Définition de portabilité en termes de modèle d'exécution pour la simulation des systèmes sur puces

Giani Velasquez[1],
Giovanni Funchal[2,3], and Matthieu Moy[2]

[1] UJF M1-INFO Stage TER
[2] Verimag, 2, avenue de Vignate 38610 Gières, France
[3] STMicroelectronics, 12 rue Jules Horowitz, 38019 Grenoble

Résumé Le travail effectué dans le présent document est une synthèse du travail réalisé au laboratoire verimag pendant le module TER.

Ce travail a été effectué sur un prototype de simulateur transactionnel pour les systèmes sur puces, appelé Jtlm.
Il s'agit d'intégrer certaines fonctionnalités pour pouvoir expérimenter différents modèles d'exécution, et mieux comprendre leur impact sur la manière d'écrire le logiciel embarqué.

1 Introduction

1.1 Système sur puce : SoC

Aujourd'hui, les systèmes embarqués sont partout dans la vie quotidienne : téléphone portable, appareil photo, systèmes électroniques d'une voiture, etc. Ces appareils doivent satisfaire des contraintes très fortes telles que :

– Performance du traitement informatique intensive.
– Consommation très basse d'energie.
– Prix raisonable.

Cette énorme puissance de traitement doit être intégrée dans une très petite surface.[4]

Les microprocesseurs des ordinateurs traditionnels ne sont pas adaptés pour ces applications, car ils consomment beaucoup pour le traitement requis et ils sont relativement chers.

Actuellement, le concept d'un système sur puce est de regrouper tous les composants électroniques nécessaires (partie matérielle) pour le fonctionnement du système (microprocesseur, composants spécifiques...) dans une seule puce. Mais un système sur puce (SoC) comprend aussi une partie logicielle, c.a.d, du logiciel embarqué qui s'exécute dans son microprocesseur.

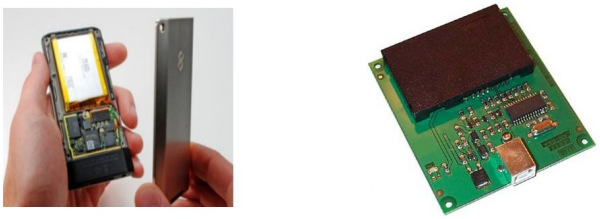

Fig. 1. Un SoC

Compte tenu de la complexité de ces systèmes et des contraintes de temps de mise sur le marché, il n'est plus possible d'attendre les premiers prototypes physiques de puces électroniques pour valider les choix d'architecture et commencer à développer le logiciel embarqué. Donc il apparaît le besoin de travailler sur des modèles. Le processus de conception doit prendre en compte l'interaction entre ces éléments hétérogènes. Mais et à quel moment ? et quels modèles ?

1.2 Flot de conception

Il est nécessaire d'avoir plusieurs modèles du système pendant le flot de conception, pour différentes utilisations, et avec différents niveaux des détails. Voici les deux types des niveaux que l'on peut distinguer :

- RTL : Très détaillé, orienté vers le développement du matériel.
- TLM : Plus abstrait et donc orienté vers le développement du logiciel.

Register Transfer Level. Les modèles RTL (modèles avec niveau d'abstraction RTL) sont un point d'entrée commun pour la production du matériel actuel. Ces modèles sont developpés en utilisant des langages de description du matériel comme VHDL ou Verilog. Ils sont destinés à être transformés automatiquement dans des modèles plus détaillés, qui finalement conduisent à la production du matériel de la puce. D'un autre coté pour développer le logiciel il est nécessaire de l'exécuter. Compte tenu de la contrainte du temps de mise sur le marché (time to market), ce n'est pas possible que les developeurs attendent le premier prototype de la puce électronique pour valider les choix d'architecture et développer le logiciel embarqué. Les modèles au niveau d'abstraction RTL pourraient être utilisés, comme un support pour exécuter le logiciel embarqué, mais, comme les modèles RTL sont très détaillés, ils ont une vitesse de simulation très lente, et donc, le temps de simulation ne conviendra pas au temps de mise sur le marché, alors il est impératif d'avoir des modèles plus abstraits que l'implantation de la puce, le plus tôt possible dans le flot de conception du logiciel. [7].

Transaction level modeling. TLM est relativement un nouvelle approche pour les systèmes sur puce, il a été initialement développé pour

accélérer la simulation de l'exécution du logiciel embarqué. Il a été rapidement adoptée par l'industrie dans le domaine des systèmes sur puces puisqu'il permet d'avoir une simulation plus rapide et nécessite moins d'effort de modélisation car la micro-architecture n'est pas détaillée [6].

Les modèles TLM représentent l'architecture par un ensemble des composants, qui sont connectés par des canaux de communication. Chaque composant contient, un ou plusieurs processus concurrents, ports, ainsi que d'autres composants. Un composant peut être :

– Initiateur (Master), celui qui envoie les transactions.
– Récepteur (Slave), celui qui reçoit les transactions.

