Unité Mixte de Recherche 5104 CNRS - INPG - UJF
Centre Equation
2, avenue de VIGNATE
F-38610 GIERES
tel : +33 456 52 03 40
fax : +33 456 52 03 50
http://www-verimag.imag.fr

Modélisation du temps dans un simulateur pour systèmes sur puces

Mohamed Taoufiq EL AISSAOUI

18 Mai 2010

This report is downloadable at the following address

Modélisation du temps dans un simulateur pour systèmes sur puces

Mohamed Taoufiq EL AISSAOUI

Verimag

18 Mai 2010

Abstract

Ce rapport est une synthèse du travail effectué au laboratoire Verimag pendant le module TER concernant l'extension de la gestion du temps qui existait déjà dans le simulateur JTLM

Keywords: JTLM, Modélisation de temps, Simulateur sur puce, Java, SystemC, TLM

Tutor: Matthieu Moy & Giovanni Funchal

Notes:

Table des matières

Première partie
Introduction

1 Prototypage virtuel

Le prototypage virtuel [4] est un procédé de simulation qui permet de tester l'efficacité d'une architecture avant de l'implémenter sur des circuits électroniques.
De nombreux avantages s'offrent lors de l'utilisation d'un simulateur contrairement à des tests effectués directement sur matériel :

- L'utilisation de la simulation sur logiciel permet de réduire les coûts de production.
- Il permet aussi de pouvoir concevoir des puces alors que le matériel nécessaire n'est pas disponible par exemple.
- Cela permet aussi de valider l'architecture, et de prouver son bon fonctionnement.
- Et puis finalement évaluer les performances d'une architecture de façon très efficace.

Dans un prototypage virtuel, deux paramètres principaux entrent en jeu, et évoluent de façon prépondérée : précision et complexité. Plus un prototypage est précis, le modèle qu'il représente est proche de la réalité, mais plus le modèle devient complexe. Inversement, moins un modèle est complexe, plus il est facile de l'appréhender, mais il est moins fidèle à la réalité.
Cette prépondérence sera évoquée dans les chapitres qui suivent.

2 TLM

2.1 SystemC

SystemC [8] [4] est une bibliothèque de C++ qui permet de réaliser des simulations d'architecture.
Cette bibliothèque ne sera pas évoquée durant ce rapport, car ce qui nous intéresse dans notre cas c'est uniquement de pouvoir faire des comparaisons entre SystemC et notre simulateur. Cependant quelques précisions sur cette bibliothèque permettent de mieux situer le contexte :

- SystemC est dit coopératif, ce qui veut dire que chaque tâche doit explicitement déclarer qu'elle a fini son travail pour qu'une autre tâche vienne prendre le relais.
- La gestion du temps dans SystemC est faite de façon discrète. Le temps est considéré comme une unité de mesure qui est incrémentée explicitement : "Discrete Event Simulation".

2.2 Design Flow

Il existe plusieurs niveaux ou couches de simulation qui permettent de simuler une architecture. Chaque niveau a ses avantages ainsi que ses inconvénients. Plus on augmente le niveau d'abstraction, plus c'est facile de réaliser des simulations, mais moins on perd de précision, et de façon similaire, plus on va vers le bas niveau, plus on gagne en précision, mais la réalisation d'une simulation devient une tâche assez complexe à faire.
Dans notre cas nous traiterons TLM (Transaction-level modeling) [5] qui est considéré dans un niveau plutôt haut, car il fait abstraction de certains détails contrairement à une simulation RTL (Register Transfer Level, une modélisation bas niveau qui décrit l'implémentation sous forme d'éléments séquentiels et de combinaisons logiques), mais fournit un confort non négligeable de programmation, d'où sa réelle utilité.

3 JTLM

JTLM [1] est le nom du simulateur qui nous intéresse, et qui existait déjà avant le début de mon travail de recherche.
Ce simulateur, écrit en Java, est préemptif (contraire de coopératif, c'est-à-dire qu'une tâche n'a pas besoin

de lancer une tâche suivante, c'est fait de façon systématique). Il a été créé dans le but de pouvoir comparer les performances d'une simulation en JTLM et une simulation sur SystemC.
La principale motivation de JTLM, c'était de voir ce que donnerait un simulateur TLM « différent » de SystemC, pour comprendre ce qui est spécifique à SystemC et ce qui est général en TLM.
Néanmoins, un travail est en cours de réalisation par Rafael Vasquez (stagiaire à Verimag) qui consiste à permettre à JTLM de devenir coopératif aussi (L'utilisateur aura le choix de coopératif ou préemptif à la compilation).

