Validation de modèles dans un cadre d'IDM dédié à la conception de systèmes sur puce

Asma Charfi** — **Abdoulaye Gamatié*** — **Antoine Honoré*** — **Jean-Luc Dekeyser*** — **Mohamed Abid****

** LIFL - UMR CNRS 8022, Université de Lille, INRIA*
Parc Scientifique de la Haute Borne
Park Plaza - Bâtiment A, 40 avenue Halley
59655 Villeneuve d'Ascq Cedex France
{abdoulaye.gamatie, antoine.honore, jean-luc.dekeyser}@lifl.fr

*** CES, ENIS Université de Sfax*
Route Sokra km 3,5
B.P:w. 3038 Sfax-Tunisie
chrafiasma@yahoo.fr, mohamed.abid@enis.rnu.tn

RÉSUMÉ. *Cet article présente un travail réalisé dans un environnement, appelé Gaspard2, dédié à la conception de systèmes embarqués sur puce à hautes performances. La conception dans Gaspard2 repose sur l'ingénierie dirigée par les modèles. Des transformations de modèles permettent de passer d'un niveau d'abstraction à un autre jusqu'à la production de code ou la synthèse de matériel. Afin de réduire les erreurs dans les modèles transformés, nous proposons la validation de leur intégrité depuis le modèle initial dans la chaîne de transformation. Pour cela, nous utilisons le langage OCL pour spécifier les propriétés que doivent vérifier les modèles considérés corrects dans Gaspard2. Ensuite, nous développons un outil, appelé GaspardValidation, permettant à l'utilisateur de vérifier la correction des modèles définis. Ainsi, ce travail contribue à une conception plus sûre des systèmes embarqués dans Gaspard2.*

ABSTRACT. *This paper presents a work within an environment, called Gaspard2, dedicated to the design of high-performance embedded system-on-chip. The design in Gaspard2 relies on model-driven engineering. Some model transformations enable to go from a level of abstraction to another one until code generation or hardware synthesis. In order to reduce errors in the transformed models, we propose the validation of their integrity from the initial model in the transformation chain. For that, we use the OCL language to specify the properties that must verify the models considered as correct in Gaspard2. Then, we develop a tool, called GaspardValidation, allowing the user to verify the correctness of defined models. This work therefore contributes to a safer design of embedded systems within Gaspard2.*

MOTS-CLÉS : *Gaspard2, contraintes OCL, système sur puce, outil de validation*

KEYWORDS: *Gaspard2, OCL constraints, system-on-chip, validation tool*

1. Introduction

L'évolution très rapide des technologies d'intégration de circuits et la croissance du nombre de transistors (de l'ordre du milliard sur une puce à l'horizon 2010) rendent possible la réalisation de circuits numériques aux fonctionnalités toujours plus nombreuses. Aujourd'hui, les concepteurs peuvent intégrer tout un système logiciel/matériel sur la même puce ; on parle alors de système sur puce ou System-on-Chip (SoC). La partie matérielle est constituée de microprocesseurs, de mémoires, de périphériques d'interface et d'autres composants nécessaires à la réalisation de la fonction attendue. La partie logicielle quant à elle est formée de blocs fonctionnels pouvant exister déjà, appelés en général Intellectual Property (IP).

La complexité grandissante de ces systèmes rend nécessaire la possibilité d'aborder leur conception à des niveaux élevés d'abstraction. En effet, les gains en surface, temps ou consommation qu'il est possible d'obtenir lors des transformations, tant algorithmiques qu'architecturales, pourraient être proportionnels au niveau d'abstraction considéré. Pour réduire les coûts de développement et augmenter l'évolutivité, l'Ingénierie Dirigée par les Modèles (IDM) représente une véritable alternative. Cette approche s'appuie principalement sur le langage UML et sur l'initiative MDA (Model-Driven Architecture) dont le principe consiste en l'élaboration de modèles indépendants de toutes plates-formes et leur spécialisation via des transformations pour l'implémentation effective des systèmes (OMG, 2003).

D'autre part, les outils de conception des SoC doivent impérativement s'appuyer sur une implémentation modulaire et flexible, basée sur des composants génériques et réutilisables. C'est dans ce cadre que l'environnement Gaspard2 (Graphical Array Specification for Parallel and Distributed Computing, version 2) a été développé. Celui-ci fournit un support pour la conception basée sur l'IDM et la réutilisation, des SoC à hautes performances. Comme illustré sur la Figure 1, il propose l'utilisation de modèles définis à différents niveaux d'abstractions et offre des transformations systématiques des modèles, qui réalisent des transitions entre les niveaux d'abstraction, suivant un processus de raffinement aboutissant à la génération du code de simulation ou de synthèse matérielle.

Au plus haut niveau, le concepteur décrit l'application logicielle et l'architecture matérielle sur laquelle l'application sera exécutée ou simulée. A partir de ces deux modèles, il est alors possible de définir comment les composants de l'application sont répartis sur l'architecture. Cette phase, dite d'association, est décrite à l'aide d'un modèle appelé aussi association. Les modèles d'application, architecture et association sont complètement indépendants de toute technologie d'implémentation. Le modèle résultant de l'association est ensuite enrichi avec des informations sous forme d'IP concernant son déploiement sur des plates-formes spécifiques. Le modèle ainsi déployé peut subir diverses transformations selon le code cible recherché : langages synchrones pour la vérification formelle, Fortran OpenMP pour le calcul parallèle, SystemC pour la simulation et VHDL pour la synthèse de matériel.