Les ports représentent les entrées et sorties des composants. Les composants communiquent entre eux directement ou par un canal de communication.

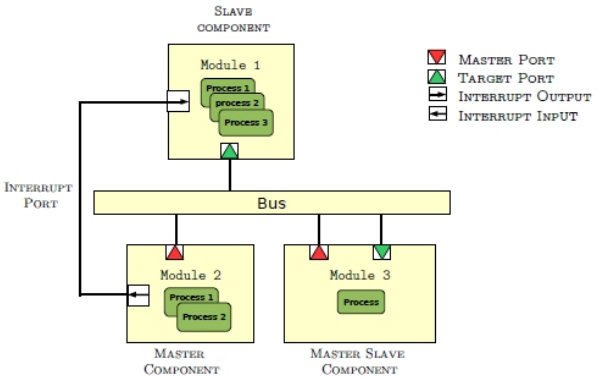

Fig. 2. Exemple d'un plateforme TLM

La plupart de temps, les modèles TLM sont representés en utilisant SystemC, présenté dans la section 1.3. Bien que le standard de fait dans l'industrie soit SystemC (basé sur C++),Verimag a developpé un simulateur alternatif en Java, appelé Jtlm, présenté dans la section 1.4.

La figure 3 compare le temps de simulation pour coder et decoder une image utilisant le codec MPEG4 d'un modèle en niveau d'abstraction RTL et TLM.

1.3 SystemC

SystemC est un langage construit au dessus du C++ standard, qui l'étend avec l'utilisation des bibliothèques, qui introduisent les concepts nécessaires à la modélisation du matériel[9].

SystemC fournit une description parallèle et une implémentation séquentielle, ce qui rend plus facile la synchronisation entre les comportements, mais, en même temps une grande granularité de transactions qui cache un grand nombre de bugs que les programmeurs ne peuvent pas voir.

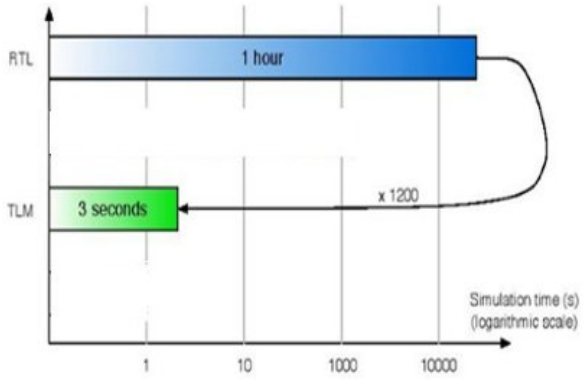

Fig. 3. RTL vs TLM

Le simulateur de SystemC possède un modèle d'exécution coopératif, c.a.d que tous les composants de la simulation doivent partager le temps du processeur, celui-ci va organiser ses processus de la façon suivante : Supposons qu'il y a trois processus : A, B, et C. Au début tous les processus sont *Eligibles*. Un seul processus est selectionné par l'ordonnanceur. Admettons qu'il choisit A. A entre alors dans la phase d'exécution (A effectue ses tâches), et les autres processus (B et C) sont en attente sur du temps, ou d'un événement de SystemC. Un processus se suspend, lorsqu'il exécute une instruction wait pour donner la main à un autre processus et qu'il puisse exécuter ses tâches, c.a.d, le processus A va se suspendre quand il va attendre sur du temps ou va se mettre en attente d'un événement. Ceci a pour conséquence que B, par exemple, puisse s'exécuter.

1.4 Introduction à Jtlm

Verimag a developpé un simulateur alternatif en Java, appelé Jtlm (qui existait déjà à mon arrivé au laboratoire)[1], pour expérimenter différents modèles d'exécution, et mieux comprendre leurs impacts sur la manière d'écrire du logiciel embarqué.

Jtlm utilise les threads de java [5] qui permettent une description parallèle et une exécution parallèle, ce qui permet d'avoir un modèle d'exécution préemptif (ce qui n'est pas le cas de SystemC).

Ce simulateur a été conçu pour exhiber le fait que TLM n'est pas forcement coopéeratif, et voir le vrai comportement de TLM sans le langage SystemC, et en même temps pour identifier ce qui appartient vraiment à TLM et ce qui appartient à SystemC.

Avantages de Jtlm. A la base SystemC avait été créé pour écrire des modèles RTL. Après l'adoption du niveau de modèle transactionnel

dans le flot de conception pour les SoCs, SystemC a été choisi comme un langage standard pour écrire des modèles TLM, et quelques primitives de SystemC ont été importées pour implémenter la bibliotèque de SystemC/TLM. Le fait de copier/coller a emmené à avoir des différences entre ce qui répresente un modèle SystemC/TLM et un vrai modèle TLM. Alors que Jtlm a été conçu directement pour représenter des modèles TLM.
De plus, Jtlm modèlise les tâches en tenant compte de la durée, pour être plus fidèle à la réalité, puisqu'en SystemC, quand il y a une transaction en cours d'exécution, le temps reste constant, et celui-ci s'écoule quand il n'y a plus de processus Eligibles.

