Modélisation UML/MARTE de SoC et analyse temporelle basée sur l'approche synchrone

Abdallah Adolf, Abdoulaye Gamatié, Jean-Luc Dekeyser

Parc Scientifique de la Haute Borne 40, avenue Halley Bât.A, Park Plaza ,
59650 Villeneuve d'Ascq - France
{Adolf.Abdallah, Abdoulaye.Gamatie, Jean-Luc.Dekeyser}@lifl.fr

---

Résumé

Les systèmes embarqués sur puce (ou *system-on-chip*, SoC) sont de plus en plus sophistiqués en intégrant de multiples fonctionnalités. Ils requièrent beaucoup de ressources pour améliorer les performances d'exécution. Leurs développements posent un véritable défi en raison, à la fois de leurs complexités et de leurs exigences en qualité de service. Cet article s'intéresse à la conception de ces systèmes en élevant le niveau d'abstraction de la spécification par le biais d'une approche dirigée par les modèles. Dans le contexte de l'environnement Gaspard, nous utilisons le profil UML/MARTE dédié à la modélisation et à l'analyse des systèmes embarqués à temps réel. Particulièrement, nous visons des systèmes effectuant du calcul intensif, à l'image des applications de traitement d'images. Ainsi, à partir de modèles définis par un utilisateur, des informations sont extraites pour analyser le système. Notre étude porte plus précisément sur l'analyse temporelle du système. Elle s'appuie sur l'approche synchrone réactive qui favorise fortement la validation formelle. Le résultat de l'analyse peut ensuite servir dans l'exploration d'architecture du SoC modélisé à haut niveau.

Mots-clés : Système embarqué, parallélisme de données, modélisation, approche synchrone, Gaspard

---

1. Introduction

L'évolution rapide des technologies embarquées en général, et en particulier des systèmes-sur-puce (ou *system-on-chip*, SoC), conduit à une explosion du nombre de fonctionnalités intégrées à un système, ainsi qu'à une augmentation du besoin en performances d'exécution pour une qualité de service raisonnable des systèmes. Cela induit naturellement une complexité quant au développement des systèmes embarqués. Un domaine emblématique où cette observation est rapidement vérifiée est celui du multimédia. Les dernières générations de téléphones portables offrent une riche palette d'applications s'exécutant sur un même support qui est une puce : musique, vidéo, internet, téléphonie, photographie. Une caractéristique importante des applications multimédia est le traitement intensif de données.

Pour répondre efficacement au défi de conception posé par les SoC dédiés au traitement intensif de données, plusieurs ingrédients doivent être pris en compte. D'une part, une exploitation intelligente du parallélisme potentiel (aux niveaux tâches et données) permettrait d'avoir des exécutions performantes des systèmes. D'autre part, des moyens de conception à un haut niveau d'abstraction permettraient d'aborder les systèmes tout en s'affranchissant, dans un premier temps, des détails qui font leur complexité. En revanche, ils doivent impérativement être accompagnés de processus de raffinements grâce auxquels une passerelle vers des mises en œuvre effectives pourrait être systématiser.

Notre environnement Gaspard [7] vise principalement à répondre à l'attente ci-dessus en offrant la possibilité de concevoir à haut niveau des systèmes embarqués à hautes performances. Il permet la modélisation d'applications parallèles ayant des traitements intensifs et réguliers de données, ainsi que la génération automatique de code ciblant différentes technologies. Pour cela, Gaspard utilise d'une part le modèle de calcul dit *répétitif* [3] et l'*ingénierie dirigée par les modèles (IDM)* [15]. Le modèle de calcul répétitif est largement inspiré du langage Array-OL [6] pour permettre la description de parallélismes de tâches et de données, de manière compacte. Il est également utilisable pour décrire des topologies et structures régulières au niveau matériel. Quant à l'IDM, il s'agit d'une approche de développement

souple dans laquelle une abstraction de la complexité rendue possible par le biais des modèles et leurs transformations associées. De cette manière, les détails d'implémentation de bas niveau sont introduits progressivement lors de raffinements pour simplifier la modélisation ainsi que l'analyse d'un système. L'IDM utilise en général UML (*Unified Modeling language*) [11] pour la description des modèles.

Dans cette étude, nous utilisons l'environnement Gaspard pour aborder la conception de systèmes embarqués avec du traitement intensif de données, tout en prenant en compte une autre technique permettant d'appréhender à haut niveau nos choix de conception. Cette technique, basée sur l'approche réactive synchrone [1], propose une notion d'horloge périodique N-synchrone [5] qui sert à analyser les décalages pouvant exister entre des composants communiquant entre eux. Ici, nous exploitons ces informations pour aborder le dimensionnement de mémoire ou bien l'ajustage des fréquences de calculs dans le système afin que les contraintes de qualité de service ou de performances deviennent satisfaisantes.

Ce papier est organisé de la manière suivante : la section 2 présente d'abord la modélisation d'un cas d'étude à l'aide de l'environnement Gaspard ; la section 3 analyse le comportement temporel du cas d'étude ; la section 4 discute de l'exploitation des résultats d'analyse obtenus pour satisfaire les contraintes non fonctionnelles imposées sur le système à haut niveau ; enfin, la conclusion et quelques perspectives sont données en section 5.

2. Conception de SoC à haute performance dans l'environnement Gaspard

Afin d'illustrer la conception d'un SoC en utilisant l'environnement Gaspard, nous considérons une application multimédia et une architecture multiprocesseur pour implémenter l'application. Cet environnement, basé sur le modèle dit *Y-Chart*, est dédié à la conception conjointe logiciel/matériel. Gaspard utilise le profil standard MARTE de l'OMG dédié à la modélisation et l'analyse des systèmes embarqués et temps réel [10]. Ce profil enrichit UML avec de nouveaux concepts pour modéliser du logiciel ainsi que du matériel pour les systèmes embarqués.

Nous considérons les fonctionnalités d'un *downscaler* comme étude de cas dans cet article. À la réception d'un flot d'images haute définition de taille 1920×1080 pixels, l'algorithme sert à réduire la taille de ces images à 640×540 pixels afin de les afficher sur l'écran. Cet algorithme de réduction du nombre de pixels est très connu dans le domaine du traitement d'images où l'affichage des vidéos avec des résolutions différentes est souvent nécessaire.

2.1. Modélisation des fonctionnalités du *downscaler*

L'algorithme de réduction d'image consiste en cinq composants principaux [4]. D'abord, l'information spatiale de chaque image est extraite et est représentée par la luminance, la chrominance rouge et la chrominance bleue. Cela permet de coder l'information RVB. Puis, un filtre horizontal et un filtre vertical permettant de réduire la taille de l'image de 1/3 et de 1/2 respectivement, sont appliqués sur chaque image. Cependant, une réorganisation de pixels est effectuée entre les deux filtres pour avoir une distribution plus adéquate des pixels [12]. Ceci aboutira à une image de taille 640×540 pixels en sortie. Le processus de réduction d'une image est illustré par la Figure 1. Une fois réduite, l'image est affichée sur l'écran.