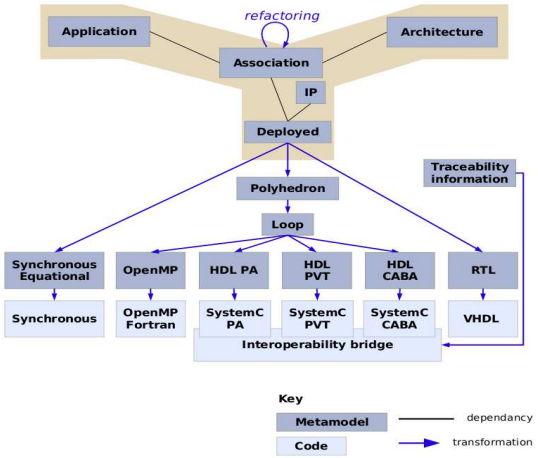

Figure 1. *Flot de conception Gaspard2.*

Le passage d'un niveau d'abstraction à un autre est susceptible d'entraîner des pertes d'informations et même des erreurs de compilation qui nécessiteraient d'effectuer des retours à des niveaux plus haut dans la chaîne de conception. Cela augmente le temps de conception des SoC. Pour diminuer ce délai et pour minimiser le nombre d'erreurs, nous proposons de valider très tôt l'intégrité des modèles manipulés lors de la conception, à l'aide du langage OCL (Object Constraint Language). Nous avons ainsi étendu l'environnement Gaspard2 avec un nouvel outil de validation de modèles, appelé *GaspardValidation*, qui est présenté en détail dans cet article.

Dans ce qui suit, la section 2 introduit le profil Gaspard2. Ensuite, la section 3 présente les différentes propriétés de modèles Gaspard2 sous forme de contraintes OCL validées à l'aide de l'outil *GaspardValidation* décrit en section 4. La section 5 discute quelques travaux connexes. Enfin, la section 6 donne les conclusions et perspectives.

2. Profil Gaspard2

Le langage de modélisation des applications Gaspard2 est UML. Par nature, un modèle UML possède une sémantique trop générale. Il faut donc spécialiser UML pour l'adapter à un domaine bien déterminé, à l'image de la proposition MARTE[1] pour le temps-réel. Ce besoin d'extension est rendu possible via la notion de *profil*, qui correspond au regroupement d'extensions et de spécialisations du langage UML du point de vue de la notation et de la sémantique. Cela est principalement réalisé à l'aide du concept de stéréotype qui représente la définition d'une propriété supplémentaire appliquée à un élément standard d'UML : classe, association ... (OMG, 2007).

1. http ://www.omgmarte.org

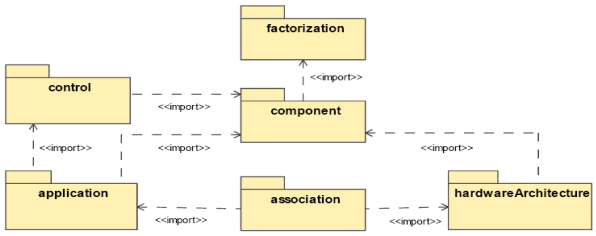

Figure 2. *Profil Gaspard2.*

Le profil Gaspard2 dédié aux SoC (Ben Atitallah *et al.,* 2007) est schématisé sur la Figure 2. Il s'articule autour de six paquetages ou *packages*. Nous ne détaillons que ceux qui sont nécessaires à la compréhension de notre propos dans ce papier.

2.1. *Les différents packages*

Package component. Ce package offre un support pour la méthodologie orientée composant en permettant la représentation de composants indépendamment de l'environnement d'utilisation. Il favorise la réutilisation d'IP logiciels et matériels.

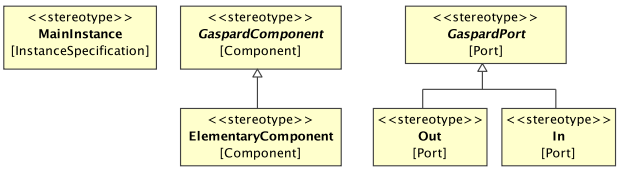

Figure 3. *Vue du package component.*

La notion la plus importante dans ce package est celle du stéréotype abstrait GaspardComponent (Figure 3). Un composant de ce type est vu comme une adaptation ou une restructuration des mécanismes utilisés pour la définition d'un composant en UML 2. Chaque composant dans Gaspard2 est donc considéré comme un élément indépendant qui peut communiquer avec son environnement externe via la notion de port, et dont sa structure et sa composition hiérarchique peuvent être définies via la notion de *part* et de *connecteur*. Un composant de type ElementaryComponent est utilisé pour représenter un composant vu en général comme une boîte noire. Un stéréotype abstrait GaspardPort permet de spécifier le type et la direction des ports, In ou Out.

Package factorization. Les mécanismes de factorisation décrits dans ce package sont fortement inspirés du langage Array-OL (Array-Oriented Language), le formalisme sous-jacent de spécification de Gaspard2 (Boulet, 2007). Ils assurent une expression compacte du parallélisme potentiel des applications et des architectures, ainsi que la

topologie des connexions entre composants : caractéristiques des connecteurs, notion de multiplicité des ports et relation entre tâches de calcul et données d'entrée / sortie.

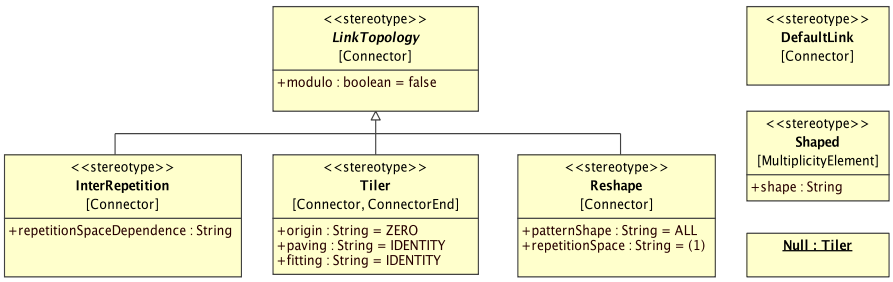

Figure 4. *Vue du package factorization.*

La description du package factorization (Figure 4) est basée sur la définition des topologies de liens principalement utilisées pour l'expression de la répétition dans les modèles Gaspard2. La définition des différentes topologies de lien est introduite via le stéréotype abstrait LinkTopology qui étend la métaclasse Connector d'UML 2. Les stéréotypes dérivant de LinkTopology sont Tiler, Reshape et InterRepetiton.