3.1 Les tâches

Une tâche c'est un ensemble d'instructions qu'exécute un composant particulier. On distinguera par la suite la notion de "temps de simulation" par le temps qu'aurait pris une tâche pour être exécutée si on était dans une implémentation matérielle (c'est à dire le temps simulé). Et "wall-clock time" le temps que voit l'utilisateur du simulateur, c'est à dire le temps réel. La nouveauté dans JTLM est l'existence de tâches avec durée (contrairement à SystemC). Chaque tâche peut avoir explicitement une durée bien précise qui la définit, dans le sens de "temps simulé", ce qui offre un confort de programmation, ainsi qu'une très bonne lisibilité du code.
Toutefois, JTLM offre aussi la possibilité de gérer le temps à la même façon que SystemC, c'est à dire que l'utilisateur peut très bien faire passer le temps de façon explicite s'il ne souhaite pas donner une durée à une tâche.
Il existe aussi un autre type de tâches dans JTLM : Les tâches instantanées. Ces tâches modélisent un phénomène instantané, et n'altèrent en aucun cas l'avancement du temps simulé, bien que, pour l'utilisateur, ces tâches peuvent être conséquentes (visible en wall-clock time).
Pour résumer, les types de tâches sont :

- Tâche avec durée (exécution).
- Tâche en attente (wait).
- Tâche instantanée.

4 Contributions

4.1 Tâches à durées inconnues : unknown

Mon sujet de recherche concernait les tâches avec durée, et plus précisément, permettre à JTLM d'offrir la possibilité de définir des tâches avec des durées non définies (toujours au sens de "temps simulé").
C'est à dire que dorénavant, l'utilisateur peut créer une tâche, et préciser à JTLM qu'elle doit elle même s'arranger pour ne pas bloquer le temps, et de considérer que cette tâche a une durée dynamique qui dépend du sort des autres tâches.
Une explication plus exhaustive du travail effectué sera présentée plus loin dans ce document.

4.2 Contributions techniques

En plus des contributions citées plus haut, plusieurs modifications techniques ont été apportées au simulateur JTLM notamment pour permettre une meilleure lisibilité du code. Ainsi quelques optimisations ont été apportées pour une simulation plus rapide et plus sûre (Nous n'entrerons pas dans les détails de ces contributions, car ce n'est pas le sujet de ce document).

Deuxième partie
JTLM

5 Java

Java est un langage de programmation orienté objet [6]. Il a la particularité d'être portable, grâce à son mode de fonctionnement sous une machine virtuelle.
Le choix de Java pour développer JTLM a été judicieusement fait à cause de la facilité de ce langage, ainsi qu'aux fonctionnalités qu'il offre et que nous détaillerons un peu plus loin.

6 Les Threads

Un thread correspond à une tâche qu'exécute le processeur [3].
Les threads sont créés par des processus, et ont un comportement très similaire à ce dernier, d'ailleurs une autre appellation d'un thread est : processus léger. Contrairement à un processus lourd, un thread partage sa zone mémoire avec le processus qui l'a créé, ainsi qu'avec le groupe de threads créés par ce même processus. La seule chose qu'il a de façon individuelle c'est sa zone de données (zone de données binaires), ainsi que sa pile d'exécution.
Les threads offrent des avantages et des inconvénients par rapport à un programme type séquentiel (Un programme dont toutes les instructions sont exécutées l'une après l'autre sans aucun parallélisme).

6.1 Avantages des threads

Sachant que les threads sont vus par le/les processeur(s) quasiment comme des processus, ceci permet de réaliser un parallélisme entre les différentes tâches.
Le partage de la zone mémoire offre la possibilité de communiquer entre les threads d'une façon très aisée (contrairement à une utilisation de messages type boîte aux lettres...).

6.2 Inconvénients des threads

Le partage de mémoire malheureusement pose quelques problèmes de synchronisation [7]. Deux threads qui essaient de modifier simultanément une même donnée peut mener à corrompre la valeur de cette donnée, ce qui peut nuire totalement au bon fonctionnement du programme.
Pour cela, Il faut prendre des précautions supplémentaires pour empêcher ce genre d'incident.

7 Les threads en Java

En Java, la gestion des threads est faite de façon native au langage [6]. C'est à dire que la notion de threads est intégrée dans le langage lui même, contrairement à d'autres langages comme C ou C++ qui utilisent des librairies tierces pour manipuler les threads.
Cette intégration native permet de faire des optimisations à la compilation [2] que d'autres langages ont du mal à faire.

8 Synchronisation

Comme indiqué dans la partie 6.2, l'utilisation des threads peut provoquer des problèmes de synchronisation.
Java a prévu quelques mécanismes qui permettent de synchroniser les threads [6], par exemple le mot clé synchronized permet d'exécuter des sections critiques (c'est à dire des sections qui doivent être exécutées de façon exclusive) de façon exclusive, sans pouvoir laisser d'autres threads y entrer.

