Rapport Technique

Laboratoire d'Informatique Fondamentale de Lille
Université Lille1, Villeneuve d'Ascq

J-M. Vincenti, C. Alias et L. Gonnord
Juillet 2012

Simulation et Visualisation de Processus Communicants

Table des matières

1 Introduction - Contexte 3

2 CRP : Processus réguliers communicants 3

3 Simulateur de processus réguliers communicants 3
3.1 Objectif . . . 3
3.2 Les processus réguliers communicants . . . 4
3.3 Langage d'entrée . . . 4
3.4 Réalisation du simulateur . . . 5
3.4.1 Les deux étapes du simulateur . . . 5
3.4.2 Représentation intermédiaire pour les CRP . . . 6
3.4.3 Création de l'historique . . . 8

4 Interface graphique de simulation 9
4.1 Objectif . . . 9
4.2 Choix des technologies . . . 9
4.3 Copies d'écran . . . 11

5 Conclusion 12

1 Introduction - Contexte

Dans le cadre du PEPS CNRS INS2I “High Level synthesis of accelerators for Real-time programs” nous étudions le modèle CRP (Communicating regular processes), qui nous semble être un modèle adéquat en terme d’expressivité, même si pour le moment aucune contrainte de temps n’est exprimée dans le modèle.

Le but de ces travaux est de réaliser un simulateur d’exécution de tels processus. Le modèle d’exécution peut être trouvé dans [1]. Il s’agit ici de récupérer la sortie d’un compilateur C vers CRP, qui a été écrit par C. Alias, et de réaliser une interface graphique qui permet de simuler une exécution.

L’outil de visualisation de CRP se découpe en deux parties. La première se comporte comme un simulateur. En effet l’outil prend, comme donnée, un historique d’actions d’un ensemble de processus sur des tableaux (dit buffer). Ainsi, il a pour but de simuler le fonctionnement des processus distincts afin de détecter d’éventuelles anomalies. L’objectif est avant tout de visualiser le comportement du système. Il peut être utilisé pour vérifier sa convergence, la bonne répartition de l’activité des processus, etc. L’objectif est donc de proposer un outil permettant de simuler le fonctionnement du système et de construire un historique des actions effectuées en étapes, ou c̈oup d horlogep̈our avoir une représentation temporelle.

Cet historique est le point de départ de la seconde partie de l’outil. Une fois la simulation effectuée, le logiciel est utilisé pour représenter graphiquement cet historique. L’interface doit permettre de visualiser schématiquement chaque processus et chaque buffer de façon statique, puis de montrer à chaque étapes les lectures et écritures. Cette représentation doit être claire, le principal défi est d’afficher un maximum d’informations sans entraver la lisibilité. Du point de vue de l’utilisateur, cette partie sera la seule visible, ainsi elle doit illustrer les résultats obtenus lors de la simulation le plus précisément possible.

Dans ce rapport, nous décrivons les fonctionnalités de l’outil ainsi que les détails d’implémentation.

2 CRP : Processus réguliers communicants

Un CRP (communicating regular processes) est un réseau de processus qui communiquent par des canaux addressables (buffers). Les processus sont *réguliers*, dans la mesure où ils exécutent toujours une séquence prédéterminée d’opérations (lectures, calcul, écritures). Cette caractéristique est essentielle, puisqu’elle rend possible de nombreuses analyses et optimisations.

Les réseaux de processus (et les CRP en particulier) constituent un modèle d’exécution, qui exprime explicitement le parallélisme pipeliné et les transferts mémoire. Dans cette mesure, les CRPs peuvent être vus comme la représentation intermédiaire d’un compilateur paralléliseur. Cette représentation intermédiaire peut ensuite être compilée sous la forme d’un circuit, ou plus classiquement vers une machine parallèle.

Dans ce contexte, il existe un fort besoin en outils pour faciliter la compréhension, l’analyse et le débuggage d’une telle représentation intermédiaire. Ce travail y apporte une réponse préliminaire.