Le concept de Tiler étend les métaclasses Connector et ConnectorEnd d'UML 2. Il est utilisé pour la spécification d'un lien particulier de répétition basé sur le langage Array-OL qui manipule les tableaux comme structure de données. Ainsi, chaque tâche d'une application consomme des tableaux en les découpant en morceaux de données de même taille, appelés *motifs*. Les informations de calcul de ces motifs sont regroupées dans une structure appelée Tiler : *origin* (origine du motif référence), *paving* (matrice de pavage, décrivant comment les motifs couvrent le tableau), *fitting* (matrice d'ajustage, décrivant comment remplir les motifs avec les éléments du tableau), *RepetitionSpace* (espace de répétition, définissant le nombre total de motifs calculés), *Spattern* (la taille du motif) et *Sarray* (la taille du tableau d'entrée).

Le stéréotype Reshape étend la métaclasse Connector d'UML 2. Il permet la représentation des topologies de liens plus complexes dans lesquelles les éléments d'un tableau multidimensionnel sont redistribués dans un autre tableau. La différence majeure entre Tiler et Reshape est que le premier est utilisé pour les connecteurs de délégation (entre un port d'un composant père et un port d'une de ses *parts*) et que le deuxième est utilisé pour les connecteurs qui relient deux *parts*. Le stéréotype InterRepetion quant à lui est utilisé pour spécifier une dépendance de données entre tâches répétées(une tâche répétée est une tâche dont l'exécution est assurée par la répétition d'instances d'elle-même). Enfin, à droite sur la Figure 4, sont représentés d'autres composants facilitant la spécification de modèles complexes de systèmes. Ceux-ci ne seront pas détaillés ici.

Package application. Le package application, illustré par la Figure 5, définit les stéréotypes utilisés pour modéliser la partie logicielle du système. Le stéréotype principal est ApplicationComponent dont dérivent deux autres : ArrayProducer et ArrayConsumer. Le premier est introduit pour structurer les données consommées par les tâches tandis que le deuxième récupère les données produites par les tâches.

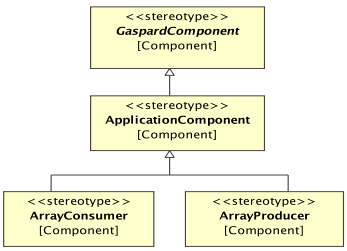

Figure 5. *Vue du package application.*

Les autres packages. Le package hardwareArchitecture définit tous les composants nécessaires à la description de l'architecture matérielle du système : Processeur, mémoire, bus, etc. Le stéréotype de base est HardwareComponent. Les différents stéréotypes dérivant directement du HardwareComponent sont : Memory pour décrire la mémoire, Processeur pour modéliser un processeur, Communication pour représenter les différentes connexions matérielles et IO pour modéliser les entrées / sorties. Le package association fournit les outils nécessaires pour établir un lien entre une application et une architecture matérielle associée. Enfin, le package control permet la prise en compte des notions de contrôle et de la spécification de modes de fonctionnement dans les applications modélisées dans Gaspard2.

2.2. *Exemple : modélisation d'une multiplication matricielle*

Pour illustrer la modélisation à l'aide du profil Gaspard2, nous présentons un exemple simple réalisé avec l'outil MagicDraw[2], un outil visuel de modélisation UML. Il s'agit d'une multiplication de deux matrices. L'algorithme correspondant est modélisé à l'aide du package application. Soient A et B deux matrices d'entrées et C celle de sortie. Chacune des trois matrices comprend 16 lignes et 16 colonnes. Deux composants élémentaires nommés respectivement *Multiplication* et *SumVector* sont introduits pour réaliser respectivement la multiplication de deux scalaires réels et la somme de tous les éléments d'un vecteur. Ces composants sont stéréotypés à la fois ApplicationComponent et ElementaryComponent (Figure 6).

Un autre composant, appelé *MultV*, est introduit pour modéliser la multiplication d'un vecteur ligne de A avec un vecteur colonne de B (Figure 7). Il génère un vecteur

2. http ://www.magicdraw.com

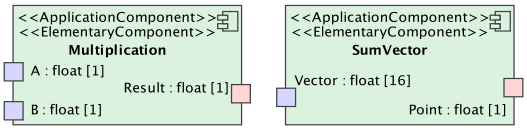

Figure 6. *Composants élémentaires dans la multiplication matricielle..*

de 16 réels à partir de deux vecteurs également de 16 réels. Il contient une *part* de type multiplication nommée lxc (pour " line x column ") qui calcule à chaque fois le produit de deux réels. Ce calcul est répété 16 fois, donc l'attribut *multiplicity* de cette *part* est fixé à 16. La valeur ZERO, ou vecteur nul, de l'attribut *origin* indique que le motif de référence commence au point de coordonnées (0,0). La valeur IDENTITY de l'attribut *paving*, de type matrice, indique que les motifs couvrent tous les éléments du vecteur d'entrée un par un. La valeur IDENTITY de *fitting* indique que le premier motif est formé du premier élément du vecteur d'entrée, le second est formé du second élément et ainsi de suite.

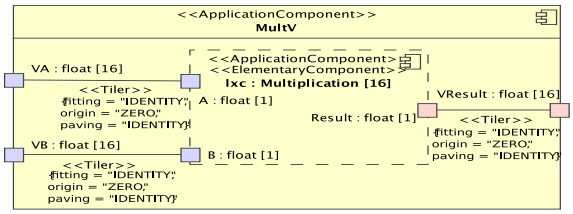

Figure 7. *Composant MultV.*

Le composant *SumRbyC* (Figure 8) permet de calculer à chaque utilisation un seul élément de la matrice de sortie en effectuant la somme de tous les éléments du vecteur généré par la *part* lxc. Cette somme s'effectue en appliquant la *part* nommée s de type *SumVector*. Ce composant va être utilisé 16x16 fois puisque la matrice de sortie C est une matrice [16,16]. *SumRbyC* est utilisé dans *MatrixMultiplication* en tant que sous composant permettant de calculer ainsi tous les éléments de la matrice C (Figure 8).La taille d'un motif est indiquée par la multiplicité du port lié à la sortie du connecteur stéréotypé Tiler. Par exemple, pour le connecteur qui relie le port A et le port Line, la taille du motif est 16. Ainsi, la valeur [1,0] de l'attribut *fitting* du Tiler appliquée à ce connecteur indique que le premier motif n'est que la première ligne de la matrice A. la valeur [(0,0),(0,1)] du *paving* indique que le prochain motif est la deuxième ligne, et ainsi de suite.