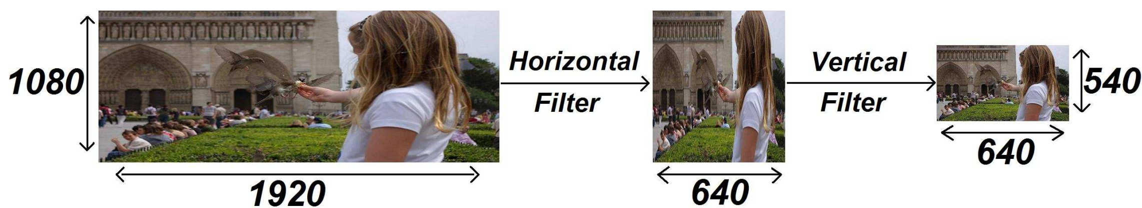

FIG. 1: Réduction de la taille d'une image.

Nous spécifions, en utilisant l'environnement Gaspard, les fonctionnalités du système introduit ci-dessus. La Figure 2 montre deux composants au niveau application (ou fonctionnel), appelé `Downscaler` et `DownscalingProcess`, qui décrivent les fonctionnalités du système. Ici, nous utilisons UML pour

créer le composant englobant, les ports d'entrées et de sorties, les connecteurs ainsi que les instances de composants contenues dans le composant englobant. De plus, nous nous servons du profil MARTE pour ajouter plus de détails relatifs à la spécification. Ces informations sont insérées à l'aide de *stéréotypes* UML. Par exemple, les ports d'entrées sont de type `FlowPort` (un stéréotype de MARTE) ayant la direction `in`. De la même façon, les ports de sorties sont stéréotypés `FlowPort` avec une direction `out`. Pour des raisons de lisibilité, ces stéréotypes sont volontairement omis dans les figures.

Le composant `Downscaler` correspond au plus haut niveau hiérarchique de la description des composants. Il contient trois composants. `FrameGenerator` et `FrameConsumer` sont des composants élémentaires. Le premier est un générateur de pixels sous forme d'images de haute définition et le deuxième reçoit ou affiche les images réduites. Le troisième composant `DownscalingProcess` est un composant hiérarchique qui décrit les différentes étapes permettant la réduction. Celui-ci est également décrit en haut de la Figure 2. Dans `DownscalingProcess`, les trois instances de composants `ReOrdering`, `HorizontalFilter` et `VerticalFilter` permettent de réduire le nombre de pixels alors que les composants `ColorInformation` et `Pixelconstructor` permettent de manipuler les informations RVB des images produites et consommées.

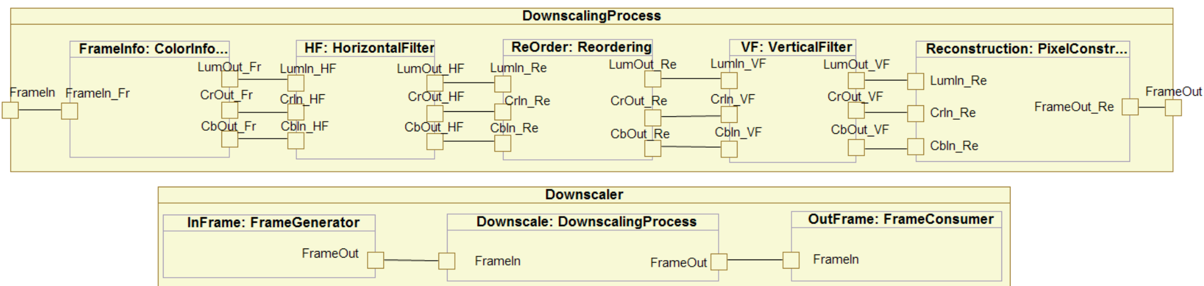

FIG. 2: Modélisation des fonctionnalités du *downscaler*.

Afin d'exprimer le parallélisme de données, on utilise les concepts de MARTE définis dans le package `RSM`. Ce package est basé sur le modèle de calcul répétitif [2]. La figure 3 exprime, à un haut niveau de spécification, le parallélisme de données d'une application manipulant des structures de données multidimensionnelles, afin de permettre une distribution efficace de tâche sur une architecture parallèle. Le stéréotype `Shaped` qui est un vecteur de valeur {4,4} exprime l'espace de répétition du composant `Lum`. Cela signifie que le composant est répété 4×4 fois. Chacune des répétitions consomme des motifs en entrée qui sont extraits des tableaux d'entrées du composant englobant. De même, chacune des répétitions produit des motifs en sortie qui seront consommés par les tableaux en sortie du composant englobant. L'attribut `fitting` de type matrice permet de déterminer les éléments du tableau qui sont liés à chaque motif. La matrice `paving` décrit l'espacement régulier des différents motifs au sein d'un tableau d'entrée ou de sortie. En d'autres termes, elle permet d'identifier l'origine de chaque motif, associée à chaque répétition de composant. Le vecteur `origin` spécifie l'origine des motifs dans un tableau.

Pour décrire la répétition d'un composant, l'espace de répétition doit être spécifié ainsi que les attributs `fitting`, `paving` et `origin` associés au stéréotype `tiler`.

2.2. Modélisation de l'architecture

Afin d'implémenter l'algorithme décrit ci-dessus, nous considérons une architecture matérielle composée de processeurs qui communiquent entre eux à travers une mémoire partagée. L'accès à la mémoire, par l'intermédiaire d'un bus partagé localisé entre les processeurs et la mémoire partagée, permettra le stockage de données en cas de délai dans la communication inter-processeur.