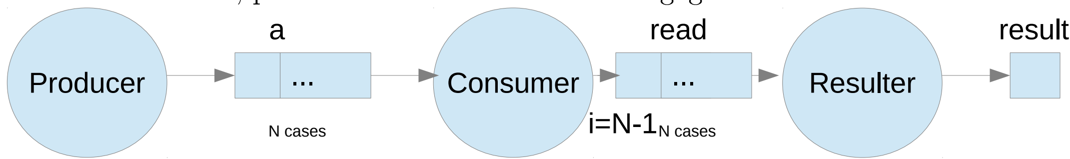
3 Simulateur de processus réguliers communicants

3.1 Objectif

L’outil de visualisation de CRP se découpe en deux parties. La première se comporte comme un simulateur. Il a pour but de représenter le fonctionnement des processus distincts. Ainsi l’outil sert avant tout à visualiser le comportement du système. Il peut être utilisé pour vérifier sa convergence, la bonne répartition de l’activité des processus, etc. L’objectif est donc de proposer un outil permettant de simuler le fonctionnement du système et de construire un historique des actions effectuées, pour les visualiser dans un second temps.

3.2 Les processus réguliers communicants

Dans le cadre des CRP, un processus est un ensemble de tâches réalisées de façon itérative. Les tâches correspondent à des lectures et écritures dans des tableaux d'entiers et sont dépendantes de l'accessibilité de leurs éléments. Les processus s'exécutent en parallèle et communiquent via ces buffers. Un buffer est représenté sous forme d'un tableau de dimension quelconque. La première partie du simulateur a pour but de déduire d'un langage d'entrée une représenation interne du système, comprenant l'ensemble des buffers, processus et leurs têches. Ce langage d'entrée est issu d'un code C.



3.3 Langage d'entrée

Le langage d'entrée est déduit (par compilation) d'un code C, composé d'un ensemble de boucles imbriquées et d'affectations. Ces affectations correspondent à des lectures et écritures dans des tableaux et variables. Voici un exemple de code C :

```
#define N 16
#define T 4

int main(void)
{
  int a[N], read;
  int i;

#pragma begin_scop

  for(i=0; i<N; i++)
#pragma schedule[i]
    a[i] = i;

  for(i=0; i<N; i++)
#pragma schedule[i+2]
    read = a[i];

#pragma schedule[N+2]
  result = read;

#pragma end_scop

  return 0;
}
```

A partir de ce code, un langage propre est déduit. Il se découpe en deux parties. Une première où les éléments sont décrit. Une liste de buffers avec leurs noms, leurs dimensions et leurs tailles est donné, ainsi que la liste des processus comprenant leurs nom, titre, et la liste de dépendances de buffers en entrée et sortie. La deuxième partie énumère la liste des tâches de chaque processus.
La première partie du simulateur correspond donc à un parser, qui traduit le fichier texte, la trace, pour créer les éléments qui composent le système. Ce parser utilise la librairie javacup et jflex. Il permet, à partir du langage de sortie, de créer l'ensemble des éléments du système.

Voici un exemple de trace :

```
Buffers
  a[3]
  read[1]
  result

Processes
  producer(Output a, "a[i] = i")
  consumer(Input a, Output read, "read = a[i]")
  resulter(Input read, Output result, "result = read")

Trace(producer)
  Write(a[0],0)
  Write(a[1],1)
  Write(a[2],2)
  Write(a[3],3)
  Write(a[4],4)
  Write(a[5],5)
  Write(a[6],6)
  Write(a[7],7)
  Write(a[8],8)
  Write(a[9],9)
  Write(a[10],10)
  Write(a[11],11)
  Write(a[12],12)
```

```
  Write(a[13],13)
  Write(a[14],14)
  Write(a[15],15)

Trace(consumer)
  FRead(a[0])
  FRead(a[1])
  FRead(a[2])
  FRead(a[3])
  FRead(a[4])
  FRead(a[5])
  FRead(a[6])
  FRead(a[7])
  FRead(a[8])
  FRead(a[9])
  FRead(a[10])
  FRead(a[11])
  FRead(a[12])
  FRead(a[13])
  FRead(a[14])
  FRead(a[15])
  Write(read[15],15)

Trace(resulter)
  FRead(read[15])
  Write(result,15)
```