9 JTLM en détail

JTLM [1] est programmée en Java, et utilise énormément les threads, car chaque composant de la simulation est attribué à un thread. Ceci permet d'un coté d'augmenter les performances de la simulations (parallélisme), mais aussi d'avoir un coté plus réel de la simulation, car cela permet de simuler de façon très réaliste le comportement des composants sur puces, et ceci grâce aussi au coté préemptif de JTLM, car ceci implique que les composants fonctionnent de façon parallèle (threads) et indépendante (préemptif), contrairement aux simulateurs coopératifs, où chaque tâche doit explicitement déclarer la fin de son travail pour qu'une tâche commence la sienne.

9.1 Le temps dans JTLM

JTLM gère un ensemble de comportements, et il faut les exécuter en parallèle ou tour à tour. JTLM repose sur le scheduler Java pour exécuter les threads éligibles (c'est-à-dire les threads qui sont prêts), mais dispose aussi d'un autre gestionnaire qui est le "time-manager" qui a pour rôle de bloquer les threads qui ne doivent pas (ou pas encore) s'exécuter. Par abus de langage, nous appellerons ce "time-manager" : scheduler JTLM. Le scheduler JTLM gère le temps en maintenant à jour une liste de tâches. Cette liste peut contenir trois types de tâches : les tâches en cours d'exécution, les tâches qui attendent un temps donné, et les tâches qui ont fini leur travail et attendent d'être supprimées de la liste (Ces tâches seront appelées "des tâches zombies"). Le temps de la simulation est modifié au fur et à mesure que cette liste est parcourue.

9.2 Algorithme du scheduler JTLM

Le scheduler JTLM [1] utilise une liste de tâches, avec une étiquette pour chacune d'elles afin de définir si une tâche est en exécution, ou en attente ou en zombie.
À chaque nouvelle définition d'une tâche avec durée, le scheduler JTLM ajoute cette tâche dans sa liste qui est en fait une file de priorité, avec comme priorité la durée de la tâche. Une fois qu'une tâche a fini de s'exécuter, le sheduler JTLM va consulter la tâche de plus petite durée dans sa file de priorité, et voir si elle correspond à la tâche qui vient de terminer. Si oui, alors il enlève cette tâche de la liste, et continue. Sinon, cela veut dire qu'en "wall-clock time" la tâche est finie, mais qu'en "temps simulé" ce n'est pas encore le cas. Donc la tâche est rendue zombie, pour garder une certaine cohérence de simulation, et elle ne sera enlevée de la file qu'une fois le "temps simulé" le permettra. S'il s'avère que la tâche de plus petite durée est une tâche qui est en attente (a fait un wait), alors que le temps de simulation passé permet de la réveiller, alors le sheduler JTLM réveille le thread qui correspond à cette tâche, et l'enlève de la liste.
La figure 1 [1] représente un exemple qui sera traité pour mieux illustrer l'algorithme du scheduler JTLM.
B1 et B2 sont attribués à deux threads différents, et modélisent deux processeurs qui fonctionnent en pa-

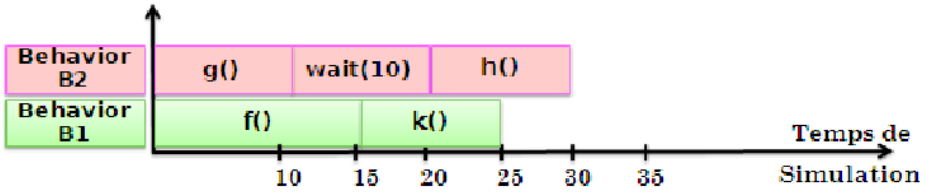

FIG. 1 – Exemple

rallèle.
Au début les deux processeurs exécutent en même temps f() et g(), pendant que le scheduler JTLM a ajouté ces deux tâches dans sa file de priorité. Dans notre cas, la tâche de plus petite durée c'est g(), alors le scheduler va attendre la fin de cette dernière pour pouvoir faire avancer le temps. Si par exemple f() finit avant, elle passe en mode zombie.
Une fois que g() termine, le temps simulé est incrémenté, et la tâche g() est enlevée de la file (maintenant

la file contient une seule tâche qui est f()).
Après vient le processeur-2 qui fait un wait(10), alors cette tâche sera ajoutée à la file de priorité, et le thread sera endormi. Une fois que la tâche f() aura fini, elle sera aussitôt enlevée de la file, et le temps de simulation sera incrémenté. Et ainsi de suite jusqu'à la fin de la simulation.
Le diagramme de la figure 2 [1] retrace le cours d'exécution de cette simulation.