La Figure 4 montre, à un haut niveau d'abstraction, le modèle d'architecture qui va implémenter les fonctionnalités du *downscaler*. Quatre processeurs ont été définis pour traiter les différents composants applicatifs du système. Un émetteur (le composant `Sensor`) et un récepteur (le composant `Actuator`) sont aussi définis pour produire et consommer les données du système. Nous définissons également

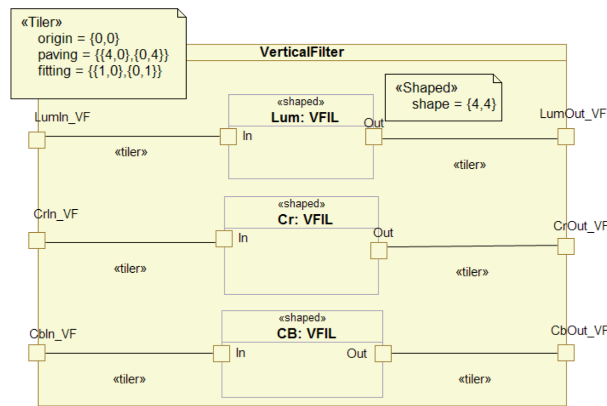

FIG. 3: Filtre vertical.

deux mémoires `SRAM` et `ROM` pour écrire, stocker et récupérer les données et les instructions du programme qui vont être traitées par les processeurs. Toutes les instances de composants, contenues dans le composant `MainArchitecture`, sont stéréotypées avec des concepts du profil MARTE (tous les noms commençant par `hw`). Toutes les communications entre les ressources physiques passent forcément par le composant `Bus` à travers les ports `Master` et `Slave`. Le concept de stéréotype, évoqué précédemment, enrichit le modèle UML initial avec des propriétés concernant les systèmes embarqués temps réel, typiquement la fréquence d'un processeur.

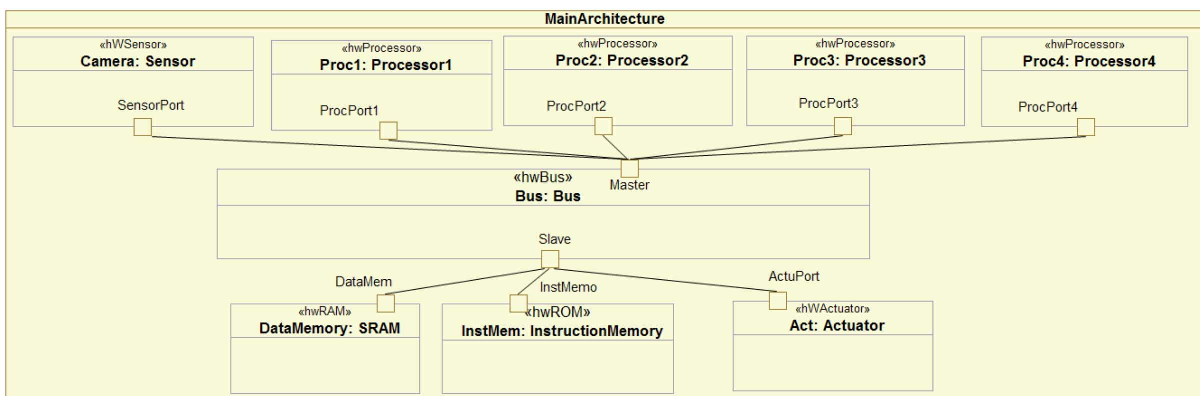

FIG. 4: Modélisation de l'architecture considérée.

2.3. Association et déploiement de composants décrivant la fonctionnalité et l'architecture

Une fois l'application et l'architecture matérielle modélisées, les éléments décrivant les fonctionnalités du système sont alloués sur les ressources physiques au niveau architecture. C'est durant la phase d'association que le concept d'allocation est exprimé.

Dans ce contexte, il faut spécifier comment l'application sera placée sur la plate-forme d'exécution. Cette phase d'allocation est cruciale pour le fonctionnement du système car les performances d'exécution en dépendent. En particulier, les caractéristiques des processeurs qui vont exécuter les fonctionnalités du système ainsi que la taille de mémoire utilisée sont d'une grande importance. Pour cette raison, dans la Section 4, nous explorons différents choix possibles concernant la plate-forme d'exécution afin d'avoir de bonnes performances durant l'exécution. Dans le modèle UML, c'est avec le stéréotype `allocate`

de MARTE que l'utilisateur exprime l'allocation. Dans la Figure 5, le composant `HorizontalFilter`, contenu dans la description fonctionnelle du système, est associé au processeur `Proc1` par le biais d'un connecteur de type `allocate` de MARTE. Aussi, les ports `FrameIn` et `FrameOut` du composant `DownscalingProcess` sont également associés à la mémoire `DataMemory`.

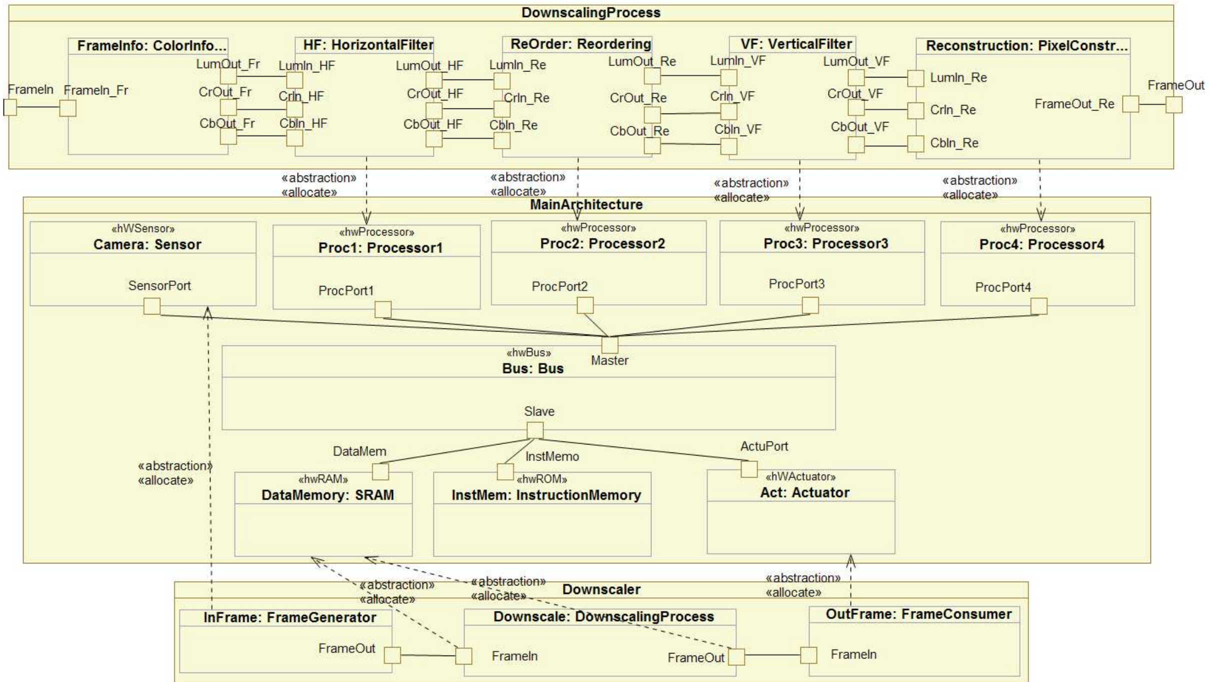

FIG. 5: Modélisation de l'association entre les fonctionnalités du *downscaler* et l'architecture choisie.