3.4 Réalisation du simulateur

3.4.1 Les deux étapes du simulateur

Le simulateur s'articule en deux parties. La première crée les éléments qui composent le système (processus, buffers) à partir du language d'entrée et de construire une représentation intermédiaire. La deuxième constitue un historique des actions effectuées par les processus à chaque étapes.
Ces deux phases sont implémentées dans une même classe.

Prog La classe Prog correspond à la classe maîtresse de la partie simulation. Elle crée une représentation intermédiaire pour les CRP, où chaque élément est décrit puis effectue la simulation et sauve l'historique des actions menées par les buffers.

1. Attributs
   (a) int nbprocesses : Cet entier correspond au nombre de processus du système.
   (b) int nbbuffers : Cet entier correspond au nombre de buffers du système.
   (c) HashMap<String,ProcessCRP> assocProcess : La table de hachage permet de stocker, parcourir et retrouver les processus par leur nom.
   (d) HashMap<String,BufferCRP> assocBuffer : Tout comme la précédente, cette table de hachage permet de retrouver un buffer par son nom et sera indispensable pour appliquer les tâches (les instructions) des processus sur les buffers.
   (e) boolean verbose : Ce booléen active les impressions pour suivre les étapes de la simulation sur la sortie standard.
2. Méthodes
   (a) Le constructeur : la représentation intermédiaire est effectué dans le constructeur de cette classe. Il prend en paramètre les informations généré par le parser, à savoir la liste de Buffer avec nom et taille, la liste de processus et leurs listes d'instructions. Avec toutes ces données le constructeur remplit les tables de hachage et crée les éléments qui composent la représentation intermédiaire pour les CRP.
   (b) void run : cette méthode correspond à la deuxième étape de la simulation, elle génère l'historique des actions menées par les processus. L'historique est représenté par une structure, récupéré vide en paramètre.
   Cette structure est représenté par une liste de listes d'action. En effet pour une étape, soit un élément de la liste, une liste d'action effectué par l'ensemble des processus est créé.
   Problème : l'algorithme de génération de l'historique est basique et peut être amélioré. Une boucle principale (la boucle While) correspond aux étapes et initialise une nouvelle liste d'action à chaque tour. A chaque étape, la table de hachage d'association de processus est parcouru et les processus sont appelé à appliquer une étape et ainsi enrichir la liste d'action.

Nous verrons plus tard comment les processus enrichissent la liste d'action par leur méthode doStep(). Cependant un défaut important demeure, car les processus appliquent directement les instructions. Du fait, l'historique est dépendant de l'ordre dans lequel les processus sont appelé car lors d'une même étape, une même donnée peut être écrite et lue par deux processus différents. Chose qui dans l'ordre inverse nécessite deux étapes distinctes. Ainsi une amélioration nécessaire doit être effectué où la création des actions et leurs applications doivent être séparées.

Ces deux méthodes correspondent aux deux étapes de la simulation. La suite du document à donc pour but de détailler les éléments qui les composent.

3.4.2 Représentation intermédiaire pour les CRP

Suite à la traduction du language d'entrée par le parser, la classe Prog génère les éléments du système dont voici les détails :

Les buffers La classe BufferCRP est utilisée lors de la simulation ainsi que lors de l'affichage. Les attributs et méthodes décrit ici ne concerne que la partie simulation.
Le buffer virtuel, de dimension et de taille quelconque est remplacé par un buffer réel, un vecteur de taille adapté. L'accès aux cases du buffer est donc régi par une fonction bijective permettant de donner les coordonnées du vecteur virtuel pour lire dans le vecteur réel. Pour une liste d'entiers (noté coord) correspondant aux coordonnées dans le buffer virtuel, et la liste d'entier (noté dim) correspondant à la taille de ses dimensions on a le calcul de l'indice :

```
int indice=0;
int multi=1;
for (int i=0;i<sdim;i++)
   indice += (coord.get(i)\%this.dim.get(i))*multi;
   multi*=this.dim.get(i);
finfor
```