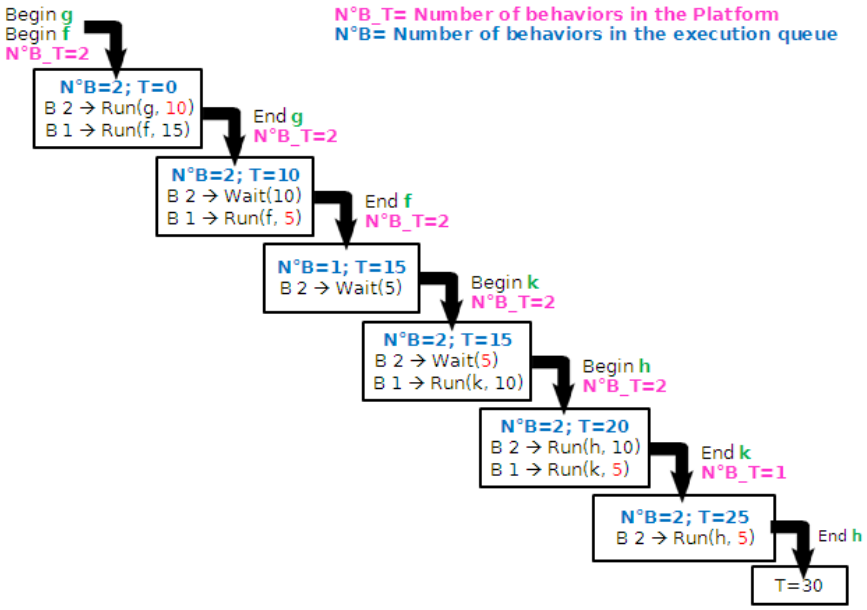

FIG. 2 – Déroulement de l'exécution

Troisième partie
Les tâches “unknown”

10 Motivations

Après la fonctionnalité qu’a offerte JTLM qui consiste à définir des tâches avec des durées, il s’est avéré qu’il serait intéressant de se pencher vers une extension de cette fonction, qui consisterait à définir des tâches avec des durées indéterminées.
Prenons l’exemple classique du polling, avec deux processeurs, et une RAM partagée.

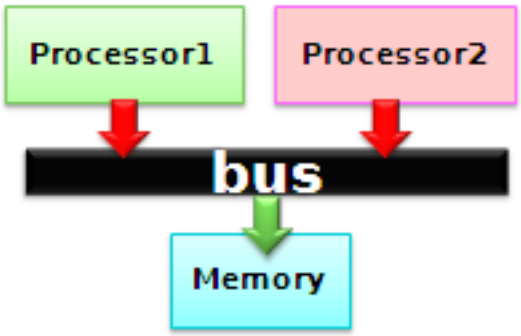

FIG. 3 – Deux processeurs et une RAM

Initialement, la zone partagée en mémoire a une valeur 0.
Le processeur-1 fait :

```
repete 3 fois {
  while (lire(zone_partage) == 0) {
      rien;
  }
  ecrire(zone_partage,0);
}
```

Le processeur-2 fait :

```
repete 3 fois {
  attendre(30);
  ecrire(zone_partage,1);
}
```

Explications :
- La fonction “lire” prend en paramètre une adresse mémoire, et retourne la valeur de la case mémoire correspondante.
- La fonction “ecrire” prend en paramètre une adresse mémoire et une valeur, et écrit la valeur donnée en paramètre dans l’emplacement mémoire qui correspond à l’adresse mémoire passée en paramètre.

Malheureusement ce programme ne pourra pas fonctionner correctement, car le premier processeur prendra la possession de l’exécution, et empêchera le scheduler JTLM d’avancer le temps, ce qui aura pour effet de ne jamais réveiller le processeur-2 car il sera toujours en train d’attendre 30. Il y aura famine, c’est-à-dire qu’un des processus n’aura jamais la parole tandis qu’un autre monopolise l’exécution.

11 Ancienne solution

Pour remédier au problème évoqué précédemment, on n'avait qu'une seule solution, c'était de mettre des petits temps d'attente dans la boucle du polling afin de faire avancer le temps (c'est également la solution utilisée dans les programmes écrits en SystemC).
Le code du premier processeur devient :

```
repete 3 fois {
  while (lire(zone_partage) == 0) {
      attendre(1);
  }
  ecrire(zone_partage,0);
}
```

De cette façon, le scheduler JTLM va incrémenter le temps au fur et à mesure, jusqu'à ce que le temps de simulation soit égal à 30 pour réveiller le deuxième processeur, et continuer normalement l'exécution.
Cette solution est lourde vis à vis de ce qu'offre JTLM comme fonctionnalités, et il aurait été dommage si on laissait uniquement cette solution à l'utilisateur.
C'est comme cela que la motivation de faire des tâches de durées indéterminées a commencé.