Jusqu'à présent, nous nous placions à un niveau de modélisation indépendant de la plate-forme d'exécution. Durant la phase de déploiement dans Gaspard, on introduit la notion de propriété intellectuelle (Intellectual Property, IP) qui permet de cibler différentes implémentations spécifiques. Nous passons ainsi à une modélisation dépendante de la plate-forme d'exécution. Chaque ressource est associée à une implémentation matérielle. Le type et les caractéristiques des ressources physiques sont aussi définis. La Figure 6 montre, dans le cas du *downscaler*, comment le processeur matériel `Proc1` est déployé sur `VProc1`. Le composant `VProc1`, de type `virtualIP`, contient deux implémentations matérielles au niveau *SystemC Transaction Level Modeling* (TLM). Le composant `VProc1-TLM1` (resp. `VProc1-TLM2`), stéréotypé `hardwareIP` et `hardwareProcessor`, a une fréquence de 15 Hz (resp. 30 Hz).
De même, pour implémenter les modèles décrivant le comportement fonctionnel du système, tous les détails d'implémentation sont insérés par une phase de déploiement de l'application. Les différentes implémentations d'un composant, rassemblées dans un `virtualIP`, décrivent le code source de la fonctionnalité.
Après avoir spécifié le système en utilisant UML et le profil MARTE, le modèle résultant est alors raffiné dans Gaspard. Ce raffinement passe par plusieurs étapes de transformation de modèle afin d'arriver à la génération de code pour diverses technologies-cibles : en langages synchrones tels que Lustre ou Signal pour la validation fonctionnelle du modèle, SystemC pour la simulation à différents niveaux d'abstraction, VHDL pour la synthèse de circuit et OpenMP pour aborder du calcul scientifique.

3. Analyse temporelle à l'aide d'horloges logiques

Dans cette section, nous analysons le comportement temporel du modèle décrit dans la section 2. Notre analyse repose principalement sur [5] dans lequel la synchronisabilité de plusieurs composants d'un

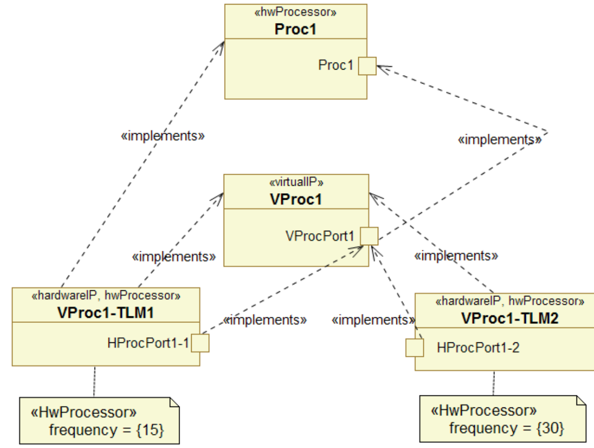

FIG. 6: Déploiement matériel.

système temps réel embarqué est étudiée.
Dans un modèle Gaspard, la spécification du comportement fonctionnel correspond à un graphe $\mathcal{C}$ qui représente l'ensemble des composants décrivant le comportement fonctionnel du système. $\mathtt{Ch}$ est l'ensemble des connecteurs qui lient un port d'entrée à un port de sortie. $\mathtt{P}_{\mathtt{In}}$ (resp. $\mathtt{P}_{\mathtt{Out}}$) est l'ensemble des ports d'entrées (resp. sorties) associés aux composants.
Afin d'étudier la synchronisabilité entre des composants, nous utilisons le concept d'horloges logiques basées sur l'occurrence d'événements à travers lesquels ces composants communiquent. Pour cela, nous définissons $\mathtt{Clk}$ comme étant l'ensemble des horloges logiques $\mathtt{clk}_i$ exprimant les activations des composants. À chaque instant d'une horloge $\mathtt{clk}_1$, le composant C_1 associé à celle-ci est activé et donc exécuté par le processeur correspondant. Les données, représentées sous forme de motifs, sont consommées par les ports d'entrée du composant C_1, traitées, puis produites et transmises à un autre composant C_2 via les ports de sortie de C_1. D'autre part, nous adoptons l'hypothèse synchrone [13] dans notre démarche : la latence des calculs pour chaque composant est négligée.

3.1. Description de comportements temporels

Chaque processeur dispose d'une horloge propre avec une certaine fréquence. À chaque *tick* de cette horloge, le processeur exécute un cycle d'instruction ou un fragment de cycle selon la taille et la complexité des instructions. La mesure du nombre de cycles par instruction sert à connaître le nombre moyen de cycles d'horloge requis pour qu'un processeur exécute une instruction. Dans le cas du modèle conçu en section 2, le nombre d'instructions que le processeur doit exécuter est fortement lié aux instances contenues dans le composant au niveau fonctionnel.
Dans la figure 6, le composant `Proc1` est déployé au niveau TLM avec deux implémentations différentes. Ces dernières sont stéréotypées `hardwareIP` de MARTE et sont encapsulées dans un `virtualIP`. Le premier `hardwareIP` a une fréquence de 15 Hz tandis que le second a une fréquence de 30 Hz. Nous avons choisi le processeur ayant comme fréquence 15 Hz. Ce choix est présenté dans la Figure 6 par la flèche stéréotypée `implements` allant de `VProc1-TLM1` vers `Proc1`.
Le composant `Proc1`, attribué à `HorizontalFilter` dans la description fonctionnelle, dispose de 16 cycles d'instructions à exécuter car le composant `HorizontalFilter` est répété 4×4 fois. La durée d'exécution des instructions par un processeur, exprimée en seconde, est obtenue via le rapport entre le nombre d'instructions et la fréquence du processeur. La durée d'exécution des différents composants dans la spécification du *downscaler* est donnée par :

$$\begin{aligned}
\mathtt{Proc1} &: F_1 = 15\text{ Hz}, \mathrm{InstCycle}_1 = 16 \Rightarrow d_1 \simeq 1.06\text{secs} \\
\mathtt{Proc2} &: F_2 = 45\text{ Hz}, \mathrm{InstCycle}_2 = 30 \Rightarrow d_2 \simeq 0.66\text{secs} \\
\mathtt{Proc3} &: F_3 = 45\text{ Hz}, \mathrm{InstCycle}_3 = 30 \Rightarrow d_3 \simeq 0.66\text{secs}
\end{aligned}$$

$\texttt{Proc4} : F_4 = 40$ Hz, $\texttt{InstCycle}_4 = 60 \Rightarrow d_4 = 1.5$secs

Nous avons défini le nombre de cycles d'instructions que chaque processeur doit exécuter ainsi que la durée d'exécution de chaque processeur. Dans la section suivante, nous propageons cette information sur les horloges binaires associées aux différents composants.

