Laboratoire de Systèmes Logiques

Simulateur de l'automate de von Neumann en langage JAVA

Edouard FORLER

Assistants
Jean-Luc Beuchat
Jacques-Olivier Haenni

Cours de Systèmes et Programmation Génétiques
Professeur Daniel Mange

Lausanne, juillet 1998

Chapitre 1

Introduction

Imaginer, c'est hausser le réel d'un ton.
– Bachelard.

1.1 Présentation du document

Le présent document regroupe l'ensemble des informations relatives à la conception et au développement du simulateur des biodules 602 en Java. On y trouvera :

- une présentation succincte du biodule 602 ;
- les pistes retenues pour le développement du simulateur ;
- les détails de l'implémentation ;
- une présentation de quelques extensions possibles.

1.2 But du projet

Dans le cadre du cours de *Systèmes et programmation génétiques* donné par le Professeur Daniel Mange au Département d'Informatique de l'EPFL, il a été jugé intéressant de pouvoir disposer d'un simulateur reproduisant le comportement de réseaux de biodules 602. Ce simulateur sera utilisé par les étudiants pour réaliser des expériences basées sur les règles de l'automate cellulaire de John von Neumann.

Afin de pouvoir le faire fonctionner sur un nombre important de plateformes et de répondre à certains critères de réutilisabilité, il est nécessaire que le simulateur soit écrit en langage JAVA. Il doit pouvoir au minimum fournir les opérations habituelles pour ce type de système (évolution automatique du système, pas à pas, arrêt sur une configuration intéressante, etc.). Aucune autre restriction n'est imposée quant aux fonctionnalités.

La réalisation pratique du mini-projet peut être consultée sur le *World Wide Web* à l'adresse `http://lslwww.epfl.ch/~eforler/neumann.html`.

Chapitre 2

Le biodule 602

La Nature agit, l'Homme fait.
– Kant.

2.1 L'automate de von Neumann

D'après [3, Chap. 2]

L'automate cellulaire de von Neumann est un automate dit universel dans le sens qu'il peut simuler une machine de Turing. Il est également dit auto-reproducteur, car il est capable de reproduire le même schéma sur une grille de cellules. Il existe d'ailleurs d'autres automates auto-reproducteurs, tel celui de Langton [5], mais qui n'ont pas la puissance de celui de von Neumann. Pour observer de telles caractéristiques, il faut toutefois un réseau de cellules gigantesque (environ 200'000 cellules selons certaines estimations), ce qui en fait un automate purement théorique.

L'automate de von Neumann est un automate synchrone bidimensionnel à 29 états, utilisant le voisinage dit de von Neumann, c'est-à-dire la cellule elle-même et ses quatre voisines. Les 29 états possibles sont les suivants :

- huit états de transmission ordinaires (4 états "excités" et 4 états "non excités") ;
- huit états de transmission spéciaux (4 états "excités" et 4 états "non excités") ;
- quatre états appelés confluents ;
- l'état quiescent ;
- huit états sensitifs.

Cet automate a fait l'objet de nombreuses études logicielles (simulation, tentative de simplification, etc.) depuis son élaboration, mais la première réalisation matérielle (logique câblée) ne date que de 1995. Elle est le fruit du travail de Jean-Luc Beuchat et Jacques-Olivier Haenni, du Laboratoire de Systèmes Logiques de l'EPFL.

Nous présentons dans ce chapitre les principales caractéristiques de l'automate et de sa réalisation matérielle. Le lecteur intéressé pourra trouver les détails de fonctionnement de l'automate de von Neumann dans [1] et la spécification technique du biodule 602 dans [2] et [3].

2.2 Etats et règle de transition détaillés

Extrait de [2, Chap. 2]

Etats		Symboles
Etat quiescent		U
Etats sensitifs		S_θ, S_0, S_1, S_{00}, S_{01} S_{10}, S_{11} et S_{000}
Etats de transmission ordinaires	non excités	$\uparrow$, $\downarrow$, $\leftarrow$ et $\rightarrow$
	excités	$\uparrow\cdot$, $\downarrow\cdot$, $\dot{\leftarrow}$ et $\dot{\rightarrow}$
Etats de transmission spéciaux	non excités	$\Uparrow$, $\Downarrow$, $\Leftarrow$ et $\Rightarrow$
	excités	$\Uparrow\cdot$, $\Downarrow\cdot$, $\dot{\Leftarrow}$ et $\dot{\Rightarrow}$
Confluents	non excités	C_{00}
	excités	C_{01}, C_{10} et C_{11}

TAB. 2.1: Les 29 états

2.2.1 L'état quiescent

Cet état, noté U, correspond à une cellule morte, n'ayant aucune influence sur ses voisines.