Le composant principal de l'application (Figure 9) contient une seule *part* de type *MatrixMultiplication* qui prend en entrée deux matrices A et B de taille [16,16] et

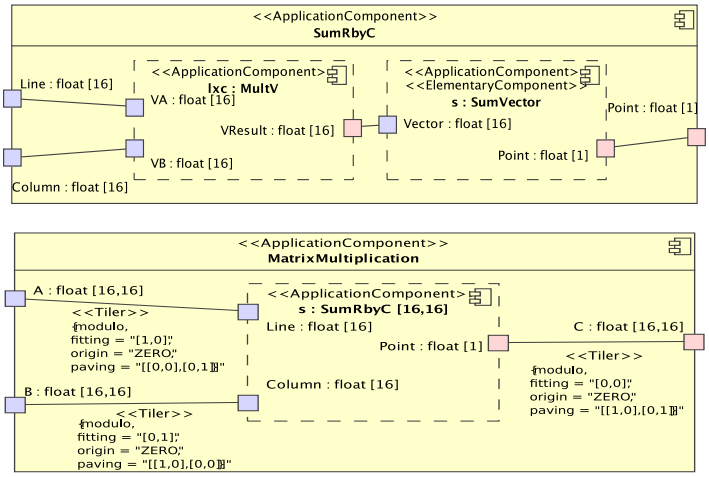

Figure 8. *Composants SumRbyC et MatrixMultiplication.*

génère en sortie une matrice C=A * B. Ainsi, la multiplication de matrices aura été modélisée à l'aide de six composants dont deux élémentaires.

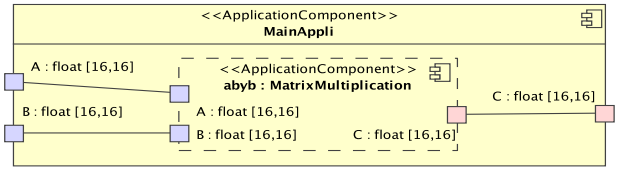

Figure 9. *Composant MainAppli.*

3. Validation de modèles dans Gaspard2

En plus de la notion de stéréotype mentionnée précédemment, un profil UML est également composé de *tagged value*. Cette notion est utilisée pour définir des propriétés additionnelles de certains éléments et elle peut être définie pour des éléments existants ou des stéréotypes. Enfin, un profil comprend aussi des *contraintes*, utilisées pour étendre la sémantique d'UML par l'ajout de nouvelles règles ou la modification des règles existantes. Le langage qui permet d'exprimer les contraintes au niveau des modèles UML est OCL.

Pour présenter les contraintes que nous avons définies dans le profil Gaspard2, nous adoptons une classification basée sur le but de chacune d'entre elles. Nous avons identifié des contraintes de construction de modèles liées au profil lui-même ; elles

forment la classe nommée **P** (pour **Profil**). D'autres contraintes ont été identifiées à partir du principe méthodologique de la conception conjointe logicielle / matérielle des SoC ; elles constituent quant à elles la classe **M** (pour **Méthodologie**). Les SoC à hautes performances visés par Gaspard2 ont eux aussi des caractéristiques intrinsèques (par exemple, un composant stéréotypé RAM ne peut pas contenir une *part* de type Processeur...) ; les contraintes liées à ces propriétés forment la classe nommée **S** (pour **Système-sur-puce**). Enfin, l'utilisation du langage Array-OL comme formalisme sous-jacent de Gaspard2 induit une dernière classe de contraintes liées à la sémantique de ce langage ; ces contraintes forment la classe nommée **A** (pour **Array-OL**). Par la suite, nous parlerons de **MAPS** pour désigner cette classification.

Dans ce qui suit, les différentes contraintes sont présentées par packages : pour chaque package, nous identifions ses contraintes et pour chacune d'elles, nous précisons le contexte, la classe, le but et l'expression correspondante en OCL. Les contraintes OCL peuvent être ajoutées aux modèles UML en utilisant le niveau M1 (niveau modèle) ou bien le niveau M2 (niveau métamodèle). Nous avons choisi d'utiliser le niveau M2. Au niveau M1, on serait obligé d'utiliser les extensions base_Component et extension_StereotypeName à chaque fois que l'on voudra accéder respectivement aux attributs et aux stéréotypes d'un composant.

3.1. *Package component*

Contexte GaspardComponent. La première contrainte impose que tout composant, dont le stéréotype dérive du stéréotype GaspardComponent, ne peut contenir que des *parts* dont le type dérive aussi du même stéréotype GaspardComponent. Elle est donc de type P et exprimée en OCL comme suit :

```
let parts :Set(Property) = self.ownedAttribute-self.ownedPort in
parts.type->forAll(p|p.getAppliedStereotypes().generalization.general->
exists(name='GaspardComponent'))
```

Un composant de type GaspardComponent peut avoir plus qu'un stéréotype : ElementaryComponent avec ApplicationComponent, ElementaryComponent avec HardwareComponent, etc. Mais stéréotyper un composant à la fois ApplicationComponent et HardwareComponent est strictement interdit selon les principes de la conception conjointe matérielle logicielle. Cette règle fait partie de la classe M, elle est exprimée par la contrainte OCL suivante :

```
not(self.getAppliedStereotypes().name->includesAll(Set
{'HardwareComponent','ApplicationComponent'}))
```

Contexte ElementaryComponent. D'après le profil Gaspard, un composant stéréotypé ElementaryComponent ne doit contenir aucune *part*. Cette contrainte fait partie de la classe P et elle est exprimée en OCL comme suit :

```
(self.ownedAttribute-self.ownedPort)->isEmpty()
```

Contexte GaspardPort. Le type des ports est une notion très délicate dans la mesure où les applications modélisées par Gaspard2 manipulent des données multidimensionnelles de types variés (float, complex, integer...). Le type DataType convient le mieux dans notre situation. En effet, les types UMLPrimitivesTypes et Enumeration sont supportés par le DataType. De même, en utilisant DataType, on peut définir de nouveaux types (par exemple le type complex). Nous considérons donc la contrainte OCL suivante, qui impose que tous les ports soient typés DataType ou ses dérivés. Cette contrainte fait partie de la classe S puisqu'elle permet de définir des données multidimensionnelles souvent utilisées dans les applicatifs des SoC.

```
self.type.oclIsKindOf(DataType)
```

3.2. *Package factorization*

Contexte LinkTopology. Un connecteur relie deux ports en permettant le passage des données d'un port à un autre. Il est alors indispensable que ces ports aient le même type. Cela est exprimé par la contrainte OCL suivante :

```
self.end.role->forAll(x,y| x.type=y.type)
```

Cette contrainte fait partie de la classe P puisqu'elle est liée à la construction des modèles selon les concepts UML.