3.2. Horloges logiques

Dans les systèmes distribués, il est souvent impossible d'avoir des horloges parfaitement synchronisées dans une échelle de temps globale. Parmi les solutions existantes, les horloges logiques [8] fournissent un moyen intéressant pour aborder ce problème. Une relation dite *s'est produite avant* est définie pour capturer les dépendances causales entre les événements observés dans le système.

D'autres travaux tels que [5] [16] manipulent également les horloges logiques pour raisonner sur la synchronisabilité entre des composants d'un système. À travers leurs propositions, les auteurs suggèrent de relâcher les contraintes de synchronisation très fortes imposée par l'hypothèse synchrone, toujours dans le but de spécifier les systèmes embarqués avec les outils liés aux langages synchrones.

Dans [16], les horloges affines ont été définies comme une extension de la notion usuelle d'horloge dans le langage Signal. Elles ont permis la modélisation et la validation des systèmes à temps réels ayant du traitement intensif de données et un comportement temporel régulier affine.

De la même manière, les réseaux de Kahn N-synchrone, définis par [5], permettent de spécifier des comportements périodiques des systèmes. Des horloges périodiques sont définies et associées aux différents composants d'un système. Une étude de synchronisation sur les horloges permet alors de vérifier certaines contraintes de synchronisabilité. Les réseaux de Kahn N-synchrone proposent une solution pour la synchronisation au sein d'un système par l'insertion d'une mémoire tampon. Ceci permet de spécifier, avec des outils synchrones, des systèmes non nécessairement synchrones.

Horloges binaires.

Les horloges logiques que nous manipulons dans cette étude, ont une notation binaire. Elles sont associées aux composants décrivant un comportement fonctionnel. Cette notation est inspirée de [5] de même que le raisonnement appliqué.

Étant donné un composant C spécifié dans la description fonctionnelle du système et contenant au moins une instance d'un autre composant, $\texttt{clk}$ dénote l'horloge binaire associée à C. Cette horloge est déduite du modèle UML de la section 2. Afin de synthétiser $\texttt{clk}$, le modèle doit contenir deux informations essentielles décrivant son comportement temporel :

- Le nombre de cycles d'instructions que le processeur doit exécuter. Cette information est déduite du nombre de répétitions du composant contenues dans C.
- La fréquence associée au processeur. Cette information est fournie par le concepteur du système au cours de la phase de déploiement des IPs. Cette information est sous la forme d'un attribut associé au stéréotype `hwProcessor` de MARTE .

Les horloges logiques binaires sont composées d'une séquence de valeurs binaires, 0 et 1. La valeur 0 indique l'état inactif du composant alors que la valeur 1 indique son activation. Ces horloges sont associées à des composants décrivant la fonctionnalité du système. La séquence 010001001111 est un exemple d'une horloge binaire.

Puisque, ces horloges ont été synthétisées à partir d'informations qui concernent la fréquence du processeur et le nombre de cycles exécutés, nous pouvons également relier leurs valeurs binaires à une description de bas niveau comme suit :

- Occurrence d'un 1 : le processeur associé à une horloge $\texttt{clk}$ est activé à cet instant logique et exécute k-cycles du composant qui lui est attribué, où $k \in \mathbb{N}^*$.
- Occurrence d'un 0 : le processeur associé à $\texttt{clk}$ est dans l'un des deux états suivants : activé et exécute 0-cycle ou bien inactif.

D'autre part, nous supposons l'existence d'une *horloge idéale* dans un système. Les différentes horloges du système pourront ainsi être calées sur cette horloge globale. Pour analyser deux ou plusieurs horloges binaires, elle nous servira de référence pour observer leurs activations. Pour cela, nous définissons la fonction *position* comme étant la position de la $n^{\text{ième}}$ activation d'un composant. Nous écrivons alors, $\text{position}(n, \texttt{clk}) = q$ où $q \in \mathbb{N}^*$. En considérant l'exemple précédent d'horloge binaire, c'est-à-dire

$\mathtt{clk} = 010001001111$, nous obtenons $\mathtt{position}(2, \mathtt{clk}) = 6$ et $\mathtt{position}(5, \mathtt{clk}) = 11$ (le parcours de la position se fait à partir de 1 au lieu de 0).
La fonction *taille*($\mathtt{clk}$) retourne le nombre de valeurs binaires dans une horloge donnée $\mathtt{clk}$. Par exemple, la longueur de l'horloge dans l'exemple ci-dessus $\mathtt{clk} = 010001001111$ est 12 qui est le nombre total d'occurrences de 0 ou de 1 dans $\mathtt{clk}$.

Distribution de calculs.

En traitement d'images, les calculs effectués sur les données (pixels ou blocs de pixels) sont souvent périodiques. Pour ce type de calcul, nous distinguons deux manières de répartir les calculs : une fonction qui est soit *linéaire* soit *affine*. En d'autres termes, les fonctions ont pour rôle de distribuer les occurrences de 1 dans une séquence de valeur binaire. Le coefficient de proportionnalité représente la période entre deux activations simultanées. La fonction exprime la position de chacune des occurrences par rapport à l'horloge idéale. La période associée à la distribution est directement liée au nombre de *ticks* dans $\mathtt{clk}_i$ et à l'horloge idéale : $\exists\, n \in \mathbb{N}$ où n est le nombre d'occurrences de 1 dans l'horloge idéale et $\exists\, m \in \mathbb{N}$ où m est le nombre d'occurrences de 1 dans $\mathtt{clk}_i \Rightarrow p = n/m$, où p est la période de $\mathtt{clk}_i$.
La Figure 7 montre la distribution des instants d'activation des horloges binaires correspondants aux quatre processeurs, tous associés à une horloge idéale $\mathtt{IdealClk}$.

$$
\begin{array}{rl}
\mathtt{IdealClk}: & 1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,.. \\
\mathtt{clk}_1: & 0\,0\,0\,1\,0\,0\,0\,1\,0\,0\,0\,1\,0\,0\,0\,1\,0\,0\,0\,1\,0\,0\,0\,1\,0\,.. \\
\mathtt{clk}_2: & 0\,1\,0\,1\,0\,1\,0\,1\,0\,1\,0\,1\,0\,1\,0\,1\,0\,1\,0\,1\,0\,1\,0\,1\,0\,.. \\
\mathtt{clk}_3: & 0\,0\,0\,0\,1\,0\,1\,0\,1\,0\,1\,0\,1\,0\,1\,0\,1\,0\,1\,0\,1\,0\,1\,0\,1\,.. \\
\mathtt{clk}_4: & 1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,..
\end{array}
$$

FIG. 7: Trace des horloges $\mathtt{clk}_1$, $\mathtt{clk}_2$, $\mathtt{clk}_3$, $\mathtt{clk}_4$.

