALIGNED short";
      end SAshort_data;

      data SAUchar_data
--    properties
--          Data_Model::Representation => "SPACE_ALIGNED unsigned char";
      end SAUchar_data;

      data VLD_data
--    properties
--          Data_Model::Representation => "vld_metrics";
      end VLD_data;

      data Uint64_data
--    properties
--          Data_Model::Representation => "Uint64_t";
      end Uint64_data;

      data int_data
--    properties
--          Data_Model::Representation => "int";
      end int_data;


      subprogram Demux_Function
      properties

            Source_Text => ("DEMUX.cpp");
            Source_Name => "demux_funct";
      end Demux_Function;

      thread Demux
      features
            to_vld : out event data port Uint_data;
            to_iqzz : out event data port Uint_data;
            to_idct : out event data port Uint_data;
            to_libu : out event data port Uint_data;
      end Demux;
```

```
thread implementation Demux.impl
calls
      loop : {code : subprogram Demux_Function;};
properties
      Period => 50ms;
      Dispatch_Protocol => Periodic;
      Compute_Entrypoint => classifier (Demux_Function);
end Demux.impl;

process Demuxp
features
      to_vld : out event data port Uint_data;
      to_iqzz : out event data port Uint_data;
      to_idct : out event data port Uint_data;
      to_libu : out event data port Uint_data;
end Demuxp;

process implementation Demuxp.impl
subcomponents
      P1 : thread Demux.impl;
connections
      c_vld : port P1.to_vld -> to_vld;
      c_iqzz : port P1.to_iqzz -> to_iqzz;
      c_idct : port P1.to_idct -> to_idct;
      c_libu : port P1.to_libu -> to_libu;
end Demuxp.impl;

subprogram VLD_Function
features
      uiCommand : in parameter Uint_data;
properties

      Source_Text => ("VLD.cpp");
      Source_Name => "vld_funct";
end VLD_Function;

thread VLD
features
      from_demux : in event data port Uint_data;
      to_iqzz : out event data port SAshort_data;
      to_block_metrics : out event data port VLD_Data;
end VLD;

thread implementation VLD.impl
calls loop : {code : subprogram VLD_Function;};
connections
      param1 : parameter from_demux -> code.uiCommand;
properties
      Period => 50ms;
      Dispatch_Protocol => Periodic;
      Compute_Entrypoint => classifier (VLD_Function);
end VLD.impl;

process VLDp
features
      from_demux : in event data port Uint_data;
      to_iqzz : out event data port SAshort_data;
```

```
      to_block_metrics : out event data port VLD_Data;
end VLDp;

process implementation VLDp.impl
subcomponents
      P2 : thread VLD.impl;
connections
      c_demux : port from_demux -> P2.from_demux;
      c_iqzz : port P2.to_iqzz -> to_iqzz;
      c_block_metrics : port P2.to_block_metrics -> to_block_metrics;
end VLDp.impl;

subprogram IQZZ_Function
features
      uiCommand : in parameter Uint_data;
      nbQuant : in parameter Uint_data;
properties

      Source_Text => ("IQZZ.cpp");
      Source_Name => "iqzz_funct";
end IQZZ_Function;

subprogram IQZZ_Before
features
      nbQuant2 : in parameter Uint_data;
      nbQuant : out parameter Uint_data;
properties

      Source_Text => ("IQZZ.cpp");
      Source_Name => "before_iqzz";
end IQZZ_Before;

thread IQZZ
features
      from_demux : in event data port Uint_data;
      from_vld : in event data port SAshort_data;
      to_idct : out event data port SAshort_data;
end IQZZ;

thread implementation IQZZ.impl
calls
      loop : {code : subprogram IQZZ_Function;};
      before : {code1 : subprogram IQZZ_Before;};
connections
      param1 : parameter from_demux -> code1.nbQuant2;
      param2 : parameter from_demux -> code.uiCommand;
      param3 : parameter code1.nbQuant -> code.nbQuant;
properties
      Period => 50ms;
      Dispatch_Protocol => Periodic;
      Compute_Entrypoint => classifier (IQZZ_Function);
end IQZZ.impl;

process IQZZp
features
      from_demux : in event data port Uint_data;
      from_vld : in event data port SAshort_data;
```

```
      to_idct : out event data port SAshort_data;
end IQZZp;

process implementation IQZZp.impl
subcomponents
      P3 : thread IQZZ.impl;
connections
      c_demux : port from_demux -> P3.from_demux;
      c_vld : port from_vld -> P3.from_vld;
      c_idct : port P3.to_idct -> to_idct;
end IQZZp.impl;

subprogram IDCT_Function
features
      uiCommand : in parameter Uint_data;
properties