Contexte Tiler. Le type commun au trois tagged value (*origin*, *paving* et *fitting*) associées à un Tiler est le type String alors qu'en réalité, *fitting* et *paving* représentent des matrices et *origin* représente un vecteur. Nous avons donc ajouté une contrainte OCL qui vérifie que le concepteur a bien respecté la forme d'une matrice en saisissant les valeurs de *fitting* et *paving*. Cette contrainte est de classe A. Son expression OCL relative à la valeur de *fitting* est la suivante :

```
let s : String = self.getValue(self.getAppliedStereotype
('Gaspard : :Tiler'),'origin').oclAsType(String)in
s.verif_form('\(\([0 − 9] + (, [0 − 9]+)*\)(, \([0 − 9] + (, [0 − 9]+)*\))*\)$')=true
```

En changeant `'fitting'` par `'origin'` et le paramètre d'appel de l'opération *verif_form* dans la contrainte ci-dessus par $'(\backslash([0-9]+(,[0-9])^*\backslash)\backslash)\$'$, nous obtenons une nouvelle contrainte qui vérifie la forme du tagged Value *origin*. Cette contrainte illustre deux points forts du langage OCL. Le premier consiste à déclarer et à définir la valeur d'un attribut qui pourra être utilisée dans d'autres expressions OCL (grâce à let...in). Le second est l'extensibilité de l'environnement OCL[3], permettant la définition de nouvelles opérations propres au concepteur, et qui s'appliquent à des types de son choix. Dans notre cas, l'opération s'appelle *verif_form* et elle est applicable à toute chaîne de caractères. Elle sera détaillée dans la section 4.

3. http ://wiki.eclipse.org/CustomizingOclEnvironments

Nous avons ajouté une autre opération *verif_dim*, qui permet de vérifier les dimensions des tagged value *origin*, *paving* et *fitting*. En effet, le langage Array-OL impose que le nombre de lignes de *fitting* et *paving* ainsi que le nombre d'éléments de *origin* soient égale à *Spattern* (section 2.1) et que le nombre de colonnes de *fitting* respectivement de *paving* soit égal à la dimension de *Sarray* respectivement *RepetitionSpace*.

L'utilisation du stéréotype Tiler est liée au concept de répétition, il est donc interdit d'utiliser un Tiler alors que l'attribut *multiplicity* de la *part* en question n'est pas spécifié (Figure 10). Cette contrainte est de classe A puisqu'elle est liée au concept de répétition défini dans le langage Array-OL. Elle est exprimée comme suit :

```
self.end->select(c|not(c.partWithPort.oclIsUndefined())).partWithPort->
exists(p|not(p.upperValue.oclIsUndefined() or
p.lowerValue.oclIsUndefined()))
```

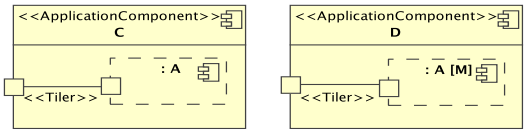

Figure 10. *Utilisation interdite du Tiler (à gauche) et utilisation légale (à droite).*

Comme nous l'avons déjà mentionné, un Tiler n'est utilisé que pour les connecteurs de délégation (Figure 11). Cela est traduit en OCL par la contrainte suivante :

```
self.end->exists(partWithPort.oclIsUndefined())
```

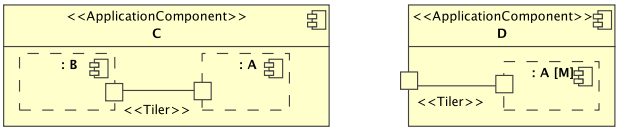

Figure 11. *Utilisation interdite du Tiler (à gauche) et utilisation légale (à droite).*

Cette contrainte exprime un invariant qui impose qu'à chaque utilisation d'un Tiler, il faut que l'une des extrémités du connecteur ne soit pas liée à une *part*. Elle est de classe P puisqu'elle est liée à la définition du stéréotype Tiler introduit dans le package factorisation du profil Gaspard. La même Figure 11, montre que les ports liés par un connecteur stéréotypé Tiler doivent être de même direction (soit les deux In, soit les deux Out). Cette contrainte de classe P est exprimée en OCL comme suit :

```
not(self.end.role.getAppliedStereotypes().name->includesAll
(Set'In','Out'))
```

Une autre contrainte de même genre a été ajoutée au stéréotype Reshape qui exige que les ports reliés par un tel connecteur doivent avoir des directions opposées.

Contexte Reshape. Toutes les contraintes relatives à ce contexte sont de classe P. Elles sont étroitement liées à celles déjà exprimées pour les Tilers. Ainsi, si un Tiler n'est utilisé que pour les connecteurs de délégation, l'utilisation du Reshape pour ce même type de connecteur est strictement interdite. Cela est exprimé par une contrainte OCL de type invariant qui impose que les deux extrémités du connecteur (Connector End) doivent être reliées à des *parts* :

```
self.end->forAll(p|not(p.partWithPort.oclIsUndefined()))
```

Nous pouvons spécifier une nouvelle règle liée à ces deux ConnectorEnd qui exprime le fait qu'ils doivent être stéréotypés Tiler. Cela permet de déterminer comment sélectionner les éléments du tableau source et comment les ranger dans le tableau destination. La contrainte OCL correspondante à cette règle est la suivante :

```
self.end->forAll(p|p.getAppliedStereotypes()->exists(name='Tiler'))
```

3.3. *Package application*

Contexte ApplicationComponent. ApplicationComponent est un stéréotype qui dérive du stéréotype abstrait GaspardComponent. Par conséquent, toutes les contraintes déjà exprimées dans le contexte GaspardComponent sont appliquées à ce nouveau stéréotype. L'ajout de nouvelles contraintes s'avère nécessaire pour exploiter au maximum les nouveaux concepts introduits dans ce package. Mais celles-ci ne doivent pas briser le lien de généralité entre ApplicationComponent et GaspardComponent. Elles doivent plutôt renforcer cette relation de généralisation.