Dans la sous-section suivante, nous insérons l'information temporelle dans le model UML initial en se servant du profile MARTE.

3.3. Insertion des contraintes horloges dans le modèle UML

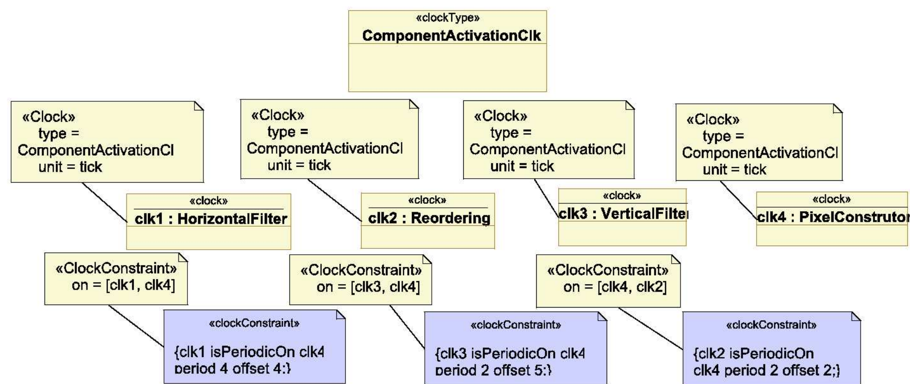

FIG. 8: Réduction de la taille d'une image.

Après avoir analysé le comportement temporel du système, nous introduisons le résultat de l'analyse au modèle UML initial en créant des horloges logiques et en les associant aux composants concernés. Ces horloges décrivent l'exécution des composants alloués à des ressources physiques. La définition des horloges se fait à l'aide du package Time de MARTE alors que la spécification des contraintes d'horloges est réalisée à l'aide du package *Clock Constraint Specification Language* (CCSL) de MARTE [9].

En haut de la Figure 8, le composant `ComponentActivationClk` est créé et stéréotypé *ClockType*. Il définit le type d'une horloge. Ensuite, nous définissons quatre horloges que nous associons aux composants correspondants. La création d'horloges est illustrée au milieu de la Figure 8. Enfin, nous pouvons spécifier les contraintes qui existent entre les différentes horloges. Le bas de la Figure 8 montre trois contraintes de périodicité.

Nous analysons dans la suite ces horloges ainsi que leurs contraintes pour déduire des informations concernant les choix retenus pour l'architecture du système.

3.4. Analyse temporelle

Dans cette section, nous analysons le comportement temporel des composants en vue d'étudier l'architecture spécifiée précédemment. Cela nous permet de modifier certaines propriétés dans la conception faite à haut niveau dans le but d'améliorer les choix de conception.

Soit $\mathtt{clk}_1$ (resp. $\mathtt{clk}_2$) l'horloge associée au composant C_1 (resp. C_2) et *nb1* (resp. *nb2*) le nombre total d'activations de composant dans $\mathtt{clk}_1$ (resp. $\mathtt{clk}_2$), nous supposons que *nb1* = *nb2*. En d'autres termes, les composants C_1 et C_2 sont activés le même nombre de fois. Nous détaillons dans l'algorithme décrit dans la Figure 9 la sémantique de l'analyse de synchronisation sur les horloges $\mathtt{clk}_1$ et $\mathtt{clk}_2$ et comment cette analyse se traduit à un bas niveau de description.

$\forall\, n \in \mathbb{N}$

if *position*(n,$\mathtt{clk}_1$) = *position*(n,$\mathtt{clk}_2$) **then**

Le canal Ch_1 qui relie P_{Out} de C_1 à P_{In} de C_2 n'a pas besoin d'une mémoire tampon ou d'un système de ralentissement, car la communication entre ces deux ports est synchrone.

else if *position*(n,$\mathtt{clk}_1$) < *position*(n,$\mathtt{clk}_2$) **then**

Le nombre d'activations de C_1 est le même que C_2 mais avec un certain retard d. Cela signifie que C_1 produit des données et C_2 consomme la même quantité de données, mais après d instants logiques. Par conséquent, des mesures devraient être prises pour permettre la synchronisation des deux horloges afin d'éviter la perte de données (voir les sections suivantes).

else if *position*(n,$\mathtt{clk}_1$) > *position*(n,$\mathtt{clk}_2$) **then**

C_2 consomme des données avant qu'elles ne soient produites par C_1, ce qui rend le système incohérent car les données ne sont pas présentes lors de l'activation de C_2. Pour cela, des mesures devraient être prises pour ralentir $\mathtt{clk}_1$ ou pour accélérer $\mathtt{clk}_2$ dans le but de satisfaire les exigences du système.

end if

FIG. 9: Algorithme d'analyse de synchronisation des horloges.

Toutefois, si $nb1 \neq nb2$, cela signifie que $\mathtt{clk}_1$ et $\mathtt{clk}_2$ ne sont pas synchronisables car ils n'ont pas le même nombre d'activations de composants. Ci-dessous, nous définissons des fonctions qui vont nous permettre de répondre aux problèmes identifiés lors de l'analyse.

Definition 1 (Quantité) *Soit* $\mathtt{clk}$ *une horloge binaire,* $\forall i \in [1, \mathtt{taille}(\mathtt{clk})]$, *les expressions* $(1.\mathtt{clk}, i)$ *et* $(0.\mathtt{clk}, i)$ *expriment respectivement que la* $i^{\text{ème}}$ *position dans l'horloge* $\mathtt{clk}$ *est* 1 *et* 0. *La fonction Quantité est alors définie récursivement comme suit :*

- $\mathit{Quantité}(1.\mathtt{clk}, i) = 1 + \mathit{Quantité}(\mathtt{clk}, i-1)$
- $\mathit{Quantité}(0.\mathtt{clk}, i) = 0 + \mathit{Quantité}(\mathtt{clk}, i-1)$
- $\mathit{Quantité}(1.\mathtt{clk}, 1) = 1$
- $\mathit{Quantité}(0.\mathtt{clk}, 1) = 0$

En appliquant la fonction *Quantité* sur les horloges $\mathtt{clk}_2$ et $\mathtt{clk}_3$, nous obtenons le nombre de cycles d'instructions exécutés jusqu'à un instant logique donné :

$$\textit{Quantité}(\mathtt{clk}_2,2)=1,\ \textit{Quantité}(\mathtt{clk}_3,2)=0$$
$$\textit{Quantité}(\mathtt{clk}_2,3)=1,\ \textit{Quantité}(\mathtt{clk}_3,3)=0$$
$$\textit{Quantité}(\mathtt{clk}_2,4)=2,\ \textit{Quantité}(\mathtt{clk}_3,4)=0$$
$$\textit{Quantité}(\mathtt{clk}_2,6)=3,\ \textit{Quantité}(\mathtt{clk}_3,6)=1$$
...