      Source_Text => ("IDCT.cpp");
      Source_Name => "idct_funct";
end IDCT_Function;

thread IDCT
features
      from_demux : in event data port Uint_data;
      from_iqzz : in event data port SAshort_data;
      to_libu : out event data port SAUchar_data;
end IDCT;

thread implementation IDCT.impl
calls loop : {code : subprogram IDCT_Function;};
connections
      param1 : parameter from_demux -> code.uiCommand;
properties
      Period => 50ms;
      Dispatch_Protocol => Periodic;
      Compute_Entrypoint => classifier (IDCT_Function);
end IDCT.impl;

process IDCTp
features
      from_demux : in event data port Uint_data;
      from_iqzz : in event data port SAshort_data;
      to_libu : out event data port SAUchar_data;
end IDCTp;

process implementation IDCTp.impl
subcomponents
      P4 : thread IDCT.impl;
connections
      c_demux : port from_demux -> P4.from_demux;
      c_iqzz : port from_iqzz -> P4.from_iqzz;
      c_libu : port P4.to_libu -> to_libu;
end IDCTp.impl;

subprogram LIBU_Function
features
      uiCommand : in parameter Uint_data;
```

```
properties

      Source_Text => ("LIBU.cpp");
      Source_Name => "libu_funct";
end LIBU_Function;

thread LIBU
features
      from_demux : in event data port Uint_data;
      from_idct : in event data port SAUchar_data;
      to_vga : out event data port Uint_data;
      to_block_metrics : out event data port Uint64_data;
      to_color_metrics : out event data port int_data;
end LIBU;

thread implementation LIBU.impl
calls loop : {code : subprogram LIBU_Function;};
connections
      param1 : parameter from_demux -> code.uiCommand;
properties
      Period => 50ms;
      Compute_Entrypoint => classifier (LIBU_Function);
      Dispatch_Protocol => Periodic;
end LIBU.impl;

process LIBUp
features
      from_demux : in event data port Uint_data;
      from_idct : in event data port SAUchar_data;
      to_vga : out event data port Uint_data;
      to_block_metrics : out event data port Uint64_data;
      to_color_metrics : out event data port int_data;
end LIBUp;

process implementation LIBUp.impl
subcomponents
      P5 : thread LIBU.impl;
connections
      c_demux : port from_demux -> P5.from_demux;
      c_idct : port from_idct -> P5.from_idct;
      c_vga : port P5.to_vga -> to_vga;
      c_block_metrics : port P5.to_block_metrics -> to_block_metrics;
      c_color_metrics : port P5.to_color_metrics -> to_color_metrics;
end LIBUp.impl;

subprogram BLOCK_METRICS_Function
properties

      Source_Text => ("BLOCK_METRICS.cpp");
      Source_Name => "metrics";
end BLOCK_METRICS_Function;

thread BLOCK_METRICS
features
      from_vld : in event data port VLD_data;
      from_libu : in event data port Uint64_data;
end BLOCK_METRICS;
```

```
thread implementation BLOCK_METRICS.impl
calls loop : {code : subprogram BLOCK_METRICS_Function;};
properties
      Period => 50ms;
      Compute_Entrypoint => classifier (BLOCK_METRICS_Function);
      Dispatch_Protocol => Periodic;
end BLOCK_METRICS.impl;

process BLOCK_METRICSp
features
      from_vld : in event data port VLD_data;
      from_libu : in event data port Uint64_data;
end BLOCK_METRICSp;

process implementation BLOCK_METRICSp.impl
subcomponents
      P6 : thread BLOCK_METRICS.impl;
connections
      c_vld : port from_vld -> P6.from_vld;
      c_libu : port from_libu -> P6.from_libu;
end BLOCK_METRICSp.impl;

subprogram COLOR_METRICS_Function
properties

      Source_Text => ("COLOR_METRICS.cpp");
      Source_Name => "color";
end COLOR_METRICS_Function;

thread COLOR_METRICS
features
      from_libu : in event data port int_data;
end COLOR_METRICS;

thread implementation COLOR_METRICS.impl
calls loop : {code : subprogram COLOR_METRICS_Function;};
properties
      Period => 50ms;
      Compute_Entrypoint => classifier (COLOR_METRICS_Function);
      Dispatch_Protocol => Periodic;
end COLOR_METRICS.impl;

process COLOR_METRICSp
features
      from_libu : in event data port int_data;
end COLOR_METRICSp;

process implementation COLOR_METRICSp.impl
subcomponents
      P7 : thread COLOR_METRICS.impl;
connections
      c_libu : port from_libu -> P7.from_libu;
end COLOR_METRICSp.impl;


device VGA
```

```
	features
		from_libu : in event data port Uint_data;

		b0 : requires bus access mjpeg::platform::AMBA;
	end VGA;


	system software
	end software;

	system implementation software.impl
	subcomponents
		demux : process Demuxp.impl;
		vld : process VLDp.impl;
		iqzz : process IQZZp.impl;
		idct : process IDCTp.impl;
		libu : process LIBUp.impl;
		block_metrics : process BLOCK_METRICSp.impl;
		color_metrics : process COLOR_METRICSp.impl;