Ainsi, nous avons ajouté une contrainte plus spécifique que celle déjà exprimée dans GaspardComponent stipulant que toutes les *parts* d'un composant de type GaspardComponent doivent être elles aussi de type GaspardComponent. La nouvelle contrainte exige que toutes les *parts* d'un composant stéréotypé ApplicationComponent soient stéréotypées ApplicationComponent (plus spécifique que GaspardComponent) ou ses dérivées :

```
let parts :Set(Property) = self.ownedAttribute-self.ownedPort in
parts.type->forAll(p|(p.getAppliedStereotypes.generalization.general->
exists(name='ApplicationComponent')or p.getAppliedStereotypes()->exists
(name='ApplicationComponent')))
```

Cette contrainte est de classe M car elle assure la séparation des deux concepts application et architecture. Le même type de contrainte a été ajouté au package HardwareArchitecture dans le contexte HardwareComponent.

Contextes ArrayProducer et ArrayConsumer. ArrayProducer a été introduit pour structurer les données en entrée consommées par les premières tâches, alors que ArrayConsumer permet de récupérer les données en sortie produites par les dernières tâches. Par conséquent, ArrayProducer ne doit avoir que des ports de sortie et ArrayConsumer ne doit avoir que des ports d'entrée. Ces deux contraintes sont de classe P

car elles sont liées à la définition même de ces deux stéréotypes. La contrainte OCL associée à la contrainte relative au contexte ArrayProducer est la suivante :

```
self.ownedPort.getAppliedStereotypes().name->includes('Out')and
self.ownedPort.getAppliedStereotypes().name->excludes('In')
```

Une contrainte similaire a été ajoutée au contexte ArrayConsumer.

3.4. *Package hardwareArchitecture*

Contexte Memory. Le stéréotype Memory dérive du stéréotype HardwareComponent. Il permet de modéliser les différents types de mémoires : RAM, ROM et Cache. Un composant de type Memory ne peut contenir que des *parts* de type Memory, RAM, ROM ou Cache. Par exemple, cette contrainte interdit d'intégrer un processeur dans une mémoire. Elle est donc de type S. Son expression OCL ressemble à celle de la première contrainte définie dans 3.3 (contexte ApplicationComponent).

Contexte Sensor et Actuator. Un composant de type IO est introduit pour modéliser les entrées/sorties au niveau architecture. De ce stéréotype dérive deux autres qui sont Sensor (analogue à ArrayConsumer) et Actuator (analogue à ArrayConsumer). La même contrainte OCL définie dans les contextes ArrayProducer et ArrayConsumer (section 3.3) a été ajoutée respectivement aux contextes Actuator et Sensor.

La Figure 12 résume la répartition des contraintes selon notre classification MAPS. Nous avons identifié 24 contraintes OCL dont 12 de classe P, 7 de classe A, 3 de classe M et uniquement 2 de classe S. Il y apparaît clairement des contraintes, davantage

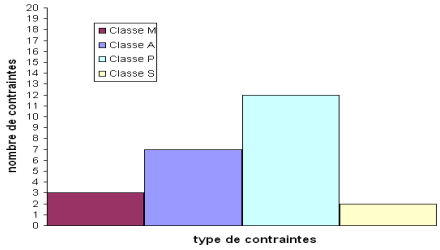

Figure 12. *Répartition des contraintes sur les classes.*

liées aux règles de construction de modèles suivant les formalismes de spécification (i.e. Gaspard/Array-OL) plutôt qu'à la sémantique à proprement parler du domaine visé (i.e. codesign des SoC). Cette observation s'explique en partie par le fait qu'aux niveaux application et architecture du flot Gaspard2 (où nous nous situons), les propriétés des SoC ne sont pas encore très explicites. En fait, celles-ci le deviennent progressivement lors des raffinements dans le flot de conception (cf Figure 1), notamment au niveau du déploiement vers la plate-forme cible retenue. Une perspective intéressante de notre étude consisterait donc à étendre la spécification des contraintes aux

couches basses dans le flot. Certaines contraintes de type S font références à des valeurs connues seulement pendant l'exécution (consommation d'énergie, temps d'exécution...). Puisque OCL est un langage statique, toutes les questions d'exécution des modèles sont évacuées de sa portée. Ainsi, Ces contraintes, qualifiées de non fonctionnelles, ne peuvent être exprimées en OCL ce qui explique encore le nombre réduit de contraintes OCL de type S.

La classe regroupant le plus grand nombre de contraintes est la classe P. cela peut être expliqué par le fait qu'un profil n'est qu'un mécaniseme d'extension lié au métamodèle UML. On peut alors ajouter beaucoup plus de contraintes au profil qu'au métamodèle puisque le premier n'est qu'une spécialisation du second pour un domaine particulier. En effet, un profil UML spécialise davantage le métamodèle en ajoutant de nouveaux stéréotypes qui peuvent représenter des nouveaux contextes pour de nouvelles contraintes.

4. Outil de validation

Cette section présente l'outil que nous avons développé pour valider les modèles décrits à l'aide du profil Gaspard2 étendu avec les contraintes précédentes. Notons tout d'abord que la version actuelle de Gaspard2 est disponible sous forme d'un *plugin* Eclipse, c'est-à-dire, un module qui se greffe à Eclipse pour lui ajouter une nouvelle fonctionnalité. Notre outil de validation est également un plugin Eclipse.