Inconvénients de Jtlm. Comme plusieurs composants peuvent intéragir en même temps, il est nécessaire de synchroniser les comportements des composants pour éviter les erreurs d'inconsistence de mémoire et l'interference de threads. Ceci est un grand inconvenient pour le programmeur parce qu'il peut y avoir des erreurs qui se produisent par intermittence pour cause de non-determinisme de l'exécution.

1.5 Mon sujet de stage

Mon travail dans ce stage porte sur deux parties :

1. Implémenter la version coopérative du simulateur Jtlm : Il s'agit de l'implémentation d'un algorithme ; où il y a un seul composant en cours d'exécution, et celui-ci doit décider quand il doit s'arrêter et donner la main à un autre composant pour que le processeur puisse être partagé par tous composants du simulateur.
2. Cette partie consiste à isoler et définir la notion de portabilité du simulateur en termes de modèle d'exécution. A cette étape, le simulateur a un modèle d'exécution aussi bien préemptif que coopératif. Donc il s'agit d'établir certaines règles pour avoir la même API (Côté utilisateur) pour ces deux modèles d'exécution.

2 L'état de l'art

2.1 Jtlm

Au début de mon stage, il y avait un prototype de simulateur écrit en java, qui à la base possédait un modèle d'exécution préemptif. Ce simulateur avait été conçu par Nabila Abdessaied[1].
Un système en Jtlm comprend, un ou plusieurs composants qui sont liéés par un canal de communication (Bus) ou par les ports d'interruption. Chaque composant contient des comportements, ports, ports d'interruption, méthodes, et information interne.
Les composants sont les blocs de base dans Jtlm. Ils sont : la mémoire, le bus, l'ITC, LCD, DMA, le Timer.

Une brève description de LCD et ITC sera donnée parce que cela va être nécessaire pour comprendre la suite du rapport.

LCD : Ce composant simule une application qui lit ou affiche l'information stockée dans la mémoire.

ITC (Contrôleur d'interruption) : Ce composant est utilisé pour envoyer des interruptions aux périphériques. Quand un périphérique a besoin d'envoyer une interruption et qu'il n'a pas une sortie d'interruption, ce composant utilise le ITC pour envoyer une interruption à un composant.

La figure 4 montre un exemple d'une plateforme en Jtlm, où il y a deux processeurs, un lcd, un bus et une mémoire.

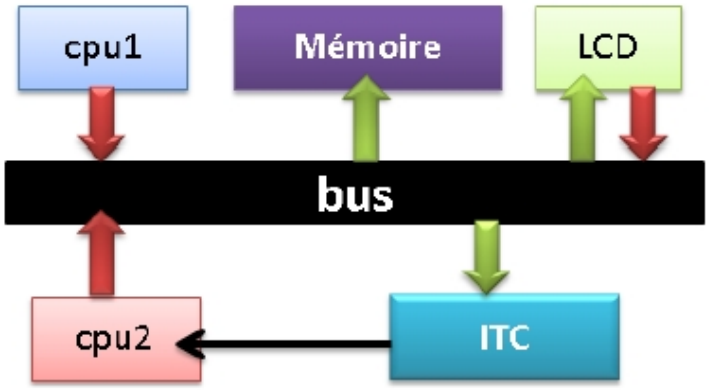

Fig. 4. Platteforme Jtlm

Ce simulateur utilise les threads, puisque chaque comportement du composant de la simulation est répresenté par un thread. Ceci permet à Jtlm d'augmenter les performances de la simulation. Cela permet aussi d'avoir une simulation plus réelle, car tous les composants fonctionnent en parallèle, et de voir les bugs du parallèlisme du vrai système.

Le temps. Pou modèliser le temps que la vraie puce peut prendre pour effectuer son travail, Jtlm utilise dans le modèle d'exécution une variable pour répresenter ce temps, c'est le temps de simulation. Alors que le temps que la simulation de la plateforme virtuelle prend est appelé "wall clock time".

D'après [2], Jtlm gère un ensemble de comportements, et il faut les exécuter en parallèle ou tour à tour. Jtlm repose sur l'ordonnanceur Java pour exécuter les threads qui sont prêts à s'exécuter, mais dispose aussi d'un autre gestionnaire qui est l'ordonnanceur JTLM qui a pour rôle de bloquer les threads qui ne doivent pas (ou pas encore) s'exécuter.

L'ordonnanceur Jtlm gère le temps en maintenant à jour une liste de comportements.
Cette liste peut contenir trois types de comportements : les comportements en cours d'exécution, les comportements qui attendent un temps donné, et les comportements qui ont fini leur travail et attendent d'être supprimés de la liste (Ces comportements sont appelées : comportements zombies). Le temps de la simulation est modifié au fur et à mesure que cette liste est parcourue.