2.2.2 Les états de transmission ordinaires

Les états de transmission ordinaires permettent l'acheminement d'information (excitations) d'un point de l'automate à un autre. Ils sont symbolisés par une flèche indiquant la direction dans laquelle ils transmettent les excitations (nord, sud, est ou ouest). La face vers laquelle pointe la flèche constitue la sortie de la cellule (les trois autres faces sont les entrées). Un tel état est soit :

- excité ; nous symboliserons l'excitation par un point à côté de la flèche ;
- non excité.

Il existe ainsi huit états de transmission ordinaires.

Lorsqu'il reçoit une excitation ordinaire (ou provenant d'un confluent) sur une ou plusieurs de ses entrées, un état de transmission ordinaire devient (ou reste) excité. Un tel état se comporte ainsi comme une porte logique OU (figure 2.1).

Un état de transmission ordinaire devient (ou reste) excité lorsqu'il reçoit, sur au moins une de ses entrées, une excitation provenant

- d'un état de transmission ordinaire ou
- d'un état confluent.

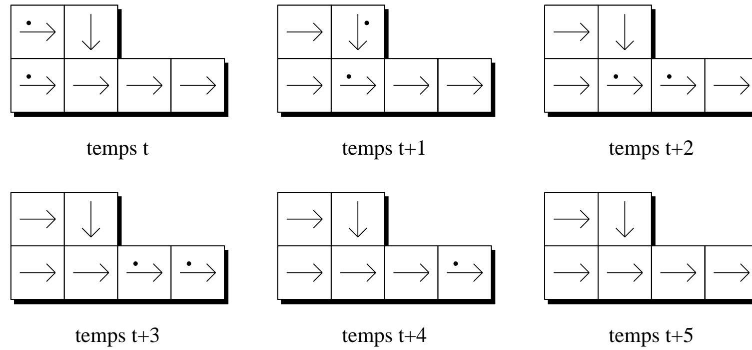


FIG. 2.1: Etats de transmission ordinaires

2.2.3 Les états de transmission spéciaux

Tout comme les états de transmission ordinaires, les états de transmission spéciaux permettent l'acheminement d'information[1] entre deux points de l'automate et se comportent comme une porte logique OU. Une cellule spéciale devient (ou reste) excitée à l'instant $t+1$ si elle reçoit une excitation provenant d'un état de transmission spécial ou d'un confluent sur une ou plusieurs de ses entrées à l'instant t.

2.2.4 Les états confluents

Les états confluents $C_{b_1 b_0}$ $(b_1, b_0 \in \{0,1\})$ remplissent quatre fonctions :

1. **Introduction d'un double délai.**
   Un état confluent se comporte comme un registre à décalage de deux bits :
   - $b_1(t+1) = b_0(t)$
   - $b_0(t+1)$ dépend de l'état des cellules voisines au temps t

   L'état d'excitation vu par les cellules voisines est $b_1(t)$.
2. **Réalisation de la fonction logique ET.**
   $b_0(t+1) = 1$ si et seulement si
   - il y a au moins une cellule voisine dans un état de transmission ordinaire excité dirigé vers $C_{b_1 b_0}$ et
   - toutes les cellules voisines dans un état de transmission ordinaire dirigé vers $C_{b_1 b_0}$ sont excitées au temps t.

[1]Nous constaterons par la suite que cette information diffère de celle transitant à travers des états de transmission ordinaires.

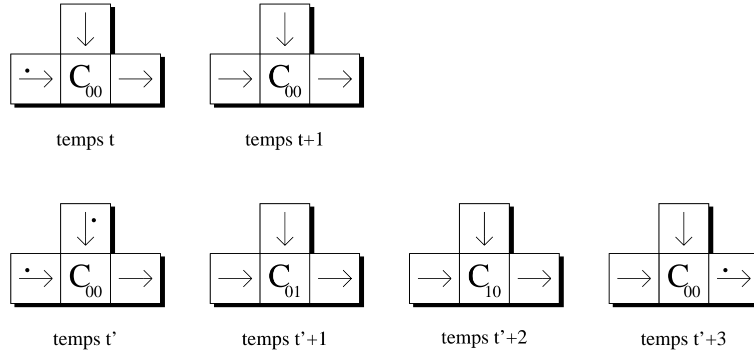


FIG. 2.2: Les états confluents se comportent comme une porte logique ET

3. **Conversion d'une excitation ordinaire en excitation spéciale.**
   Bien qu'elle ne puisse être excitée que par des états de transmission ordinaires, une cellule $C_{b_1 b_0}$ transmet son excitation à des états de transmission ordinaires ou spéciaux.