Grâce à la fonction *Quantité*, nous sommes en mesure de déduire le délai entre deux horloges distinctes.

Definition 2 (Délai) *Soient* $\mathtt{clk}_x$, $\mathtt{clk}_y \in \mathtt{Clk}$ *et* $\forall\ i \in$ *[1, taille(*$\mathtt{clk}_x$*)]. La fonction Différence() est définie comme suit : Différence(*$\mathtt{clk}_x$*,*$\mathtt{clk}_y$*,i) = |Quantité(*$\mathtt{clk}_x$*,i) - Quantité(*$\mathtt{clk}_y$*,i)|.*

En appliquant la fonction *Différence* à des horloges, nous pouvons déterminer le décalage entre $\mathtt{clk}_2$ et $\mathtt{clk}_3$, à chaque instant logique :

$$\textit{Diff}(\mathtt{clk}_2,\mathtt{clk}_3,2)=1$$
$$\textit{Diff}(\mathtt{clk}_2,\mathtt{clk}_3,3)=1$$
$$\textit{Diff}(\mathtt{clk}_2,\mathtt{clk}_3,4)=2$$
$$\textit{Diff}(\mathtt{clk}_2,\mathtt{clk}_3,6)=2$$
...

3.5. Analyse de synchronisation

Nous considérons les notions introduites dans la section 3 pour analyser des problèmes de synchronisation entre des composants exécutés sur différents processeurs.
La figure 10 est une simulation des résultats obtenus au cours de la synthèse des horloges binaires dans la section 3 à l'aide de l'outil TimeSquare [14] dédié à la modélisation et à l'analyse temporelle de systèmes temporisés. L'horloge $\mathtt{clk}_2$ est créée en appliquant une transformation linéaire sur l'horloge idéale définie par : $f : x \longrightarrow 2x$, où 2 représente la période de $\mathtt{clk}_2$. Quant à $\mathtt{clk}_3$, elle est créée en appliquant une transformation affine sur l'horloge idéale définie par : $f : x \longrightarrow 2x + 3$, où 2 représente la période de $\mathtt{clk}_3$ et 3 représente la phase pour le début des activations.

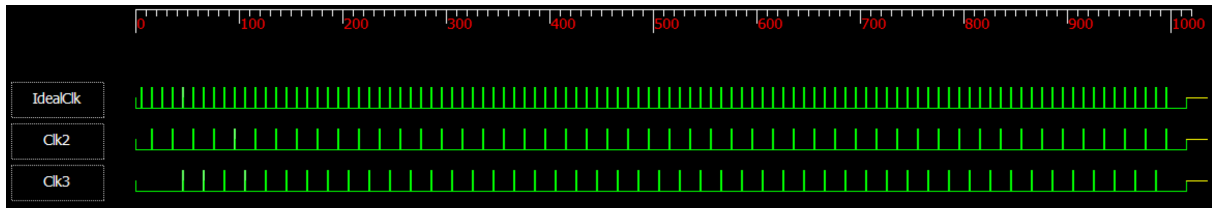

FIG. 10: Simulation du comportement temporel avec l'outil TimeSquare.

Nous pouvons remarquer un problème de synchronisation entre les horloges $\mathtt{clk}_2$ et $\mathtt{clk}_3$ vu que les instants logiques des deux horloges ne sont pas superposables. Le composant `Proc2` (associé à $\mathtt{clk}_2$) exécute le même nombre de cycles d'instructions que `Proc3` (associé à $\mathtt{clk}_3$) mais avec un délai supplémentaire d en termes d'instants logiques. La section suivante discute des mesures possibles qui peuvent être prises afin que les communications soient réalisées de façon sûre.

4. Vers l'exploration d'architecture

Supposons qu'un composant C_1 écrive des données consommées par un autre composant C_2. Si la latence de communication entre les deux composants est nulle (ceci est un cas idéal), les données sont produites et consommées simultanément. De plus, les instants d'activations de C_1 pourraient être calés sur les instants d'activations de C_2. Cependant, cela est rarement le cas et un délai de plusieurs instants logiques peut avoir lieu entre des horloges *mal ajustées*. Pour synchroniser de telles communications, le concepteur d'un modèle Gaspard peut procéder de plusieurs manières, déclinées ci-après.

4.1. Modification de la fréquence

Nous souhaitons par cette approche minimiser la taille des données qui sont générées par C_1 et qui ne sont pas encore consommées par C_2. Pour cela, nous pouvons envisager de "resserrer" au niveau fréquence la synchronisation entre les composants C_1 et C_2. En d'autres termes, nous allons minimiser le décalage entre les instants d'activations de l'horloge de C_1 et ceux de l'horloge de C_2.

Le concepteur devrait alors opter pour une modification du choix des implémentations matérielles. Cela est faisable en changeant le déploiement avec un IP ayant des caractéristiques mieux adaptées. Pour une implémentation donnée, un `virtualIP` peut contenir plusieurs IP ayant différentes fréquences. Donc le concepteur peut sélectionner un autre IP avec une fréquence plus petite au composant C_1. Il peut aussi choisir un IP avec une fréquence plus élevée pour le composant C_2.

Cette modification de la fréquence par le choix des IP va engendrer une modification de la fonction de distribution appliquée aux horloges correspondantes. Cela va induire une minimisation du décalage entre les instants d'activation des deux horloges, et donc de la mémoire tampon nécessaire pour synchroniser le calcul. Pour l'instant, le changement d'IP est effectué de façon manuelle dans notre approche. Cependant, cette tâche pourrait être rendue systématique en vue de son automatisation.

Il faut noter que la modification de la fréquence a un impact sur les contraintes de qualité de service. Par exemple, la diminution de la fréquence d'un processeur peut altérer la qualité de l'affichage des images produites. Enfin, l'augmentation de la fréquence peut être également gênante pour le système du point de vue physique, car elle est généralement accompagnée d'une augmentation de la température du matériel.

4.2. Allocation de mémoire

Le concepteur peut envisager un mécanisme de communication entre composants, contenant une mémoire tampon partagée, permettant de mémoriser temporairement les données échangées entre les composants. L'insertion de cette mémoire tampon doit être limitée à l'endroit où cela est nécessaire pour économiser le coût de conception de la puce ainsi que sa taille. Notre approche permet d'identifier les endroits qui nécessitent une insertion de mémoire, mais cela n'est pas suffisant. En effet, il faut pouvoir estimer une taille minimale de mémoires tampons à utiliser pour un résultat optimal.