1. Attributs
   (a) String name : nom du buffer.
   (b) List<Integer> dim : liste d'entiers, chaque élément correspond à la taille d'une dimension du buffer virtuel.
   (c) int sdim : nombre de dimension du buffer.
   (d) int val[] vecteur contenant les valeurs du buffer. La taille de ce vecteur correspond au nombre d'élément du buffer virtuel.
   (e) boolean check[] : ce vecteur de booleens, de structure identique au
   (f) boolean check[] : ce vecteur de booleens, de structure identique au vecteur contenant les valeurs, restreint la disponibilité de la case associé en lecture et écriture. La valeur `false` correspond à une écriture autorisée et une lecture/écriture impossible.
   (g) int size : taille des tableaux check et val.
2. Méthodes
   (a) Le constructeur : il prend en paramètre le nom du buffer ainsi que la liste d'entier pour déterminer sa taille. Les variables (qui ont une liste d'entier nulle) est remplacé par un vecteur de taille 1. La taille des vecteurs est calculée comme le produit de la taille de chaque dimension du buffer virtuel et toute les cases du vecteur de booléen est initialisé à "false".
   (b) String getName : retourne le nom du buffer.
   (c) Boolean getCheck : prend en paramètre la liste d'entiers correspondant aux coordonnées dans le buffer virtuel et retourne le booléen de restriction correspondant.

(d) int getValue : de façon similaire à getCheck, prend en paramètre les coordonnées dans le buffer virtuel et retourne la valeur correspondante contenue dans le vecteur.

(e) int fgetValue : cette méthode retourne un entier identique à getValue, et modifie la case du vecteur de booléen en false. Ce qui interdit les lectures et autorise les écritures.

(f) void setValue : à l'inverse de la précédente, cette méthode prend en paramètre les coordonnées et un entier, et rempli la case correspondante dans le vecteur par celui ci. Puis modifie la case du vecteur de booléen en true, pour interdire les écritures et autoriser les lectures.

(g) void print : imprime sur la sortie standard le nom, les tailles et la dimension du vecteur virtuel.

Les processus La classe ProcessCRP est utilisée lors de la simulation ainsi que lors de l'affichage. Les attributs et méthodes décrits ici ne concernent que la partie simulation.

1. Attributs
   (a) String name : nom du processus.
   (b) String label : titre du processus, correspond à la ligne dans le code C d'origine.
   (c) List<InOut> list_inputs : liste des liens avec les buffers d'entrées. InOut est une classe permettant de dinsinguer les liens directs avec les multiplexer.
   (d) list<InOut> list_outputs : liste des liens avec les buffers de sortie.
   (e) list<InstructionCRP> thetrace : liste des instructions à effectuer par le processus, il existe différents type de lectures et écritures.
   (f) int current : entier correspondant à l'indice de la dernière instruction effectué lors de la simulation, il est initialisé à zéro.
2. Méthodes
   (a) Les constructeurs : deux constructeurs sont possibles, avec et sans la liste d'instruction en entrée. Dans les deux cas il prend en paramètre le nom, le titre, les listes de liens.
   (b) storeTrace : prend en paramètre une liste d'instructions, associe le nom du processus à chacune d'entre elle et l'associe à l'attique correspondant.
   (c) getName - getLabel - getIn - getOut : retourne l'attribut correspondant.
   (d) boolean doStep : la méthode prend en paramètre une liste d'Action et une table de hachage de buffer. Une Action est l'historique de l'application d'une tache d'un processus cette méthode est donc utilisé lors de la création de l'historique du système. La table de hachage permet de retrouver le buffer à partir de son nom.
   En fonction de la valeur de l'indice de l'instruction courante, la méthode crée une action. Si l'indice pointe vers la fin de la liste, l'action retourné est de type "End". Sinon elle tente d'appliquer la tache au buffer. La méthode ministep de la classe instruction est appelé, si l'instruction n'est pas effectué (en fonction de la restriction en lecture/ecriture) elle renvoi une action de type "Wait", sinon elle renvoi une action de même type que l'instruction.
   Chaque action créé est ajouté à la liste en entrée. Les instructions sont toute effectuées à la suite jusqu'à la création d'une action de type "End", "Wait" ou "Write". La méthode renvoi également un booléen (sleep) de valeur true si aucune instruction n'a été appliqué. A savoir si aucune action est de type lecture ou écriture.
   (e) void print : imprime le nom, le titre et la liste des liens (d'entrée et sortie) du processus.