Les tâches. Un comportement en Jtlm peut effectuer plusieurs tâches, ces tâches peuvent se diviser en deux types :
- tâches instantanées : Ces tâches ne prennent pas de temps. exemple : Un comportement effectue une tache f() ;
- tâches avec durée : Ces tâches vont prendre un certains temps à s'effectuer, le temps est passé en paramètre.
  exemple : Un comportement effectue une tâche f'() avec un temps de 10 unités.

```
1 consume(){
2         f'();
3 }.during(10);
```

- tâches avec durée indéterminé [2] : Ces tâches vont prendre un certains temps à s'effectuer, mais leur durée n'est pas connue.

Les primitives de l'utilisateur : Wait . L'utilisateur utilise ces primitives pour bloquer l'exécution d'un comportement :
- waitTime() : Bloque le comportement et le met en attente sur du temps.
- waitEvent() : Bloque le comportement et le met en attentte d'un événement.
- waitInterruption() : Bloque le comportement et le met en attente d'une interruption.

Les primitives de l'ordonnanceur : Wake. L'ordonnanceur utilise ces primitives pour débloquer un comportement.
- wakeUpFromWaitingTime() : Débloque le comportement qui est en attente sur du temps.
- wakeUpFromWaitingEvent() : Débloque le comportement qui est en attentte de cet événement.
- sendInterruption() : Débloque le comportement et qui est en attente d'une interruption.

A cet étape, Jtlm a un ordonnanceur qui gère les comportements d'une façon préemptive, alors il y a eu le besoin d'implémenter un ordonnanceur qui gère les comportements de façon coopéerative 3.1.
Pourquoi le besoin d'une version de Jtlm coopérative ? Une version de

Jtlm coopérative va permettre de comparer ces deux modèles d'exécution (Préemptif et Coopératif) et voir dans quelles situations il vaut mieux utiliser l'un ou l'autre.

Coopératif vs Préemptif. Pour nous, un modèle d'exécution décrit notamment le type de parallélisme utilisé lors de la simulation.

- Coopératif : Un seul processus s'exécute à la fois. Et donc, les changements de contexte doivent être initiés par le programme lui-même.
- Préemptif : Plusieurs processus s'exécutent en même temps. Et donc, les changements de contexte sont transparents pour le programmeur.

D'après l'exemple retiré de [3] : "Considérons un objet qui se déplace sur un cercle et projetons le sur l'axe des x et sur l'axe des y. On obtient un déplacement en cosinus sur les x et en sinus sur les y. Maintenant, considérons la démarche inverse : il y a deux programmes, l'un qui provoque le déplacement de l'objet suivant l'axe des x et l'autre suivant l'axe des y".

Si ces deux programmes sont exécutés ensemble, quel sera le résultat attendu de l'exécution ?.

Tout dépend du type de modèle d'exécution.

Dans le modèle d'exécution coopératif, le résultat va toujours être un cercle, parce que l'objet va bouger une fois sur l'axe x, et la fois d'après sur l'axe y.

Dans le modèle d'exécution préemptif, ce n'est pas possible d'être sûr du résultat de l'exécution puisque les deux threads peuvent s'exécuter en même temps, et ils peuvent écrire chacun de leur côté et avoir comme résultat une figure bizarre.

La figure 5 montre une exécution possible des ces deux modèles.

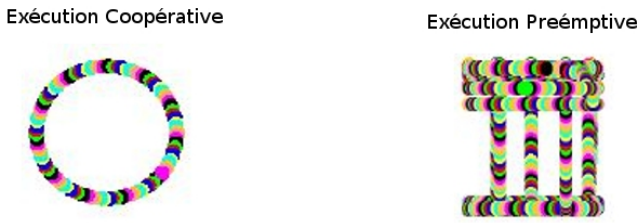

Fig. 5. Exécution coopérative et préemptive

La figure 6, modèlise les deux types de tâches existantes en Jtlm (les tâches instantanées, et les tâches avec durée), et l'attente sur du temps dans les deux modèles d'exécution.