Dans la figure 10, nous pouvons remarquer un problème de synchronisation entre les deux horloges $\mathtt{clk}_2$ et $\mathtt{clk}_3$. Cela veut dire, du point de vue architecture, que `Proc2` (associé à $\mathtt{clk}_2$) exécute le même nombre de cycles d'instructions que `Proc3` (associé à $\mathtt{clk}_3$) mais avec un certain retard d. Par conséquent, la communication n'est pas instantanée. Pour stocker les données produites par `Proc2` et qui ne sont pas encore consommées par `Proc3`, nous devrons déterminer la taille minimale de la mémoire tampon qui permet le stockage de données sans perte d'information. Pour cela nous définissons la fonction *TailleMinimale* suivante.

Definition 3 (Taille minimale) $\forall\ \mathtt{clk}_x,\ \mathtt{clk}_y \in \mathtt{Clk}$ *et* $\forall\ i \in$ *[1,taille(*$\mathtt{clk}_x$*)], TailleMinimale(*$\mathtt{clk}_x$,$\mathtt{clk}_y$*) = Maximum(Différence(*$\mathtt{clk}_x$,$\mathtt{clk}_y$*,i))*

À partir des résultats obtenus dans la section 3, et en appliquant la fonction *TailleMinimale*, nous pouvons ainsi conclure que la communication entre $\mathtt{clk}_2$ (associé à `Proc2`) et $\mathtt{clk}_3$ (associé à `Proc3`) a besoin d'un délai de communication qui peut être représenté par une mémoire tampon de taille minimale égale à 2. L'unité de la mémoire tampon est strictement liée à la nature et à la taille des données traitées.

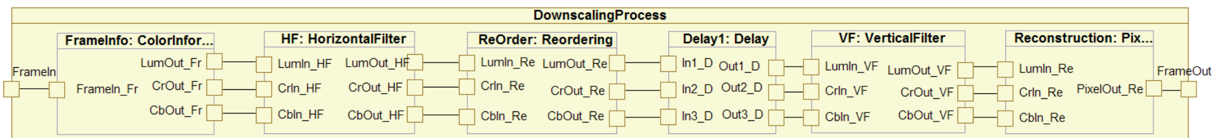

FIG. 11: Ajout d'un composant `Delay1` entre les composants `ReOrder` et `VF`.

Après avoir défini toutes les communications entre composants qui requièrent une resynchronisation,

nous modifions le modèle UML initial en ajoutant des composants `Delay` entre chaque couple de composants nécessitant un délai supplémentaire de communication. La Figure 11 montre l'insertion du composant `Delay1` introduit entre les composants `ReOrder` et `VF`. Ce composant `Delay1` sera alloué sur une mémoire durant la phase d'association.

5. Conclusions et perspectives

Nous avons présenté dans cet article une modélisation à haut niveau d'un système multimédia avec du calcul intensif, en utilisant UML et le profil MARTE. Nous avons déduit une représentation formelle du système global résultant et nous avons analysé le comportement temporel de ses composants. Nous avons essentiellement abordé le problème de la synchronisabilité entre composants. De plus, nous avons proposé une démarche de raisonnement pour résoudre ce problème à haut niveau et d'un point de vue pragmatique : ajustage des fréquences de processeurs et dimensionnement de mémoires tampons servant à la communication entre composants exécutés sur différents processeurs.

Cette étude a été réalisée à un niveau élevé d'abstraction, d'où la non prise en compte de certaines informations (latence de calcul des composants) qui pourraient contribuer à rendre les résultats d'analyse plus précis. Une perspective importante de ce travail consiste donc à étudier une amélioration de notre approche dans cette direction.

Bibliographie

1. Benveniste (A.), Caspi (P.), Edwards (S.), Halbwachs (N.), Le Guernic (P.) et de Simone (R.). – The synchronous languages twelve years later. *Proceedings of the IEEE*, vol. 91, n1, January 2003, pp. 64–83.
2. Boulet (P.). – *Array-OL Revisited, Multidimensional Intensive Signal Processing Specification*. – Research Report n RR-6113, INRIA, février 2007. `http://hal.inria.fr/inria-00128840/en`.
3. Boulet (P.). – *Formal Semantics of Array-OL, a Domain Specific Language for Intensive Multidimensional Signal Processing*. – Rapport technique, INRIA, France, March 2008. available online at `http://hal.inria.fr/inria-00261178/fr`.
4. Chamski (Z.), Duranton (M.), Cohen (A.), Eisenbeis (C.), Feautrier (P.) et Genius (D.). – Application-domain-driven system design for pervasive video processing, 2003.
5. Cohen (A.), Duranton (M.), Eisenbeis (C.), Pagetti (C.), Plateau (F.) et Pouzet (M.). – N-sychronous Kahn networks. *In : ACM Symp. on Principles of Programming Languages (PoPL'06)*. – Charleston, South Carolina, USA, January 2006.
6. Demeure (A.) et Del Gallo (Y.). – An array approach for signal processing design. *In : Sophia-Antipolis conference on Micro-Electronics (SAME'98), System-on-Chip Session, France*. – October 1998.
7. Gamatié (A.), Beux (S. Le), Piel (É.), Etien (A.) et Ben-Atitallah (R.). – *A Model Driven Design Framework for High Performance Embedded Systems*. – Research Report n 6614, INRIA, 2008. `http://hal.inria.fr/inria-00311115/en`.
8. Lamport (L.). – Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, vol. 21, n7, July 1978, pp. 558–565.
9. Mallet (F.). – Clock constraint specification language : specifying clock constraints with uml/marte. *Innovations in Systems and Software Engineering*, vol. 4, n3, October 2008, pp. 309–314.
10. Object Management Group. – A uml profile for marte, 2007. `http://www.omgmarte.org`.
11. Object Management Group. – Unified modeling language, version 2.2 2009. `http://www.omgmarte.org`.
12. Paiva, A. R. C. Pinho (A. J.). – Evaluation of some reordering techniques for image vq index compression. *In : Proceedings of the International Conference on Image Analysis and Recognition (ICIAR'04), Porto, Portugal*, pp. 302–309. – 2004.
13. Potop-Butucaru (D.), Simone (R. De) et Talpin (J.-P.). – The syncrhonous hypothesis and synchronous languages. *In : Embedded Systems Handbook,R. Zurawski*. – CRC Press and ed., 2005.
14. Project-team AOSTE. – Models and methods for the analysis and optimization of systems with real-time and embedding constraints. `http://www-sop.inria.fr/aoste/?r=9&s=30&l=en`.
15. Schmidt (Douglas C.). – Guest editor's introduction : Model-driven engineering. *Computer*, vol. 39, n2, 2006, pp. 25–31.
16. Smarandache (I.M.), Gautier (T.) et Le Guernic (P.). – Validation of mixed signal-alpha real-time systems through affine calculus on clock synchronisation constraints. *In : World Congress on Formal Methods (2)*, pp. 1364–1383. – 1999.