InOut La classe InOut représente un ou des liens entre un processus et un ou des buffers. Si un InOut ne possède qu'un lien avec un buffer, il est dit simple, sinon il représente un multiplexeur.

1. Attributs
   (a) int tag : entier correspondant au type (simple ou alternatif) de l'élément InOut.
   (b) List<String> wires : liste des noms des buffers liés.

2. Méthodes
   (a) Le Constructeur : prend en paramètre les deux attributs.
   (b) List<String> getListWires : retourne la liste de nom de buffer.
   (c) int getSize : retourne le nombre de liens.
   (d) int getTag : retourne la valeur du tag.
   (e) String getTheName : prend en paramètre l'indice du nom du buffer à retourner.
   (f) String getOneName : retourne le premier nom de buffer de la liste.

InstructionCRP Cette classe abstraite représente une unique instruction effectué par un process sur un buffer. Les instructions peuvent être de deux types : une lecture (ReadInstructionCRP) ou écriture (WriteInstructionCRP). Une écriture libère la case du buffer en lecture et interdit l'écriture. A l'inverse il existe deux cas de lecture. Une lecture de type FRead libère la case en écriture et verrouille en lecture, alors qu'une lecture de type Read ne modifie pas les restrictions.

1. Attributs
   (a) String caller : nom du processus dont dépend l'instruction.
   (b) String arrayname : nom du buffer où sera appliqué l'instruction.
   (c) List<Interger> index : coordonnées du buffer où sera appliqué l'instruction.
   (d) Integer value (Write) : valeur à écrire dans le buffer lors de l'écriture.
   (e) Boolean isFR (Read) : définit le type de lecture, Read ou FRead.
2. Méthodes
   (a) le constructeur : Prend en paramètre l'ensemble des attributs de la classe pour les mémoriser.
   (b) void putCaller : associe l'instruction au processus.
   (c) print : imprime les propriétés de l'instruction.
   (d) Action ministep : cette méthode a pour but de tenter d'appliquer l'instruction. Elle prend en paramètre la table de hachage de buffer pour le retrouver à partir de son nom.
   En fonction du type de l'instruction, et la restriction du buffer, elle renvoie une action de type "Wait" si non réussi, ou du même type sinon.

Une fois l'ensemble des éléments qui composent le système créés, la partie simulation est commencée. Au cours de cette simulation, un historique du résultat de l'application des instructions est généré.

3.4.3 Création de l'historique

L'historique est représenté par la classe HistoCRP contenant des éléments de type Action. Ces deux classent contiennent toutes les informations nécessaire par la suite pour la visualtion de ces étapes.

Action Une action est généré par les processus lors de l'application d'une unique instruction. Cette action mémorise le résultat de cette application et les variation qu'elle entraîne.

1. Attributs
   (a) String type : Il y a cinq types d'actions différent. Lorsque le processus parvient à appliquer une instruction, les types sont équivalent (Read, FRead, Write). A ces trois types d'instructions, deux autres sont ajoutés. Le type End traduit l'inactivité d'un processus qui a déjà appliqué la totalité de ses instructions. Le type Wait quand à lui, est créé quand une instruction n'a pu être effectuée, l'instruction sera donc rappelé à l'étape suivante.
   Remarque : le type Wait et End sont tout deux synonymes d'inactivité du processus
   (b) String caller : le nom du processus qui a effectué cette action.
   (c) String arrayname : le nom du buffer où est appliquée l'action.
   (d) List<Integer> index : les coordonnées de la case où est appliqué l'action.
   (e) int valueIn : la valeur de la case du buffer avant l'application de l'action.