		vga : device VGA;
	connections
		C_demux_vld : port demux.to_vld -> vld.from_demux;
		C_demux_idct : port demux.to_idct -> idct.from_demux;
		C_demux_iqzz : port demux.to_iqzz -> iqzz.from_demux;
		c_demux_libu : port demux.to_libu -> libu.from_demux;
		c_vld_iqzz : port vld.to_iqzz -> iqzz.from_vld;
		c_vld_block_metrics : port vld.to_block_metrics ->
block_metrics.from_vld;
		c_iqzz_idct : port iqzz.to_idct -> idct.from_iqzz;
		c_idct_libu : port idct.to_libu -> libu.from_idct;
		c_libu_vga : port libu.to_vga -> vga.from_libu;
		c_libu_block_metrics : port libu.to_block_metrics ->
block_metrics.from_libu;
		c_libu_color_metrics : port libu.to_color_metrics ->
color_metrics.from_libu;
	end software.impl;

end mjpeg::software;




package mjpeg::platform
public
with SpaceStudio;


	processor a9
	features
		socket : requires bus access AMBA;
	end a9;

	processor implementation a9.impl
	properties
		SpaceStudio::Subtype_component => armCortexA9;
```

```
	end a9.impl;

	processor ub
	features
		socket : requires bus access AMBA;
	end ub;

	processor implementation ub.impl
	properties
		SpaceStudio::Subtype_component => uBlaze;
	end ub.impl;

	bus AMBA
	properties
		SpaceStudio::Subtype_component => AMBA_AXIBus;
	end AMBA;


	memory BRAM
	features
		b0 : requires bus access AMBA;
	properties
	SpaceStudio::Subtype_component => XilinxBRAM;
	end BRAM;

	system platform
	end platform;

	system implementation platform.ubimpl
	subcomponents
		proc : processor ub.impl;
		busAmba : bus AMBA;
		mem : memory BRAM;
	connections
		b0 : bus access busAmba <-> proc.socket;
		b1 : bus access busAmba <-> mem.b0;
	end platform.ubimpl;

	system implementation platform.a9impl
	subcomponents
		proc : processor a9.impl;
		busAmba : bus AMBA;
		mem : memory BRAM;
	connections
		b0 : bus access busAmba <-> proc.socket;
		b1 : bus access busAmba <-> mem.b0;
	end platform.a9impl;

end mjpeg::platform;



package mjpeg::integration
public
with mjpeg::software;
with mjpeg::platform;
```

```
	system integration
	end integration;

	system implementation integration.ubimpl
	subcomponents
		demux : process mjpeg::software::Demuxp.impl;
		vld : process mjpeg::software::VLDp.impl;
		iqzz : process mjpeg::software::IQZZp.impl;
		idct : process mjpeg::software::IDCTp.impl;
		libu : process mjpeg::software::LIBUp.impl;
		block_metrics : process mjpeg::software::BLOCK_METRICSp.impl;
		color_metrics : process mjpeg::software::COLOR_METRICSp.impl;
		vga : device mjpeg::software::VGA;
		proc : processor mjpeg::platform::ub.impl;
		busAmba : bus mjpeg::platform::AMBA;
		mem : memory mjpeg::platform::BRAM;
	connections
		C_demux_vld : port demux.to_vld -> vld.from_demux;
		C_demux_idct : port demux.to_idct -> idct.from_demux;
		C_demux_iqzz : port demux.to_iqzz -> iqzz.from_demux;
		c_demux_libu : port demux.to_libu -> libu.from_demux;
		c_vld_iqzz : port vld.to_iqzz -> iqzz.from_vld;
		c_vld_block_metrics : port vld.to_block_metrics ->
block_metrics.from_vld;
		c_iqzz_idct : port iqzz.to_idct -> idct.from_iqzz;
		c_idct_libu : port idct.to_libu -> libu.from_idct;
		c_libu_vga : port libu.to_vga -> vga.from_libu;
		c_libu_block_metrics : port libu.to_block_metrics ->
block_metrics.from_libu;
		c_libu_color_metrics : port libu.to_color_metrics ->
color_metrics.from_libu;
		b0 : bus access busAmba <-> proc.socket;
		b1 : bus access busAmba <-> mem.b0;
		b2 : bus access busAmba <-> vga.b0;
	properties
		Actual_Processor_Binding => (reference (proc)) applies to demux;
		Actual_Processor_Binding => (reference (proc)) applies to idct;
		Actual_Processor_Binding => (reference (proc)) applies to iqzz;
		Actual_Processor_Binding => (reference (proc)) applies to libu;
		Actual_Memory_Binding => (reference (mem)) applies to demux;
	end integration.ubimpl;

	system implementation integration.a9impl
	subcomponents
		demux : process mjpeg::software::Demuxp.impl;
		vld : process mjpeg::software::VLDp.impl;
		iqzz : process mjpeg::software::IQZZp.impl;
		idct : process mjpeg::software::IDCTp.impl;
		libu : process mjpeg::software::LIBUp.impl;
		block_metrics : process mjpeg::software::BLOCK_METRICSp.impl;
		color_metrics : process mjpeg::software::COLOR_METRICSp.impl;
		vga : device mjpeg::software::VGA;
		proc : processor mjpeg::platform::a9.impl;
		busAmba : bus mjpeg::platform::AMBA;
		mem : memory mjpeg::platform::BRAM;
	connections
```