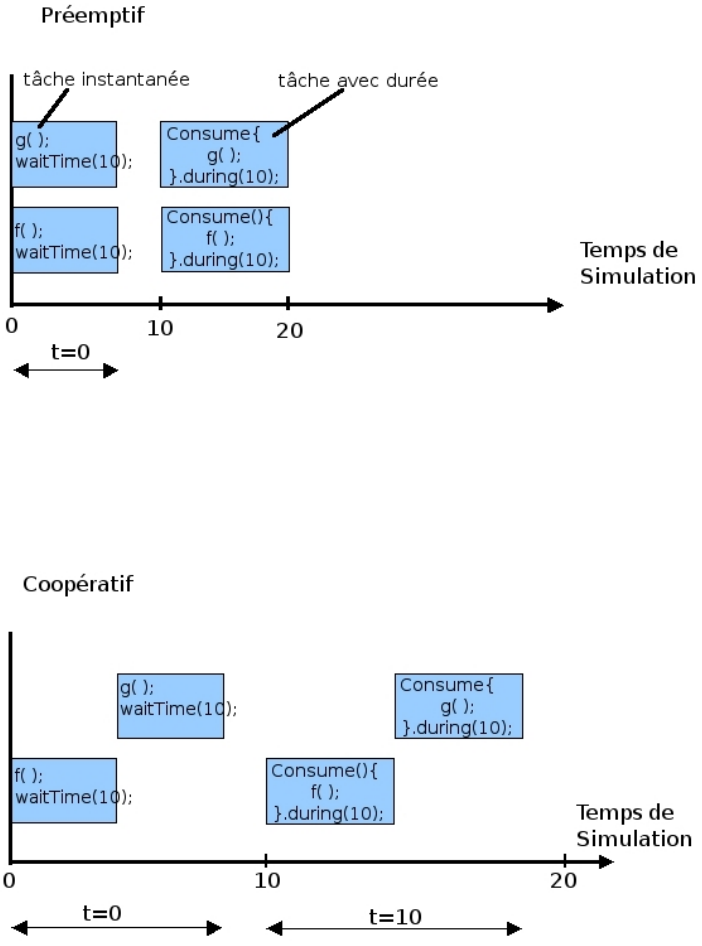

Fig. 6. Evolution du temps dans une exécution coopérative et préemptive en Jtlm

Nous pouvons remarquer que dans la figure 6 le modèle d'exécution préemtpive, f() et g(), s'exécutent au même instant de simulation et en même "wall clock time", alors que dans le modèle d'exécution coopératif, f() et g() s'exécutent au même instant de simulation, mais, f() s'exécute avant g() en "wall clock time".

3 Contribution

3.1 Implémentation de l'Ordonnanceur coopératif

Cette première partie de mon stage consiste à l'implémentation en Jtlm d'un ordonnanceur qui consiste en un algorithme qui gère le partage du temps du processeur dans un modèle d'exécution coopératif entre les différents comportements des composants de la simulation (Un composant peut avoir un ou plusieurs comportements).
Pour faire ceci, nous avons choisi de créer une liste appelée *Eligible*, qui va contenir les comportements des composants qui sont prêts à s'exécuter (vu qu'un seul comportement peut s'exécuter à la fois), mais qui doivent attendre que le comportement courant leur donne la main.

Début de la simulation. Supposons qu'il y a trois composants, appelons les P1, P2, LCD et chacun est associé à un comportement B1, B2, B3 respectivement.

Comme dans tous les programmes, il y a un thread Main qui est le programme principal, celui-ci se charge de lancer la simulation (Simulation.start()).
Pseudo-code de l'initialisation des comportements.

```
1 start (){
2         B1.start ()
3         B2.start ()
4         B3.start ()
5 }
```

Au tout début de la simulation la liste Eligible va contenir tous les comportements existants, c.a.d, au début tous les comportements sont prêt à s'exécuter.
Liste Eligible : B1, B2, B3.

Puisqu'il s'agit d'un modèle d'exécution coopératif, alors un seul comportement peut s'exécuter à la fois, nous avons opté pour arrêter tous les comportements dès qu'ils sont lancés (behavior.start()), pour que l'ordonnanceur ait le contrôle sur chacun d'eux, et qu'il puisse choisir quel comportement sera le premier à s'exécuter.
Ceci se fait en apellant la fonction waiting() au début de l'exécution de chaque comportement la, cette fonction va bloquer l'exécution du comportement courant.

Simulation. Une fois que tous les comportements ont été bloqués, l'ordonnanceur va choisir un comportement de la liste Eligible. Ce sera le premier élément (un comportement) de cette liste qui va être pris pour que ce soit le premier comportement à s'exécuter. Ce choix est tout à fait arbitraire, car à ce moment là il n'y a pas de raison pour exécuter un comportement plutôt qu'un autre.

Le principe d'un modèle d'exécution coopératif est qu'un seul comportement s'exécute à la fois, c.a.d, un comportement effectue une tâche quelconque et ensuite il décide de donner la main à un autre comportement.

Un composant donne la main à un autre en faisaint appel à une des primitives wait (décrites ci-dessous).

Les Primitives côté utilisateur : Wait.

- waitTime (temps) : Le comportement va se bloquer et se mettre en attente sur du temps.
- consume() : Le comportement va faire d'abord (à l'instant où il appelle cette primitve) une ou plusieurs tâches instantanées. Ensuite, il va se bloquer et se mettre en attente sur du temps.
  Les comportements qui sont en attente sur du temps ou qui sont en train de effectuer une tâche avec durée, vont être ajoutés dans une liste qui est appelée "waitingQueue", pour faciliter le langage, cette liste va s'appeller *liste d'Attente*.

- waitEvent() : Le comportement courant se bloque et se met en attente d'un événement.
- waitInterruption() : Le comportement courant se bloque et se met en attente d'une interruption.