   (f) int valueOut : la valeur de la case après l'application de l'action.

2. Les Méthodes
   (a) Les constructeurs : deux cas sont possibles. Dans le cas de l'application réussie d'une instruction, l'ensemble des attributs est rempli. Dans le cas d'une action End ou Wait, seul le type et le nom du processus sont nécessaires.
   (b) void print : cette méthode imprime les informations disponibles de l'action.
   (c) les méthodes "get" : l'ensemble des attributs possèdent une méthode permettant de récupérer.

HistoCRP La classe HistoCRP propose les outils nécessaires à l'organisation et l'exploitation des actions effectuées lors de la simulation.

1. Attributs
   (a) List<List<Action» histo : l'unique attribut de cette classe est une structure où toutes les actions de toutes les étapes sont stockées. Une liste d'action est la liste des actions effectuées par tout les processus pendant une étape. Ainsi l'ensemble des étapes forme une liste de listes d'action.
2. Méthodes
   (a) Le constructeur : Initialise la structure "liste de liste" d'action.
   (b) void print : imprime les actions par étapes pour visualiser l'historique.
   (c) int getSize : retourne le nombre d'étapes de l'historique.
   (d) List<Action getList : retourne la liste d'actions pour l'étape d'indice donné en paramètre.

Une fois la simulation terminée et l'historique généré, toutes les données nécessaires à l'affichage sont disponibles.

4 Interface graphique de simulation

4.1 Objectif

La deuxième partie de l'outil est la visualisation du système dans une interface graphique. L'objectif est de représenter l'ensemble des buffers, processus ainsi que leurs actions pour toutes les étapes. Pour cela, l'interface exploite l'historique généré par le simulateur. Ainsi les informations visibles sont directement issu de la partie précédente.
Le principal défis de la conception de l'interface est la visualisation des buffers avec les éléments qui les composent. En effet lors de l'écriture d'une donnée dans un buffer, il faut être capable de représenter cette cellule en lecture possible. Or l'historique ne permet de visualiser que les actions des processus, pas l'état courant des buffers. Ainsi la problématique sera de gérer lors de l'avancée ou le recul dans les étapes, l'état de chacun des éléments des buffers.

4.2 Choix des technologies

Afin de réaliser l'interface homme machine (IHM), on a besoin de bibliothèques Java permettant de dessiner des formes simples comme des traits, des rectangles. On a également besoin d'insérer des images, du texte et des boutons permettant à l'utilisateur d'interagir avec le programme.
Les bibliothèques utilisées sont javax.swing et java.awt, la première est une bibliothèque de dessin. Elle permet de généré une fenêtre, le conteneur, et de dessiner les formes souhaitées ou incruster des images. La seconde propose des objets pour interagir et communiquer avec un utilisateur tel que les boutons.

Fenêtre Cette classe est la fenêtre de l'interface, elle est hérité de la classe JFrame de la bibliothèque javax.swing. Elle est découpé en plusieurs sous fenêtres qui contiennent les dessins et les boutons.