12 Tâches Unknown

Avec les tâches unknown, il est désormais possible de définir des tâches à durées indéterminées, qui n'empêcheront pas l'avancement du temps de simulation.
Pour la suite du rapport, nous prendrons ces deux conventions de syntaxe :

Tâches avec durée :

```
  tache_avec_duree {
    tache;
  }.during(x);
```

Cette syntaxe veut dire qu'on déclare la tâche "tache" en tant que tâche avec durée, et en lui attribuant "x" comme durée.
Tâches à durée indéterminée :

```
  tache_unknown {
    tache;
  }
```

On déclare la tâche "tache" comme une tâche à durée indéterminée.

Maintenant, en reprenant l'exemple cité plus haut, on peut faire :

```
repete 3 fois {
  tache_unknown {
    while (lire(zone_partage) == 0) {
      rien;
    }
  }
  ecrire(zone_partage,0);
}
```

Ceci aura pour effet de déclencher le polling, sans pour autant bloquer le temps. Ce qui permettra au scheduler JTLM de faire avancer le temps quand même pour arriver à atteindre 30, ce qui réveillera le deuxième processeur.

13 Algorithme des tâches unknown

L'implémentation de tâches unknown a été faite en introduisant une nouvelle liste qui contient les tâches déclarées en tant que tâches à durées indéterminées, afin de ne pas gêner la file de priorité expliquée plus haut. Quand une tâche unknown finit son travail, le scheduler JTLM l'enlève de la liste, sans pour autant toucher à la file de priorité ni au temps simulé. De cette façon, le scheduler JTLM peut faire avancer le temps sans problèmes, tout en tenant compte des tâches qui sont unknown.
Reprenons l'exemple de la figure 1, avec un petit changement : B2 fait une tâche unknown après avoir fait g() au lieu de faire wait(10).
Au départ, rien ne change, les deux processeurs commencent à exécuter f() et g(), et ces deux tâches sont ajoutées à la file de priorité. Après on a toujours le cas où f() finit avant g() dans lequel cas f() sera mise en zombie.
Donc après que g() termine, elle est retirée de la file, le temps de simulation est incrémenté, et là, le scheduler voit qu'il y a une tâche unknown à exécuter. Il l'ajoute alors à la liste des tâches unknown pendant que le processeur-2 exécute cette tâche unknown.
Plusieurs scénarios peuvent arriver maintenant, nous allons en détailler un : f() finit avant la tâche unknown du processeur-2, et la tâche unknown termine avant k() : f() est retiré de la file, le temps de simulation est incrémenté, et k() entre en jeu. Pendant que k() est en train de s'exécuter, la tâche unknown finit son travail, alors la scheduler JTLM l'enlève de la liste des tâches unknown, et fait rentrer h() dans la file de priorité (à l'entrée de h(), le temps de simulation vaut 15). k() termine, donc elle est retirée de la file de priorité, le temps de simulation vaut 25, puis h() termine, et le temps de simulation vaut 30.
Le déroulement des situations dépend énormément du temps que prend la tâche unknown. Par exemple si elle avait fini avant f(), le temps de simulation final serait 25. Si la tâche unknown finit après k(), le temps de simulation final serait de 35.
On ne peut pas déterminer à l'avance le sort d'une simulation, et d'ailleurs c'est le but d'une tâche unknown, car son résultat n'est déterminé qu'à l'exécution.

Quatrième partie
Expérimentations

14 D'autres avantages

En plus des avantages de la fonctionnalité unknown cités précédemment, il s'est avéré que cette fonctionnalité offre plus de choses que prévu. Elle permet de reproduire des bugs qui peuvent éventuellement exister dans une implémentation sur matériel.
Lors des tests de l'exemple du Polling expliqué plus haut, la plupart des cas le programme échouait. Tandis qu'avec la solution du chapitre 11, le programme finissait toujours avec succès.
En fait le scheduler Java peut donner la main plus longtemps au processeur-2, ce qui aura pour effet par exemple d'exécuter trois fois :

```
attendre(30);
ecrire(zone_partage,1);
```

et terminer, alors que le processeur-1 est encore dans sa première boucle, car le scheduler Java ne lui a pas encore laissé le temps pour s'exécuter.
Ce qui se passe, c'est que après que le processeur-2 ait fini son travail, le processeur-1 fait une fois sa boucle while ainsi que la fonction "ecrire" :

```
while (lecture(zone_partage) == 0) {
  rien;
}
ecrire(zone_partage,0);
}
```

Puis il reste bloqué indéfiniment car le processeur-2 ne va plus réécrire 1 dans la zone mémoire partagée, il a déjà écrit 3 fois dans cette zone, et a déjà fini son travail.
Ce comportement plutôt étrange reflète un cas réel où l'on aurait deux processeurs avec des vitesses d'exécution très différentes. Dans notre exemple, le processeur-2 est très rapide devant le processeur-1. Pour le processeur-2, 30 se sont écoulés très vite, alors que le processeur-1 était encore en train d'exécuter pour la première fois son while.