```
            C_demux_vld : port demux.to_vld -> vld.from_demux;
            C_demux_idct : port demux.to_idct -> idct.from_demux;
            C_demux_iqzz : port demux.to_iqzz -> iqzz.from_demux;
            c_demux_libu : port demux.to_libu -> libu.from_demux;
            c_vld_iqzz : port vld.to_iqzz -> iqzz.from_vld;
            c_vld_block_metrics : port vld.to_block_metrics ->
block_metrics.from_vld;
            c_iqzz_idct : port iqzz.to_idct -> idct.from_iqzz;
            c_idct_libu : port idct.to_libu -> libu.from_idct;
            c_libu_vga : port libu.to_vga -> vga.from_libu;
            c_libu_block_metrics : port libu.to_block_metrics ->
block_metrics.from_libu;
            c_libu_color_metrics : port libu.to_color_metrics ->
color_metrics.from_libu;
            b0 : bus access busAmba <-> proc.socket;
            b1 : bus access busAmba <-> mem.b0;
            b2 : bus access busAmba <-> vga.b0;
      properties
            Actual_Processor_Binding => (reference (proc)) applies to demux;
            Actual_Processor_Binding => (reference (proc)) applies to iqzz;
            Actual_Processor_Binding => (reference (proc)) applies to libu;
            Actual_Memory_Binding => (reference (mem)) applies to demux;
      end integration.a9impl;

end mjpeg::integration;
```

ANNEXE F - SCRIPT PYTHON GENERE DANS L'EXEMPLE

```
import os
import shutil
import tempfile

def create_project(project_name, base_dir):
                                return projectEngine.createProject(project_name, base_dir)

def populate_project(project, module_names, device_names, base_dir):
                                path = base_dir + '\import\src'
                                os.mkdir(path)

shutil.copyfile('C:\workspace\SpaceStudioProject\deployement.h', path + '\deployement.h')

shutil.copyfile('C:\workspace\SpaceStudioProject\communication.h', path + '\communication.h')
                                for module_name in module_names:
                                    name = module_name[0]
                                    module = project.createModule(name)
                                    resource_dir =
os.path.join(os.environ['SPACE_CODESIGN_ENV'],  'C:\workspace\SpaceStudioProject')
                                    shutil.copyfile(os.path.join(resource_dir, name + '.h'),
module.getHeaderPath())
                                    shutil.copyfile(os.path.join(resource_dir, name + '.cpp'),
module.getSourcePath())
                                for device_name in device_names:
                                    name = device_name[0]
                                    device = project.createDevice(name, False)
                                    #resource_dir =
os.path.join(os.environ['SPACE_CODESIGN_DEV'],
'workspace/PrototypeExploration/src/main/resources/com/spacecodesign/prototype/exploration')
                                    #shutil.copyfile(os.path.join(resource_dir, name + '.h'),
device.getHeaderPath())
                                    #shutil.copyfile(os.path.join(resource_dir, name + '.cpp'),
device.getSourcePath())

def create_designs(project, name, module_names, device_names, processor_names, bus_names,
memory_names):
                                Design_name = name + '.design'
                                bus = {}
                                processor = {}
                                design = project.createArchitecturalDesign(Design_name)
                                for bus_name in bus_names :
                                    bus[bus_name[0]] =
design.createComponentInstance('Bus',bus_name[1])
                                Timer = design.createComponentInstance('Timer', 'XilinxTimer')
                                Timer.connectTo(bus[bus_names[0][0]])
                                for memory_name in memory_names :
                                    memory = design.createComponentInstance('Memory',
memory_name[1])
                                    memory.connectTo(bus[memory_name[2]])
                                for processor_name in processor_names :
                                    processor[processor_name[0]] =
design.createProcessorInstance(bus[processor_name[2]], processor_name[1])
                                for module_name in module_names :
                                    module = design.createModuleInstance(module_name[0])
                                    if not module_name[1] == 'Hardware' :
```

```
                                    module.mapTo(processor[module_name[1]])
                                else :
                                    module.mapTo(bus[bus_names[0][0]])
                            for device_name in device_names:
                                device = design.createDeviceInstance(device_name[0])
                                device.connectTo(bus[device_name[1]])
                            design.save()