1. Attributs
   (a) HistoCRP histo : afin d'exploiter l'historique des étapes générées lors de la simulation elle sera directement mémorisé en temps qu'attribut de cette classe. La classe HistoCRP est présentée dans la partie précédente.
   (b) int curStep : cet entier correspond au numéro de l'étape courante et est incrémenté ou décrémenté en fonction des consignes de l'utilisateur. Il est borné entre 0 et le nombre d'étape. Et est initialisé à -1 avant le premier dessin.
   (c) Prog prog : la classe Prog est réutilisé après la simulation afin d'atteindre les éléments qui composent le système (process, buffer) ainsi que leurs tables de hachage.
   (d) Panneau pan : le système composé des processus et buffers sont dessinés dans ce panneau.
   (e) Button next, previous, play, stop : ces quatre boutons servent à se déplacer dans les étapes de la simulations. Ils sont les seuls éléments qui permettent à l'utilisateur d'interagir avec l'interface.
   (f) JPanel container : cette sous fenêtre contient le panneau «pan» dans lequel sera dessiné le système.
   (g) Jpanel buttonMenu, buttonBoxFake, buttonBox : Ces éléments permettent de dessiner les boutons de l'interface. le buttonMenu, une sous partie de la fenêtre, est divisé en trois zones et prend toute la largeur de la fenêtre. les deux buttonBoxFake (0 et 1) servent à forcer le dimentionnement des buttons pour le tiers de la cette largeur. Ainsi buttonBox est rempli des quatre boutons. Visiblement seul buttonMenu et ses quatre boutons sont dessinés.
   (h) boolean PlayB : ce booléen correspond à l'activation du mode auto, où les étapes défilent sans l'intervention de l'utilisateur.
2. Méthodes
   (a) Le constructeur : il associe à chaque attributs les éléments de la simulation. Il fixe le nom de la fenêtre, sa taille, son type.
   Une foi les attributs initialisés, il créé le panneau contenant le système et dessine les boutons.
   (b) void plusStep : cette méthode est appelé par le bouton next, elle permet de visualiser l'étape suivante. Si une étape suivante existe, cette méthode incrémente le compteur curStep, récupère la liste des actions à effectué. Cette liste est alors donné en paramètre à la méthode apply du panneau pour appliquer cette étape et l'afficher.
   (c) void minusStep : le bouton previous permet d'appeler cette méthode. Elle permet de revenir à l'étape précédente.
   Pour y parvenir, il ne suffit pas

Panneau

1. Attributs
   (a) HashMap<String,ProcessCRP> assocProcess : table de hachage contenant tout les process stockés suivant leur nom.
   (b) HashMap<String,BufferCRP> assocBuffer : table de hachage contenant les buffers, stockés suivant leur nom.
   (c) List<List<ProcessCRP» axesProc : Chaque liste de processus correspond à la liste des processus affiché sur un seul et même axe vertical. Une liste de liste de processus correspond à la liste des axes classé du plus à gauche au plus à droite.
   (d) List<List<BufferCRP» axesBuff : Comme l'attribut précédent, cette liste classe les buffers par axes verticaux du plus à gauche au plus à droite.
   (e) List<MuxCRP> listMux : Liste de tous les multiplexeurs.

   (f) List<Line> listLine : liste des lignes, une ligne représente l'objet graphique qui illustre une action d'un processus (lecture, écriture).

2. Méthodes
   (a) Le constructeur : stocke les deux tables de hachage.
   (b) void printAxes : imprime les axes de processus et buffer.
   (c) void ConstructAxis : construit les axes verticaux de processus et buffers. Il créé également une liste de multiplexeurs. Le principe est de déterminer la position de chaque processus et buffer sur l'interface. Pour cela il créé le premier axe de processus. Pour cela il parcours la table de processus et récupère tout les processus sans buffer d'entrée. Puis il construit le premier axe de buffer constitué de toute les buffers en sortie des processus de l'axe précédent. Finalement il crée un second axe de processus constitué de ceux qui n'ont pas encore été "dessinés" et qui ont pour entrée les buffers de l'axe précédent. Et ainsi de suite jusqu'à ce qu'aucun processus ne peut être ajouté lors d'un passage.
   (d) void setPosition : cette fonction permet de définir pour chaque élément des axes, leurs coordonnées dans l'interface (en pixel).
   (e) void initialise : initialise l'objet, lance la construction des axes et la définitions des coordonnées.
   (f) void apply : avance d'une étape, dessine les actions liées
   (g) void back : recule de deux étapes, puis avance d'une. Ces actions sont nécessaire pour retrouver l'état "orange" de nouvelle lecture.
   (h) void drawLink : dessine toute les lignes des actions de l'étape courante.
   (i) void paintComponent : dessine le système, composé de buffers, process et multiplexers.