4.1. *Architecture générale de l'outil*

Une caractéristique essentielle de la plate-forme Eclipse est l'extensibilité de l'environnement assurée par la notion de plugin. La structure d'un plugin peut être résumée de la façon suivante : c'est un fichier JAR classique contenant, en plus de ses classes java, deux autres fichiers, le fichier MANIFEST.MF et le fichier plugin.xml. En exploitant les informations contenues dans le premier fichier, le noyau d'Eclipse gère le cycle de vie des plugins et leurs dépendances. Le deuxième fichier sert à concrétiser les fonctionnalités d'extensibilité de la plate-forme. Via ce fichier, des plugins déclarent des points d'extension auxquels d'autres plugins se branchent. Dans notre cas, l'outil *GaspardValidation* utilise les points d'extension org.eclipse.uml2.uml.editor et org.eclipse.ocl.ecore déclarés respectivement par les plugins UML et OCL.

En utilisant l'interface org.eclipse.ocl.ecore.EcoreEnvironmentFactory, nous pouvons étendre l'environnement d'analyse des contraintes OCL et ajouter ainsi l'opération *verif_form* (cf section 3.2). Nous définissons une classe MyEnvironmentFactory qui implémente cette interface. Cette classe permet de créer deux environnements : MyEnvironnement et MyEvaluationEnvironment. Le premier permet d'analyser la syntaxe des contraintes OCL et le deuxième permet d'évaluer ces contraintes sur les modèles (Figure 13). Ainsi, notre nouvelle opération *verif_form* a été ajoutée dans la classe MyEnvironnent en utilisant la méthode defineCustomOperations(). L'appel de

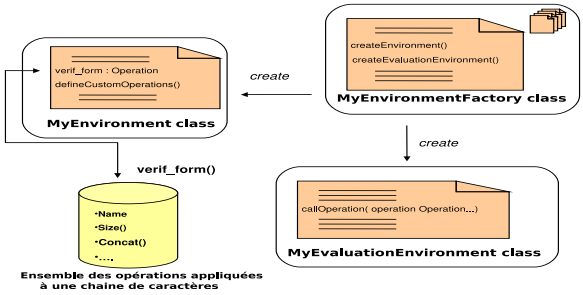

Figure 13. *Extension de l'environnement OCL.*

la méthode callOperation() de la classe MyEvaluationEnvironment permet d'exécuter le code défini dans le corps de la méthode defineCustomOperations().

4.2. *Utilisation de l'outil de validation dans la chaîne de conception Gaspard2*

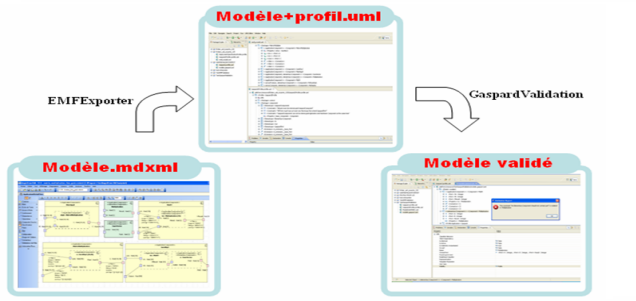

Figure 14. *Utilisation de GaspardValidation dans la chaîne de conception Gaspard2.*

Les contraintes OCL présentées dans la section 3 ont été ajoutées au profil Gaspard2 en utilisant la version 12.0 de l'outil MagicDraw. Une fois les contraintes ajoutées, le profil accompagné du modèle, est exporté vers Eclipse en utilisant l'outil EMFExporter[4], formant ainsi un fichier UML qui constitue le point d'entrée du plugin *GaspardValidation* (Figure 14). L'outil de validation *GaspardValidation* est développé au sein d'Eclipse de façon à pouvoir traiter tout fichier au format UML indépendamment de l'outil de modélisation ayant servi à le générer. Le choix de l'éditeur MagicDraw n'est pas exclusif. D'autres éditeurs tels que Papyrus[5] ou Topcased[6] peuvent également être envisagés.

4. Cet outil est une propriété de l'INRIA.
5. http ://www.papyrusuml.org
6. http ://www.topcased.org, etc.

Illustration concrète. Pour tester notre plugin sur l'exemple présenté dans la section 2.2, nous illustrons la violation des deux contraintes suivantes : *un composant élémentaire ne doit contenir aucune "part"* (C1) et *l'utilisation d'un Tiler doit être accompagnée d'une spécification de l'attribut "multiplicity" de la "part" en question* (C2). Dans la Figure 15, le composant *Multiplication* présente une violation de C1 tandis que le composant *MultV* présente une violation de C2. La Figure 16 illustre la violation des deux contraintes C1 et C2 dans le fichier UML obtenu suivant le schéma de la figure 14, ainsi que le résultat de l'invocation de l'outil de validation sur les modèles ci-dessus.

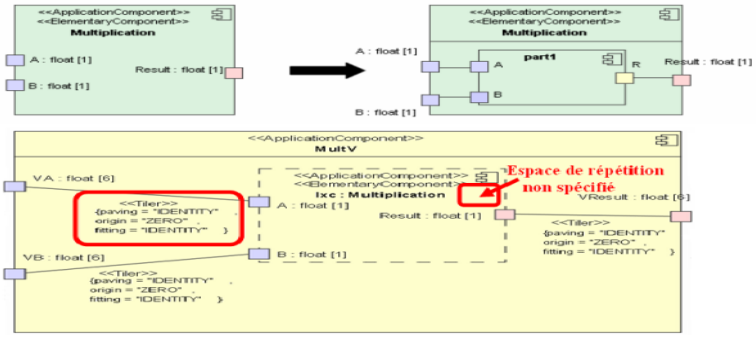

Figure 15. *Violation des contraintes C1 (en haut) et C2 (en bas).*

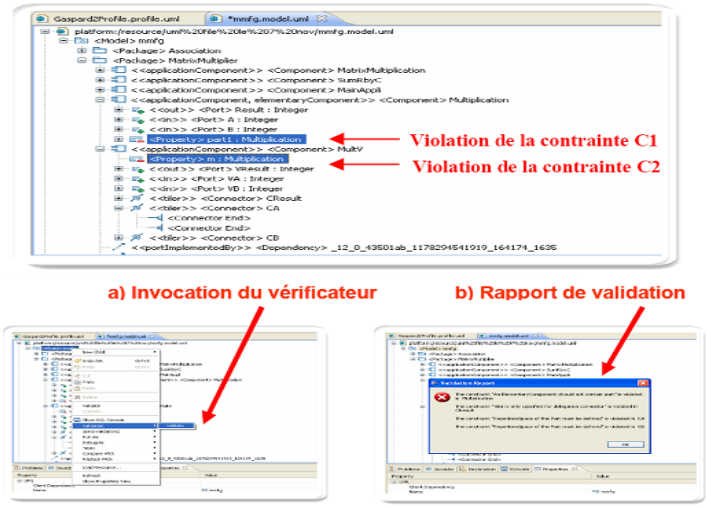

Figure 16. *Invocation de GaspardValidation et résultats d'évaluation de contraintes.*