Implémentation de ces primitives :

- waitTime() : L'ordonnanceur va ajouter le comportement en cours d'exécution dans la liste d'Attente, et va l'enlever de la liste Eligible.
- waitEvent() : L'ordonnanceur va enlever le comportement en cours d'exécution de la liste Eligible.
- waitInterruption() : L'ordonnanceur va enlever le comportement en cours d'exécution de la liste Eligible.

La section (ci-dessus) montre comment bloquer un comportement avec une des quatres primitives waits.

Mais comment l'ordonnaceur débloque ces comportements ?

Les primitives de l'ordonnanceur : Wake. Il existe trois primitives pour faire ceci :

- wakeUpFromWaitingTime() : Débloque le(s) comportement(s) qui est(sont) en attente sur du temps, ou qui ont fait une tâche avec durée.
- e.wakeUpFromWaitingEvent() : Débloque le(s) comportement(s) qui est(sont) en attente sur l'evenement e.
- sendInterruption() : Envoie une interruption qui débloque le comportement en attente de cet interruption et bloque l'exécution du comportement qui envoie l'interruption.

Implémentation de ces primitives :

- wakeUpFromWaitingTime() : L'ordonnanceur rajoute le comportement courant dans la liste Eligible et l'enlève de la liste d'Attente.
- e.wakeUpFromWaitingEvent() : L'ordonnanceur rajoute le comportement qui était un attente de l'événement e dans la liste Eligible.
- sendInterruption() : L'ordonnanceur rend la main au comportement qui attendait l'interruption, ensuite bloque l'exécution du comportement courant et le remet dans la liste d'Eligible.

Ordonnanceur. Les quatres primitives waits font appel à l'ordonnanceur puisque avant que le comportement courant se bloque, il est indispensable de débloquer un autre comportement, sinon la simulation va rester bloquée.

Cela se fait avec la fonction nextToRun(), cette fonction appartient à l'ordonnanceur et c'est celle qui va s'occuper du partage de temps du processeur entre les différents comportements de composants. Elle s'occupe tout d'abord d'enlever le comportement qui est en train de s'exécuter (comportement courant) de la liste Eligible (car il ne doit plus s'exécuter à cet instant). Ensuite, l'ordonnanceur va regarder s'il y a des comportements qui sont dans la liste Eligible (pour vérifier s'il y a encore des comportements qui doivent s'exécuter à cet instant).

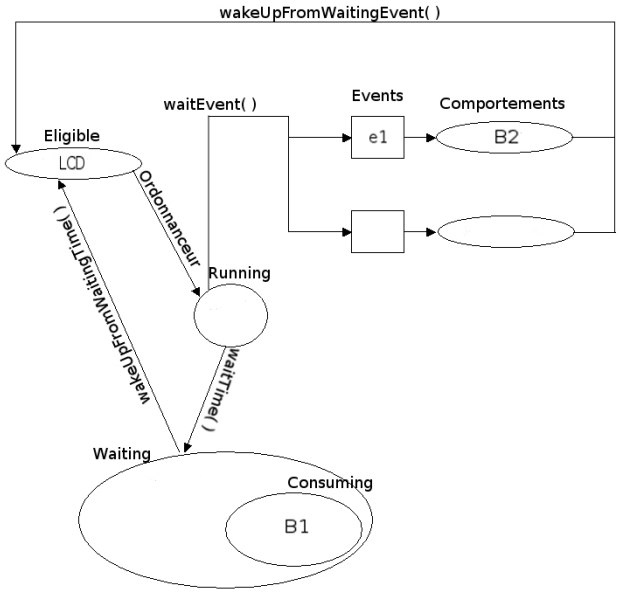

Fig. 7. Ensembles des comportements selon leur état

La figure 7 illustre les différents ensembles des comportements selon leur état, où B1 a effectué une tâche avec durée, B2 est en attente d'un événement e1, et B3 est dans Eligible (n'est pas encore entré en phase d'exécution).

Si la liste Eligible est vide, cela veut dire qu'il n'y a plus de comportements prêts à s'exécuter, l'ordonnanceur va donc regarder dans la liste d'Attente pour voir s'il y a des comportements qui sont en attente sur du temps ou qui ont fait une tâche avec durée. Si cette liste aussi vide, la simulation va se terminer. Sinon, l'ordonnanceur, va mettre les comportements qui sont en attente sur du temps, ou qui ont fait une tâche avec durée avec le temps minimal, de nouveau dans la liste Eligible, et avancer le temps de simulation.

Après l'ordonnanceur récupère le comportement suivant qui doit être exécuté (celui-ci va être à chaque fois le premier élément (comportement) de la liste Eligible) appelons ce comportement "leSuivant", ensuite l'ordonnanceur débloque ce comportement, et finalement bloque le comportement courant pour éviter d'avoir deux ou plus des comportements qui s'exécutent en même temps.