PROJECT_NAME = 'integration.ubimpl'
BUS_NAMES = [('busAmba','AMBA_AXIBus')]
MEMORY_NAMES = [('BRAM','XilinxBRAM','busAmba')]
# processor (processor_name, processor_type, binding_bus, scheduling_protocol)
PROCESSOR_NAMES = [('proc','uBlaze','busAmba','round_robin')]
# module (module_name, binding_processor, priority, dispatch_protocol, period)
MODULE_NAMES = [('system_demux_P1','proc',18,'Periodic',50),
('system_vld_P2','Hardware',18,'Periodic',50), ('system_iqzz_P3','proc',18,'Periodic',50),
('system_idct_P4','proc',18,'Periodic',50), ('system_libu_P5','Hardware',18,'Periodic',50),
('system_block_metrics_P6','Hardware',18,'Periodic',50),
('system_color_metrics_P7','Hardware',18,'Periodic',50)]
DEVICE_NAMES = [('system_vga','busAmba')]
temp_dir = tempfile.mkdtemp(dir='c:/temp')
try:
                            project = create_project(PROJECT_NAME, temp_dir)
                            populate_project(project, MODULE_NAMES, DEVICE_NAMES, temp_dir
+ '\\' + PROJECT_NAME)
                            create_designs(project, PROJECT_NAME, MODULE_NAMES,
DEVICE_NAMES, PROCESSOR_NAMES, BUS_NAMES, MEMORY_NAMES)
finally:
                            print 'fin'
```

ANNEXE G - FICHIER DEPLOYEMENT.H GÉNÉRÉ DANS L'EXEMPLE

```
#ifndef DEPLOYEMENT_H_
#define DEPLOYEMENT_H_

#include "PlatformDefinitions.h"


enum MODULES_DEVICES {
system_demux_P1,
system_vld_P2,
system_iqzz_P3,
system_idct_P4,
system_libu_P5,
system_block_metrics_P6,
system_color_metrics_P7,
system_vga
};

#define system_demux_P1_to_vld system_vld_P2
#define system_demux_P1_to_iqzz system_iqzz_P3
#define system_demux_P1_to_idct system_idct_P4
#define system_demux_P1_to_libu system_libu_P5
#define system_vld_P2_from_demux system_demux_P1
#define system_vld_P2_to_iqzz system_iqzz_P3
#define system_vld_P2_to_block_metrics system_block_metrics_P6
#define system_iqzz_P3_from_demux system_demux_P1
#define system_iqzz_P3_from_vld system_vld_P2
#define system_iqzz_P3_to_idct system_idct_P4
#define system_idct_P4_from_demux system_demux_P1
#define system_idct_P4_from_iqzz system_iqzz_P3
#define system_idct_P4_to_libu system_libu_P5
#define system_libu_P5_from_demux system_demux_P1
#define system_libu_P5_from_idct system_idct_P4
#define system_libu_P5_to_vga system_vga
#define system_libu_P5_to_block_metrics system_block_metrics_P6
#define system_libu_P5_to_color_metrics system_color_metrics_P7
#define system_block_metrics_P6_from_vld system_vld_P2
#define system_block_metrics_P6_from_libu system_libu_P5
#define system_color_metrics_P7_from_libu system_libu_P5
#define system_vga_from_libu system_libu_P5

static const bool isModule[8] = {1, 1, 1, 1, 1, 1, 1, 0};



#end
```

ANNEXE H - MODÈLE AADL DU SYSTÈME DE RÉGULATION DE VITESSE

```
--
--  This model is part of the Speed Regulation model
--  that is a case-study that uses AADL for making
--  analysis of different architecture variations.
--
--  A full description of this case study can be found
--  on https://wiki.sei.cmu.edu/aadl/index.php/SpeedRegulation
--
--  Copyright Carnegie Mellon Software Engineering Institute, 2014.
--

package speed_regulation::integration

public

with speed_regulation::devices;
with speed_regulation::software;
with speed_regulation::platform;

system integration
end integration;

system implementation integration.generic
subcomponents
-- input devices
      obstacle_camera : device speed_regulation::devices::camera;
      obstacle_radar : device speed_regulation::devices::radar;
      wheel_sensor : device speed_regulation::devices::speed_wheel_sensor;
      laser_sensor : device speed_regulation::devices::speed_laser_sensor;
      gps1 : device speed_regulation::devices::gps.impl1;
      gps2 : device speed_regulation::devices::gps.impl2;

-- software
    image_acquisition : process speed_regulation::software::image_acquisition.i;
    radar_acquisition : process speed_regulation::software::radar_acquisition.i;

    speed_estimate    : process speed_regulation::software::speed_estimate.i;

    position_voter    : process speed_regulation::software::position_voter.i;

    obstacle_detection : process speed_regulation::software::obstacle_detection.i;

    obstacle_distance_evaluation : process
speed_regulation::software::obstacle_distance_evaluation.i;

    speed_threshold_calculation : process
speed_regulation::software::speed_threshold_computation.i;

    speed_controller : process speed_regulation::software::speed_controller.i;

    emergency_detection : process
speed_regulation::software::emergency_detection.i;

    warning_activation : process speed_regulation::software::warning_activation.i;
```