15 D'autres exemples

Un autre exemple intéressant est l'algorithme de Peterson des sections critiques.
On prend l'exemple de 2 processeurs et une RAM partagée. Les 2 processeurs doivent entrer dans une section critique en utilisant l'algorithme de Peterson.
Voici en pseudo code l'algorithme utilisé (Pour la syntaxe exacte cf. l'annexe) :

Processeur 1 :

```
ecrire(my_flag,1);
ecrire(turn,0);

/* Tentative d'entrer dans la section critique */
tache_unknown {
  while (lire(other_flag) == 1 && lire(turn) == 0) {
    rien;
  }
}
```

```
/* Section critique */
tache_avec_duree {
  ecrire(11,11); // Juste pour l'illustration
}.during(30);

/* Fin de la section critique */
ecrire(my_flag,0);
```

Processeur 2 :

```
ecrire(my_flag,1);
ecrire(turn,1);

/* Tentative d'entrer dans la section critique */
tache_unknown {
  while (lire(other_flag) == 1 && lire(turn) == 1) {
    rien;
  }
}

/* Section critique */
tache_avec_duree {
  ecrire(10,10); // Juste pour l'illustration
}.during(30);

/* Fin de la section critique */
ecrire(my_flag,0);
```

Explications :

- my_flag correspond à une adresse mémoire, qui est vue par l'autre processeur comme "other_flag". Par exemple si le processeur-1 écrit 5 à l'adresse my_flag, le processeur-2 lira 5 à l'adresse other_flag.
- turn par contre est une adresse mémoire globale qui ne change pas de sémantique d'un processeur à un autre. Si le processeur-1 écrit 5 dans turn, la processeur-2 lira 5 dans turn.

En sortie, ce programme fait bien ce qu'on souhaite faire, c'est à dire protéger la section critique, mais aussi incrémenter correctement le temps de simulation. On trouve un temps de simulation qui vaut 60 (30 + 30). Le diagramme de la figure 4 reflète l'exécution de la simulation en supposant que le processeur-1 a réussi à passer avant le processeur-2.

Ce programme illustre bien la facilité et le confort qu'offrent les tâches de durées indéterminées.

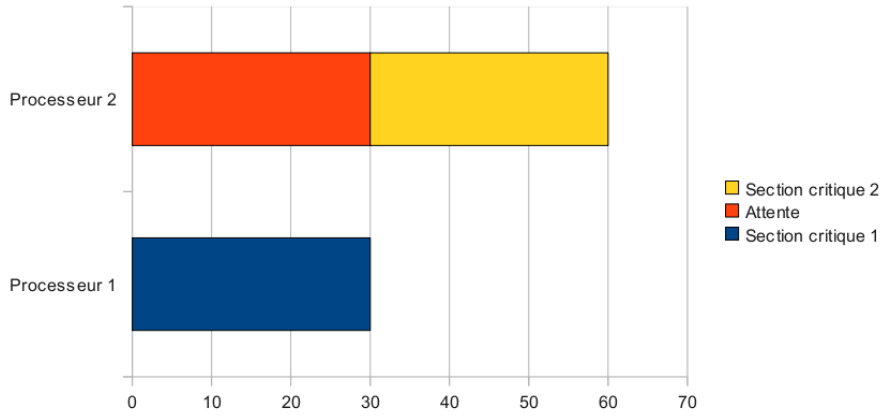

FIG. 4 – Diagramme d'exécution

Cinquième partie
Conclusion

16 Résumé

JTLM est un simulateur écrit en Java qui offre la possibilité de définir des tâches avec des durées. Le but de ce TER était d'implémenter la notion de tâches à durées indéterminées dans le simulateur JTLM. Après avoir fini, on se rend compte que cette fonctionnalité offre plusieurs perspectives, notamment une bonne lisibilité du code, découverte de bugs, simulation d'une situation tout en étant proche de la réalité, implémenter aisément des algorithmes qui autre fois étaient complexes...

17 Perspectives

Ce qu'on peut ajouter à JTLM, est la fonction coopérative (travail en cours de réalisation par Rafael Velasquez, stagiaire à Verimag), puis comparer les performances des simulations pour voir si finalement le choix du préemptif était judicieux ou pas.
Puis on peut essayer de voir si avec une implémentation coopérative on peut facilement adapter les tâches unknown...

18 Remerciements

Un très grand merci à Matthieu Moy qui sans lui ni ce rapport ni même ce TER auraient eu lieu.
Un merci également à Giovanni Funchal qui m'a aidé à mieux comprendre le fonctionnement de JTLM, et à trouver des exemples d'illustration pertinents.
Et finalement merci à Florence Maraninchi d'avoir créé le module TER, ce fut fort intéressant.