Algorithme que l'ordonnanceur utilise pour obtenir le nouveau comportement à mettre en exécution :

```
1  nextToRun(Behavior behav) {
2          boolean b = false;
3          Eligible.remove(behav);
4          Behavior leSuivant;
5
6          Si (Eligible vide) {
7                  if(Liste d'Attente vide){
8                          fin de la simulation
9                  }
10                 else{
11                 L'ordonnanceur va placer les comportements
12                 qui sont dans la Liste d'Attente avec un
13                 temps minimal dans la liste Eligible.
14                         wakeUpFromWatingTime();
15                 }
16         }
17
18         leSuivant = Eligible.remove(0);
19
20         Si (leSuivant En attente sur du temps){
21                 debloquer Lesuivant;
22         }
23 }
```

Exemple. Voici un exemple d'une simulation des trois comportements des trois composants différents en Jtlm coopératif. Rappel : B1, B2, B3 sont les comportements de P1, P2, LCD.

```
B1{
        consume(){
                Ecrire ();
        }.during(10};

        Ecrire ();
        wakeUpFromWaitingEvent();
        waitTime(10);
        Ecrire ();
        sendInterruption();
        Lire ();
}

B2{
        Ecrire ();
        waitEvent();
        Ecrire ();
        waitTime(5);
        Ecrire ();
}

B3{
        waitInterruption ();
        Lire ();
        Lire ();
}
```

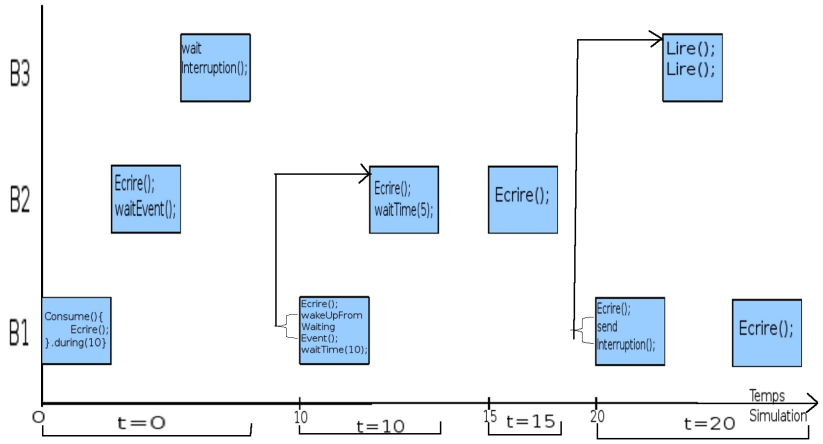

Fig. 8. Simulation des trois composants

Maintenant, nous allons expliquer les différentes étapes de la simulation dans un modèle d'exécution coopératif :

1. temps de simulation=0
   L'ordonnanceur choisit d'exécuter le comportement B1, celui-ci fait une tâche avec durée de 10 unités, il écrit dans la mémoire. Ensuite il donne la main à B2 et B1 se bloque (L'ordonnanceur enlève ce comportement d'Eligible). B2 écrit en mémoire aussi, ensuite donne la main à B3, et se met en attente d'un événement (L'ordonnanceur enlève ce comportement d'Eligible). B3 se met en attente d'une interruption (L'ordonnanceur enlève ce comportement d'Eligible).
   A ce moment là, la liste Eligible est vide, donc l'ordonnanceur va vérifier s'il y a des éléments dans la liste d'Attente. Comme elle contient B1 qui avait fait une tâche avec durée, L'ordonnanceur va avancer le temps de simulation, et va mettre B1 de nouveau dans la liste Eligible.
2. temps de simulation=10
   B1 écrit, et ensuite, l'ordonnanceur va mettre B2 dans la liste eligible. B1 continue son exécution, et fait une attente sur du temps de 10 unités, donc il donne la main à B2 et se bloque. Ensuite, B2 écrit, et se met en attente sur du temps de 5 unités.
   Comme il n'y a plus de comportements dans Eligible et la liste d'Attente n'est pas vide (contient B1, B2), l'ordonnanceur avance le temps et met dans Eligible B2 (parce que c'est le comportement qui a le temps le plus petit).
3. temps de simulation=15
   B2 continue son exécution, écrit en mémoire, et finalement se termine. Donc il est enlevé d'Eligible.
   Ceci implique qu'Eligible est à nouveau vide, et comme la liste d'Attente n'est pas vide (contient B1), l'ordonnanceur va avancer le temps de simulation mettre B1 dans Eligible.
4. temps de simulation=20
   Maintenant, B1 va écrire, et ensuite envoie une interruption à B3, donc l'ordonnanceur donne de suite la main à B3, bloque B1, et le rajoute dans la liste Eligible pour qu'il puisse continuer son exécution après.
   Donc B3 lit deux fois de suite et se termine, donc B1 prend la main, écrit et se termine.
   Comme la liste Eligible est vide, et la liste d'Attente est vide aussi, alors la simulation se termine.

Il faut faire attention avec ce principe, parce que admettons qu'un comportement entre dans une boucle infinie (livelock) [8], alors les autres vont être en famine (starvation) et toute la simulation va être bloqué.