```
-- output devices
      warning_alert : device speed_regulation::devices::warning_device;
      brake         : device speed_regulation::devices::brake;
      acceleration  : device speed_regulation::devices::acceleration;
connections
-- obstacle camera to image acquisition
   c000 : port obstacle_camera.picture -> image_acquisition.picture;

-- obstacle radar to radar acquisition
   c010 : port obstacle_radar.distance_estimate ->
radar_acquisition.obstacle_distance;

-- speed wheel sensor to speed estimate
   c020 : port wheel_sensor.speed -> speed_estimate.wheel_sensor;

-- laser sensor to speed estimate
   c030 : port laser_sensor.speed -> speed_estimate.laser_sensor;

-- GPS1 to GPS voter
      c040 : port gps1.position -> position_voter.position1;

-- GPS2 to GPS voter
      c050 : port gps2.position -> position_voter.position2;

-- Image acquisition to obstacle detection
   c060 : port image_acquisition.obstacle_detected -> obstacle_detection.camera;

-- Radar acquisition to obstacle detection
   c070 : port radar_acquisition.obstacle_detected -> obstacle_detection.radar;

-- obstacle detection to obstacle distance evaluation
   c080 : port obstacle_detection.obstacle_detected ->
obstacle_distance_evaluation.obstacle_detected;

-- obstacle distance evaluation to emergency detection
   c090 : port obstacle_distance_evaluation.obstacle_distance ->
emergency_detection.obstacle_distance;

-- emergency detection to warning activation
   c100 : port emergency_detection.emergency_detected ->
warning_activation.emergency_detected;

-- warning activation to warning device
   c110 : port warning_activation.activate_warning -> warning_alert.warning;

-- speed estimate to speed threshold computation

   c120 : port speed_estimate.speed -> speed_threshold_calculation.current_speed;

-- speed estimate to speed controller
      c130 : port speed_estimate.speed -> speed_controller.current_speed;

-- speed estimate to obstacle distance evaluation
      c140 : port speed_estimate.speed ->
obstacle_distance_evaluation.current_speed;
```

```
-- speed estimate to emergency detection
      c150 : port speed_estimate.speed -> emergency_detection.current_speed;

-- emergency detection to speed controller
      c160 : port emergency_detection.emergency_detected ->
speed_controller.emergency_detected;

-- speed threshold computation to speed controller
      c170 : port speed_threshold_calculation.threshold ->
speed_controller.threshold;

-- position voter to speed controller
      c180 : port position_voter.position -> speed_controller.position;

-- position voter to speed threshold computation
      c190 : port position_voter.position -> speed_threshold_calculation.position;

-- speed controller to brake
      c200 : port speed_controller.cmd -> brake.cmd;

-- speed controller to acceleration
      c210 : port speed_controller.cmd -> acceleration.cmd;
flows
    --  From the camera to the warning device
      f0 : end to end flow    obstacle_camera.f0 -> c000 ->
                                        image_acquisition.f0 -> c060 ->
                                        obstacle_detection.f0 -> c080 ->
                                        obstacle_distance_evaluation.f1 -> c090 -
>
                                        emergency_detection.f1 -> c100 ->
                                        warning_activation.f0 -> c110 ->
                                        warning_alert.f0 {Latency => 700ms .. 900
ms;};
    --  From the camera to the brake
      f1 : end to end flow    obstacle_camera.f0 -> c000 ->
                                        image_acquisition.f0 -> c060 ->
                                        obstacle_detection.f0 -> c080 ->
                                        obstacle_distance_evaluation.f1 -> c090 -
>
                                        emergency_detection.f1 -> c160 ->
                                        speed_controller.f3 -> c200 ->
                                        brake.f0;
    --  From the camera to the acceleration
      f2 : end to end flow    obstacle_camera.f0 -> c000 ->
                                        image_acquisition.f0 -> c060 ->
                                        obstacle_detection.f0 -> c080 ->
                                        obstacle_distance_evaluation.f1 -> c090 -
>
                                        emergency_detection.f1 -> c160 ->
                                        speed_controller.f3 -> c210 ->
                                        acceleration.f0;
      -- From the wheel speed sensor to brake
      f3 : end to end flow wheel_sensor.f0 -> c020 ->
                            speed_estimate.f0 -> c120 ->
                            speed_threshold_calculation.f1 -> c170 ->
                            speed_controller.f2 -> c200 ->
                            brake.f0;
```

```
f4 : end to end flow wheel_sensor.f0 -> c020 ->
                     speed_estimate.f0 -> c130 ->
                     speed_controller.f1 -> c200 ->
                     brake.f0;