Sixième partie
Annexe

19 Polling

Processeur 1 :

```
package test.time_polling;


import jTlm.base.*;
import jTlm.exception.AddressOutOfRangeException;


public class Processor1 extends Component {
  public MasterPort initiatorPort = new MasterPort();
  public SlavePort targetPort = new SlavePort();


  private int memoryRegisterAddress;
  final static int  POLLING_MEMORY_ADDRESS = 0 ,
 ITC_ADDRESS = 110 ;
  final static int CPU_ADDRESS = 100 ,
 CPU_MEMORY_REGISTER_ADDDRESS = 1;
  final static int SET_TO_ZERO = 0;
  public Processor1() {
    targetPort.setOwner(this);
    new ReadingBehavior();

  }

  private class ReadingBehavior extends Behavior {
    public    void run () {

      for (int i = 0; i < 5; i ++) {
        new Consume(this){
          public void consume(){
            h();
          }
        }.unknown();
      }
      removeMeFromBehaviorGroup();
    }

    private void h() {

      try {
        System.out.println("Avant polling " +
        Thread.currentThread().getName());
        while (initiatorPort.read(POLLING_MEMORY_ADDRESS)
        ==SET_TO_ZERO) {
        }
```

```
            System.out.println("Apr\'es polling " +
            Thread.currentThread().getName());

          } catch (AddressOutOfRangeException e) {
            e.printStackTrace();
          }
          try {
            initiatorPort.write(POLLING_MEMORY_ADDRESS,
            SET_TO_ZERO);
          } catch (AddressOutOfRangeException e) {
            e.printStackTrace();
          }
          read();
        }

        private void read(){
          int data = 0;
          try {
            data = initiatorPort.read(memoryRegisterAddress);
          } catch (AddressOutOfRangeException e) {
            e.printStackTrace();
          }
          Debug.debugMessage(Thread.currentThread().getName()+
          " Transaction OK: read  " + data + " at address "
          + memoryRegisterAddress);
        }
      }

      public void write(int address,int data) throws
      AddressOutOfRangeException{
        if(( address ==  CPU_MEMORY_REGISTER_ADDDRESS) ){
          memoryRegisterAddress = data;
          //Debug.traceMessage(address );
        }
        else
        throw new AddressOutOfRangeException("!! CPU
      Address outside of range");

      }

    }
```

Processeur 2 :

```
  package test.time_polling;

  import jTlm.base.Behavior;
  import jTlm.base.Component;
  import jTlm.base.Debug;
  import jTlm.base.MasterPort;
  import jTlm.exception.AddressOutOfRangeException;
```

```
public class Processor2 extends Component {
  public MasterPort initiatorPort = new MasterPort();
  final static int  MEMORY_ADDRESS = 2,
  POLLING_MEMORY_ADDRESS = 0 ,
  SET_TO_ONE = 1;
  final static int CPU_ADDRESS = 100 ,
  CPU_START_ADDRESS =0 ,
  CPU_MEMORY_REGISTER_ADDDRESS = 1;


  public Processor2() {
    new WriteBehavior();
  }


  private class WriteBehavior extends Behavior {

    public    void run () {
      int data =1;

      for (int i = 0; i < 4; i += 1) {
        data = i + 1;
        write(data);
        waitTime(30);
      }
      data += 1;

      write(data);
      removeMeFromBehaviorGroup();
    }
    private void write(int data){
      try {
        initiatorPort.write(MEMORY_ADDRESS,
        data);
      } catch (AddressOutOfRangeException e) {
        e.printStackTrace();
      }
      Debug.debugMessage(
      Thread.currentThread().getName()+
      "␣Transaction␣OK:␣wrote␣" + data +
      "␣at␣address␣" +
      MEMORY_ADDRESS);

      try {
        initiatorPort.write(CPU_ADDRESS+
        CPU_MEMORY_REGISTER_ADDDRESS,
        MEMORY_ADDRESS);
        initiatorPort.write(
        POLLING_MEMORY_ADDRESS,
        SET_TO_ONE );
      } catch (AddressOutOfRangeException e) {
        e.printStackTrace();
      }
    }
```

```
    }
}
```