3.2 Portabilité

Jtlm a un modèle d'exécution coopératif et préemptif. Comme Jtlm n'impose pas un modèle d'exécution, le code doit être portable en termes de modèle d'exécution.

Cette partie du stage consiste à isoler et donner des règles pour définir la notion de portabilité en termes de modèle d'exécution dans le cadre de ce simulateur. L'idée dans cette partie, est d'avoir la même API (côté utilisateur) dans les deux modèles d'exécution.

La manière dont un comportement commence son exécution a été changé. Avant le code de l'utilisateur était placé dans la méthode run (méthode qui est appelé quand un comportement commence son exécution).

```
1 run(){
2         Ecrire ();
3         waitTime(10);
4         Consume(){
5                 Ecrire ();
6         }
7 }
```

Maintenant, la méthode run est toujours appelé, mais cette méthode a changé, voici son contenu :

```
1 run(){
2         Waiting();
3         Execution();
4 }
```

Le code de l'utilisateur est donc placé dans une méthode appelé execution().

```
1 execution(){
2         Ecrire ();
3         waitTime(10);
4         Consume(){
5                 Ecrire ();
6         }
7 }
```

Ceci a été fait pour pouvoir cacher une méthode dont le mode coopératif avait besoin pour commencer son exécution et non pas le mode préemptif. De cette façon l'API du côte de l'utisateur reste égal dans le deux modèles.

Nous avons pu tester quelques cas de la simulation dans les deux modèles, comme les tâches instantanées, l'attente sur du temps, l'attente d'un événement, l'attente d'une interruption, et les tâches avec durée. Mais il reste d'autres cas , qui n'ont pas pu être testé pour une question de temps.

4 Conclusion et suggestions d'améliorations

En conclusion, Jtlm a maintenant un modèle d'exécution aussi bien coopératif que préemptif. Donc Jtlm est dans un certain sens plus général

que SystemC parce qu'on n'impose pas un modèle d'exécution particulier. Cependant, écrire du code pour Jtlm pose plus des contraintes au niveau de l'implantation car le code doit être portable en termes de modèle d'exécution.

Nous pouvons remarquer que dans le modèle coopératif la grande granularité dans les transactions peut cacher des bugs aux programmeurs, et donc le logiciel embarqué pourrait ne pas bien fonctionner. D'un autre côté, le modèle préemptif, ne cache pas ces bugs, mais grâce au mécanismes de synchronisation on peut réduire le nombre de bugs.

De fait on peut considérer que, pour garantir qu'un logiciel va fonctionner correctement, il faut trouver un juste compromis entre ces deux modèles d'exécution.

Suggestions d'améliorations :

1. Implémenter les tâches avec durée inconnue. Ceci consiste à l'implémentation d'une fonctionnalité dans la version coopérative de Jtlm, où un comportement pourrait effectuer une tâche sans avoir besoin de préciser le temps qu'elle va prendre.
2. Traiter le cas de polling dans la version coopérative. Le cas où il y a deux processeurs, un qui entre dans une boucle infinie (livelock) et l'autre qui est en famine (starvation).
3. Donner d'autres règles pour définir d'une façon plus exhaustive la portabilité en termes de modèle d'exécution.

5 Remerciements

Je remercie énormément Giovanni Funchal et Matthieu Moy qui m'ont beaucoup aidé à comprendre le sujet du stage, et qui m'ont toujours données des bonnes idées pendant toute la durée de mon stage. Merci aussi à Jean-Baptiste Cervera qui m'a aidé avec la rédaction de ce rapport.

Références

[1] Abdessaied, N. : Design of a java simulator for fast prototyping of system-on-chip (2008-2009), http://www-verimag.imag.fr/~funchal/docs/NabilaReport.pdf

[2] AISSAOUI, M.T.E. : Modélisation du temps dans un simulateur pour systèmes sur puce (2010), http://www-verimag.imag.fr/~funchal/docs/MohamedReport.pdf

[3] Boussinot, F. : Les threads et la programmation concurrente (2006), http://www-sop.inria.fr/indes/rp/FairThreads/FTJava/french-presentation/index.html

[4] Helmstetter, C. : Validation de modèles de systèmes sur puce en présence d'ordonnancements indéterministes et de temps imprécis. Ph.D. thesis, Institut National Polytechnique de Grenoble - INPG (3 2007)

[5] Microsystems, S. : Threads (2003), http://java.sun.com/j2se/1.4.2/docs/api/java/lang/Thread.html

[6] Moy, M. : Modélisation transactionnelle des systèmes sur puces (2009-2010), http://www-verimag.imag.fr/~moy/cours/tlm/

[7] Moy, M. : Techniques and Tools for the Verification of Systems-on-a-Chip at the Transaction Level. Ph.D. thesis, INPG, Grenoble, France (December 2005), http://www-verimag.imag.fr/~moy/phd/

[8] et Nir Shavit, M.H. : The art of Multiprocessor Programming (2008)

[9] (OSCI), O.S.I. : Systemc (2010), http://www.systemc.org/community/about_systemc/