-- From the wheel speed sensor to acceleration
f5 : end to end flow wheel_sensor.f0 -> c020 ->
                     speed_estimate.f0 -> c120 ->
                     speed_threshold_calculation.f1 -> c170 ->
                     speed_controller.f2 -> c210 ->
                     acceleration.f0;
f6 : end to end flow wheel_sensor.f0 -> c020 ->
                     speed_estimate.f0 -> c130 ->
                     speed_controller.f1 -> c210 ->
                     acceleration.f0;

-- From the laser speed sensor to brake
f7 : end to end flow laser_sensor.f0 -> c030 ->
                     speed_estimate.f1 -> c120 ->
                     speed_threshold_calculation.f1 -> c170 ->
                     speed_controller.f2 -> c200 ->
                     brake.f0;

f8 : end to end flow laser_sensor.f0 -> c030 ->
                     speed_estimate.f1 -> c130 ->
                     speed_controller.f1 -> c200 ->
                     brake.f0;

-- From the laser speed sensor to acceleration
f9 : end to end flow laser_sensor.f0 -> c030 ->
                     speed_estimate.f1 -> c120 ->
                     speed_threshold_calculation.f1 -> c170 ->
                     speed_controller.f2 -> c210 ->
                     acceleration.f0;

f10 : end to end flow laser_sensor.f0 -> c030 ->
                     speed_estimate.f1 -> c130 ->
                     speed_controller.f1 -> c210 ->
                     acceleration.f0;

-- From gps1 to brake
f11 : end to end flow gps1.f0 -> c040 ->
                      position_voter.f0 -> c190 ->
                      speed_threshold_calculation.f0 -> c170 ->
                      speed_controller.f2 -> c200 ->
                      brake.f0;
f12 : end to end flow gps1.f0 -> c040 ->
                      position_voter.f0 -> c180 ->
                      speed_controller.f0 -> c200 ->
                      brake.f0;
-- From gps1 to acceleration
f13 : end to end flow gps1.f0 -> c040 ->
                      position_voter.f0 -> c190 ->
                      speed_threshold_calculation.f0 -> c170 ->
                      speed_controller.f2 -> c210 ->
                      acceleration.f0;
f14 : end to end flow gps1.f0 -> c040 ->
```

```
                        position_voter.f0 -> c180 ->
                        speed_controller.f0 -> c210 ->
                        acceleration.f0;

    -- From gps2 to brake
    f15 : end to end flow gps2.f0 -> c050 ->
                        position_voter.f1 -> c190 ->
                        speed_threshold_calculation.f0 -> c170 ->
                        speed_controller.f2 -> c200 ->
                        brake.f0;
    f16 : end to end flow gps2.f0 -> c050 ->
                        position_voter.f1 -> c180 ->
                        speed_controller.f0 -> c200 ->
                        brake.f0;

    -- From gps2 to acceleration
    f17 : end to end flow gps2.f0 -> c050 ->
                        position_voter.f1 -> c190 ->
                        speed_threshold_calculation.f0 -> c170 ->
                        speed_controller.f2 -> c210 ->
                        acceleration.f0;
    f18 : end to end flow gps2.f0 -> c050 ->
                        position_voter.f1 -> c180 ->
                        speed_controller.f0 -> c210 ->
                        acceleration.f0;


    -- From wheel speed sensor to warning device
    f19 : end to end flow wheel_sensor.f0 -> c020 ->
                       speed_estimate.f0 -> c140 ->
                                obstacle_distance_evaluation.f0 -> c090 ->
                                emergency_detection.f1 -> c100 ->
                                warning_activation.f0 -> c110 ->
                                warning_alert.f0;

    f20 : end to end flow wheel_sensor.f0 -> c020 ->
                       speed_estimate.f0 -> c150 ->
                                emergency_detection.f0 -> c100 ->
                                warning_activation.f0 -> c110 ->
                                warning_alert.f0;

    -- From laser speed sensor to warning device
    f21 : end to end flow laser_sensor.f0 -> c030 ->
                       speed_estimate.f1 -> c140 ->
                                obstacle_distance_evaluation.f0 -> c090 ->
                                emergency_detection.f1 -> c100 ->
                                warning_activation.f0 -> c110 ->
                                warning_alert.f0;

    f22 : end to end flow laser_sensor.f0 -> c030 ->
                       speed_estimate.f1 -> c150 ->
                                emergency_detection.f0 -> c100 ->
                                warning_activation.f0 -> c110 ->
                                warning_alert.f0;