4.3 Copies d'écran

Les Figures 1 et 2 montrent notre outil en cours d'exécution. Les processus sont représentés par les blocs noirs, et les buffers par les blocs rouges. En cours d'exécution, les lectures et écritures sont "animées" par l'apparition de traits verts et rouges entre les processus et les buffers.

FIGURE 1 – Simulation en cours d'exécution sur l'exemple producteur/consommateur simple

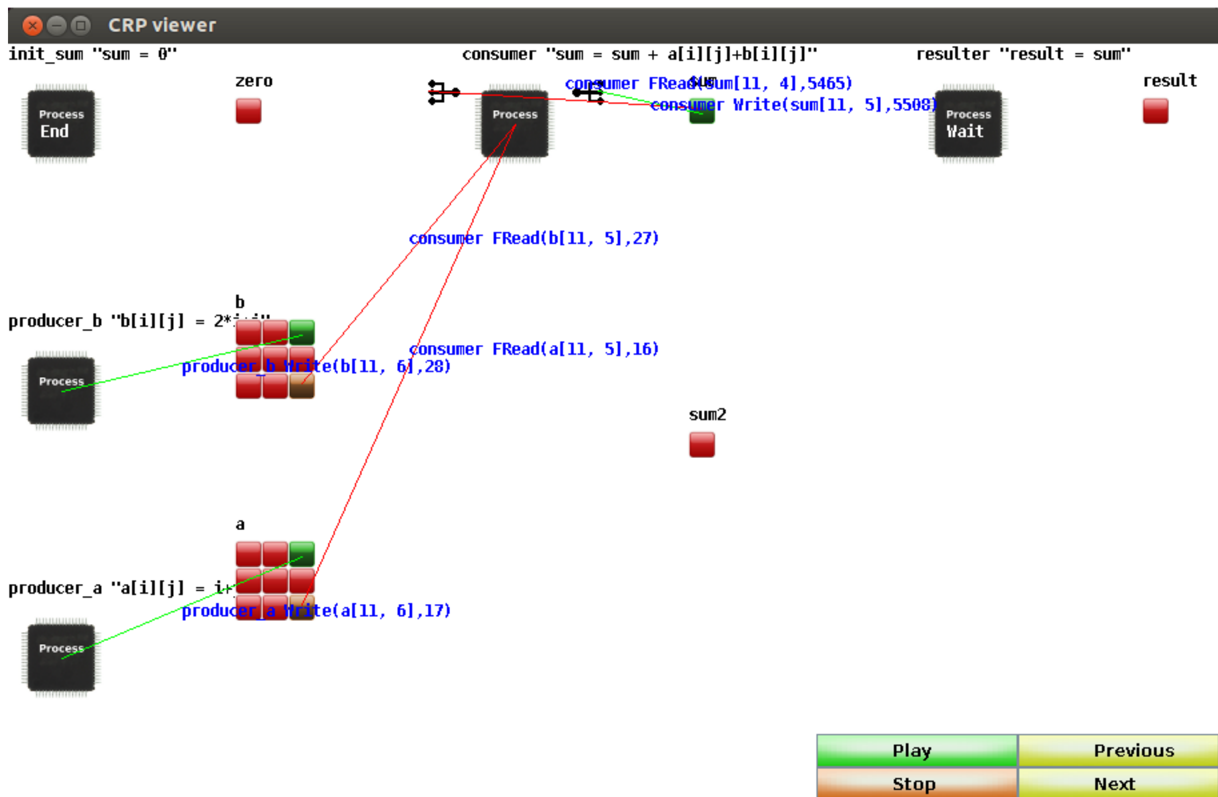


FIGURE 2 – Simulation en cours d'exécution sur l'exemple producteur/consommateur simple

5 Conclusion

L'interface graphique de simulation est fonctionnelle et permet de visualiser les exécutions des processus.

Références

[1] Paul Feautrier. Scalable and structured scheduling. *International Journal of Parallel Programming*, 34(5) :459–487, October 2006.