20 Peterson

Processeur 1 :

```
import jTlm.base.*;
import jTlm.exception.AddressOutOfRangeException;


public class Processor1 extends Component {
  public MasterPort initiatorPort = new MasterPort();
  public SlavePort targetPort = new SlavePort();


  private int memoryRegisterAddress;
  final static int  MY_MEMORY_ADDRESS = 1 ,
  OTHER_MEMORY_ADDRESS = 0 ,
  TURN = 2, ITC_ADDRESS = 110 ;
  final static int CPU_ADDRESS = 100 ,
  CPU_MEMORY_REGISTER_ADDDRESS = 1;
  final static int FALSE = 0, TRUE = 1;
  public Processor1() {
    targetPort.setOwner(this);
    new ReadingBehavior();

  }

  private class ReadingBehavior extends Behavior {
    public     void run () {
      new Consume(this){
        public void consume(){
          demande();
        }
      }.unknown();

      new Consume(this){
        public void consume(){

          try {
            initiatorPort.write(11,11);
          } catch (AddressOutOfRangeException e) {
            e.printStackTrace();
          }

        }
      }.during(30);

      fin();

      removeMeFromBehaviorGroup();
```

```
    }

    private void demande() {

      try {

        initiatorPort.write(MY_MEMORY_ADDRESS,TRUE);
        initiatorPort.write(TURN,FALSE);
        while (initiatorPort.read(OTHER_MEMORY_ADDRESS)==TRUE
        && initiatorPort.read(OTHER_MEMORY_ADDRESS)==FALSE) {
        }

      } catch (AddressOutOfRangeException e) {
        e.printStackTrace();
      }
    }

    private void fin() {

      try {
        initiatorPort.write(MY_MEMORY_ADDRESS,FALSE);
      } catch (AddressOutOfRangeException e) {
        e.printStackTrace();
      }
    }
  }
```

Processeur 2 :

```
import jTlm.base.*;
import jTlm.exception.AddressOutOfRangeException;


public class Processor2 extends Component {
  public MasterPort initiatorPort = new MasterPort();
  public SlavePort targetPort = new SlavePort();


  private int memoryRegisterAddress;
  final static int  MY_MEMORY_ADDRESS = 0 ,
  OTHER_MEMORY_ADDRESS = 1 ,
  TURN = 2, ITC_ADDRESS = 110 ;
  final static int CPU_ADDRESS = 100 ,
  CPU_MEMORY_REGISTER_ADDDRESS = 1;
  final static int FALSE = 0, TRUE = 1;
  public Processor2() {
    targetPort.setOwner(this);
    new ReadingBehavior();

  }

  private class ReadingBehavior extends Behavior {
    public    void run () {
      new Consume(this){
```

```
        public void consume(){
          demande();
        }
      }.unknown();

      new Consume(this){
        public void consume(){

          try {
            initiatorPort.write(10,10);
          } catch (AddressOutOfRangeException e) {
            e.printStackTrace();
          }

        }
      }.during(30);

      fin();

      // after finishing his job this behavior remove his
      self from the behavior group
      removeMeFromBehaviorGroup();
    }

    private void demande() {

      try {

        initiatorPort.write(MY_MEMORY_ADDRESS,TRUE);
        initiatorPort.write(TURN,TRUE);
        while (initiatorPort.read(OTHER_MEMORY_ADDRESS)==TRUE
        && initiatorPort.read(OTHER_MEMORY_ADDRESS)==TRUE) {
        }

      } catch (AddressOutOfRangeException e) {
        e.printStackTrace();
      }
    }

    private void fin() {

      try {
        initiatorPort.write(MY_MEMORY_ADDRESS,FALSE);
      } catch (AddressOutOfRangeException e) {
        e.printStackTrace();
      }
    }
  }
```

Septième partie

Bibliographie

Références

[1] Nabila Abdessaied. Design of a java simulator for fast prototyping of system-on-chip. Master's thesis, VERIMAG, Grenoble, France, 2008. 3, 9, 9.2, 9.2

[2] Bruno Blanchet. Escape Analysis for Java(TM). Theory and Practice. *ACM Transactions on Programming Languages and Systems*, 25(6) :713–775, November 2003. 7

[3] Paul Hyde. *Java Thread Programming*. Sams, 2001. 6

[4] Florence Maraninchi, Matthieu Moy, Jérôme Cornet, Laurent Maillet Contoz, Claude Helmstetter, and Claus Traulsen. SystemC/TLM Semantics for Heterogeneous System-on-Chip Validation. In IEEE, editor, *2008 Joint IEEE-NEWCAS and TAISA Conference 2008 Joint IEEE-NEWCAS and TAISA Conference*, page unknown, Montréal Canada, June 2008. B.6.3, D.2.4, D.3.1, F.4.3, F.3.1, B.8.1. 1, 2.1

[5] Matthieu Moy. *Techniques and Tools for the Verification of Systems-on-a-Chip at the Transaction Level*. PhD thesis, INPG, Grenoble, France, December 2005. 2.2

[6] Oracle. *The Java Language Specification*. 5, 7, 8

[7] Noel De Palma. Cours de sepc à l'ensimag 2a. 6.2

[8] systemC. *systemC Manual*. 2.1