--properties
```

```
--    Timing => immediate applies to      c000, c010, c020, c030, c040, c050, c060,
c070, c080, c090,
--                                                c100, c110, c120, c130, c140,
c150, c160, c170, c180, c190,
--                                                c200, c210;

end integration.generic;


system implementation integration.implementation1 extends integration.generic
subcomponents
      ecu1 : processor speed_regulation::platform::ecu_can_one_connector;
      ecu2 : processor speed_regulation::platform::ecu_can_one_connector;
      can  : bus speed_regulation::platform::can;
connections
      b0 : bus access can <-> ecu1.socket;
      b1 : bus access can <-> ecu2.socket;
      b2 : bus access can <-> obstacle_camera.socket1;
      b3 : bus access can <-> obstacle_radar.socket1;
      b4 : bus access can <-> wheel_sensor.socket1;
      b5 : bus access can <-> laser_sensor.socket1;
      b6 : bus access can <-> gps1.socket1;
      b7 : bus access can <-> gps2.socket1;
      b8 : bus access can <-> warning_alert.socket1;
      b9 : bus access can <-> brake.socket1;
      b10 : bus access can <-> acceleration.socket1;
properties
      Actual_Connection_Binding => (reference (can)) applies to c080;
      Actual_Connection_Binding => (reference (can)) applies to c120;
      Actual_Connection_Binding => (reference (can)) applies to c130;
      Actual_Connection_Binding => (reference (can)) applies to c140;
      Actual_Connection_Binding => (reference (can)) applies to c150;
      Actual_Connection_Binding => (reference (can)) applies to c180;
      Actual_Connection_Binding => (reference (can)) applies to c190;
      Allowed_Processor_Binding => (reference (ecu1)) applies to image_acquisition;
      Allowed_Processor_Binding => (reference (ecu1)) applies to radar_acquisition;
      Allowed_Processor_Binding => (reference (ecu1)) applies to speed_estimate;
      Allowed_Processor_Binding => (reference (ecu1)) applies to position_voter;
      Allowed_Processor_Binding => (reference (ecu1)) applies to
obstacle_detection;
      Allowed_Processor_Binding => (reference (ecu2)) applies to
obstacle_distance_evaluation;
      Allowed_Processor_Binding => (reference (ecu2)) applies to
speed_threshold_calculation;
      Allowed_Processor_Binding => (reference (ecu2)) applies to speed_controller;
      Allowed_Processor_Binding => (reference (ecu2)) applies to
emergency_detection;
      Allowed_Processor_Binding => (reference (ecu2)) applies to
warning_activation;
end integration.implementation1;


system implementation integration.implementation2 extends integration.generic
subcomponents
      ecu1 : processor speed_regulation::platform::ecu_rs232_one_connector;
      ecu2 : processor speed_regulation::platform::ecu_rs232_one_connector;
      ecu3 : processor speed_regulation::platform::ecu_rs232_two_connectors;
```

```
      bus1  : bus speed_regulation::platform::rs232;
      bus2  : bus speed_regulation::platform::rs232;
connections
      b0 : bus access bus1 <-> ecu1.socket;
      b1 : bus access bus2 <-> ecu2.socket;
      b2 : bus access bus1 <-> ecu3.socket1;
--    b3 : bus access bus2 <-> ecu3.socket2;
      b4 : bus access bus1 <-> obstacle_camera.socket2;
      b5 : bus access bus1 <-> obstacle_radar.socket2;
      b6 : bus access bus1 <-> wheel_sensor.socket2;
      b7 : bus access bus1 <-> laser_sensor.socket2;
      b8 : bus access bus1 <-> gps1.socket2;
      b9 : bus access bus1 <-> gps2.socket2;
      b10 : bus access bus2 <-> warning_alert.socket2;
      b11: bus access bus2 <-> brake.socket2;
      b12: bus access bus2 <-> acceleration.socket2;
properties
      Actual_Connection_Binding => (reference (bus1)) applies to c080;
      Actual_Connection_Binding => (reference (bus1)) applies to c120;
      Actual_Connection_Binding => (reference (bus1)) applies to c130;
      Actual_Connection_Binding => (reference (bus1)) applies to c140;
      Actual_Connection_Binding => (reference (bus1)) applies to c150;
      Actual_Connection_Binding => (reference (bus2)) applies to c180;
      Actual_Connection_Binding => (reference (bus2)) applies to c190;
      Actual_Processor_Binding => (reference (ecu1)) applies to image_acquisition;
      Actual_Processor_Binding => (reference (ecu1)) applies to radar_acquisition;
      Actual_Processor_Binding => (reference (ecu1)) applies to speed_estimate;
      Actual_Processor_Binding => (reference (ecu2)) applies to position_voter;
      Actual_Processor_Binding => (reference (ecu1)) applies to obstacle_detection;
      Actual_Processor_Binding => (reference (ecu3)) applies to
obstacle_distance_evaluation;
      Actual_Processor_Binding => (reference (ecu3)) applies to
speed_threshold_calculation;
      Actual_Processor_Binding => (reference (ecu3)) applies to speed_controller;
      Actual_Processor_Binding => (reference (ecu3)) applies to
emergency_detection;
      Actual_Processor_Binding => (reference (ecu3)) applies to warning_activation;
end integration.implementation2;



end speed_regulation::integration;
```