5. Travaux relatifs

De nombreux outils[7] ont été développés pour assurer l'intégrité des modèles UML en analysant les contraintes OCL. Certains, **Objecteering** ou **Poseidon**, offrent un éditeur UML supportant OCL, mais sans la vérification des contraintes ; ces dernières étant simplement considérées comme des notes sur les modèles (Ratté, 2007). Ce n'est pas le cas des outils **USE**[8] et **ArgoUML**[9] qui eux supportent la vérification de contraintes OCL. Dans notre étude, nous nous sommes plutôt intéressés aux outils pouvant être intégrés dans Eclipse. La première alternative consiste à utiliser l'outil **EMF Validation**. Quant à la seconde, il n'existe pas d'outil à proprement parler ; il faut combiner les plugins **EMF** et **OCL**.

Le plugin **EMF Validation** faisait partie du projet EMFT qui a été lancé pour ajouter de nouvelles technologies qui complètent EMF. D'après les exemples fournis par les développeurs de ce plugin, il existe quatre méthodes pour valider les contraintes OCL (Damus, 2006). La plupart de ces méthodes sont inadaptées à notre besoin. D'une part, elles imposent à l'utilisateur la connaissance à l'avance des contraintes ainsi qu'une définition par lui-même du traitement à faire en cas d'échec ou succès. D'autre part, les contraintes manipulées doivent être spécifiées dans un fichier spécifique, or dans notre cas, on souhaite les extraire à partir du profil. Ainsi, parmi les quatre méthodes, seule une est susceptible de répondre à nos attentes. Elle consiste à écrire du code java permettant d'extraire les contraintes à partir d'un métamodèle. L'outil de validation de contraintes de l'environnement Papyrus utilise cette méthode en l'adaptant pour les profils.

La seconde alternative pour valider les modèles sous Eclipse consiste à utiliser EMF et le plugin OCL. Ce dernier fait partie du projet MDT. La solution proposée ici est similaire à la dernière méthode évoquée ci-dessus pour EMF Validation. C'est elle que nous avons privilégiée pour développer l'outil *GaspardValidation*.

6. Bilan et perspectives

Dans cet article, nous avons abordé la validation dans l'environnement Gaspard2 dédié à la conception de systèmes embarqués sur puce à l'aide de l'IDM. Gaspard2 permet d'effectuer des transformations de modèles de haut niveau, qui aboutissent à la génération de code et à la synthèse de matériel. Ces modèles sont décrits à l'aide d'un profil spécifique. Pour assurer l'intégrité des modèles transformés, nous avons enrichi le profil avec des contraintes OCL. La description de certaines contraintes a nécessité la définition d'opérations nouvelles, en plus de celles proposées par le langage OCL. Nous avons ensuite développé un outil de validation, appelé *GaspardValidation*, sous forme d'un plugin Eclipse. Cet outil présente l'avantage d'être générique et indépendant vis-à-vis du profil Gaspard2.

---

7. http ://www.um.es/giisw/ocltools/ocl.htm
8. http ://www.db.informatik.uni-bremen.de/projects/USE/
9. http ://argouml.tigris.org/

Pour améliorer la validation proposée actuellement dans Gaspard2, il serait intéressant d'étendre celle-ci avec davantage d'analyse sémantique liée aux SoC : propriétés d'architectures particulières ou d'exécutions. Cette perspective pourrait remettre en question l'utilisation de la version statique d'OCL. Une alternative serait alors l'extension xOCL (eXecutable OCL) de l'outil XMF-Mosaic (Clark *et al.,* 2004).

Une fois le modèle d'entrée du flot Gaspard (Application, Architecture, Association et Déploiement) est valide, on peut penser à étendre nos travaux de vérification pour le reste du flot de conception, et valider ainsi les différentes transformations de modèles (Figure 1). Selon les travaux de (Cariou *et al.,* 2004), Le langage de validation des transformations n'est pas forcément OCL. il fallait étudier les outils et les langages de transformations de modèles telque le langage QVT (Query / View / Transform) adopté en 2005 par l'OMG (OMG, 2005).

7. Bibliographie

Ben Atitallah R., Boulet P., Cuccuru A., Dekeyser J., Honoré A., Labbani O., Le Beux S., Marquet P., Piel E., Taillard J., Yu H., Gaspard2 uml profile documentation, Rapport technique n° 0342, INRIA, septembre, 2007.

Boulet P., Array-OL Revisited,Multidimensional Intensive Signal Processing Specification, Rapport de recherche n° 6113, INRIA, février, 2007.

Cariou E., Marvie R., Seinturier L., Duchien L., « OCL for the Specification of Model Transformation Contracts », *proceedings «UML» 2004 Workshop OCL and Model Driven Engineering, Lisbon, Portugal*, 2004.

Clark T., Evans A., Sammut P., Willans J., « Applied Metamodelling - A Foundation for Language Driven Development », *version 0.1.*, 2004.

Damus C., « EMFT 1.0 Release Review (OCL, Query, Transaction, and Validation) », *eclipse con*, 2006.

OMG, « MDA Guide Version 1.0.1 », *omg/2003-06-01*, 2003.

OMG, « Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification », *ptc/05-11-01*, 2005.

OMG, « OMG Unified Modeling Language(OMG UML),Superstructure », *V2.1.2*, 2007.

Ratté S., UML et OCL, Document thématique, Université de Québec,École de technologie supérieure, Département de Génie logiciel et des TI, 2007.