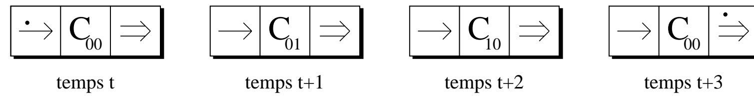


FIG. 2.3: Conversion d'une excitation ordinaire en une excitation spéciale à l'aide d'un confluent

4. ***Fan-out.***
   Un état confluent émet de l'information dans quatre directions (nord, sud, est et ouest). Il est ainsi possible de multiplier un signal en connectant plusieurs états de transmission ordinaires ou spéciaux à un confluent (figure 2.4).

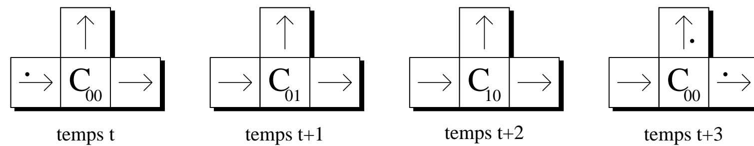


FIG. 2.4: Un état confluent se comporte comme un *fan-out*

2.2.5 Les états sensitifs

Les huit états sensitifs sont des états transitoires permettant le passage de l'état U à $\leftarrow$, $\rightarrow$, $\uparrow$, $\downarrow$, $\Leftarrow$, $\Rightarrow$, $\Uparrow$, $\Downarrow$ ou C_{00}. Lorsqu'un état quiescent reçoit une excitation ordinaire ou spéciale, il passe dans l'état S_θ. A chaque coup d'horloge, l'état évolue selon le graphe de la figure 2.5.

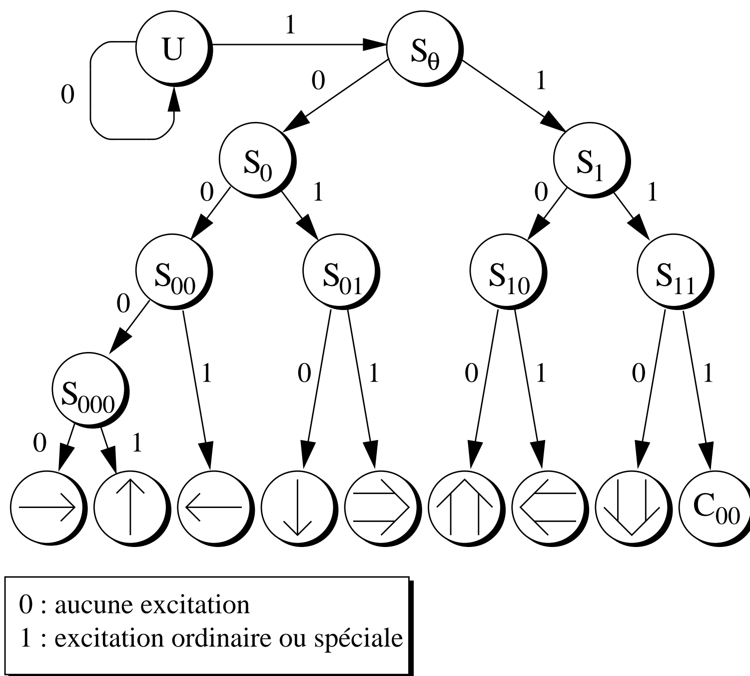


FIG. 2.5: Les états sensitifs

2.2.6 Destruction d'une cellule

La destruction constitue une transition d'un état de transmission ou confluent vers l'état quiescent. Ce processus survient

- lorsqu'un état de transmission ordinaire ou un confluent reçoit une excitation spéciale sur l'un de ses côtés ou
- lorsqu'un état de transmission spécial reçoit une excitation ordinaire sur l'un de ses côtés.

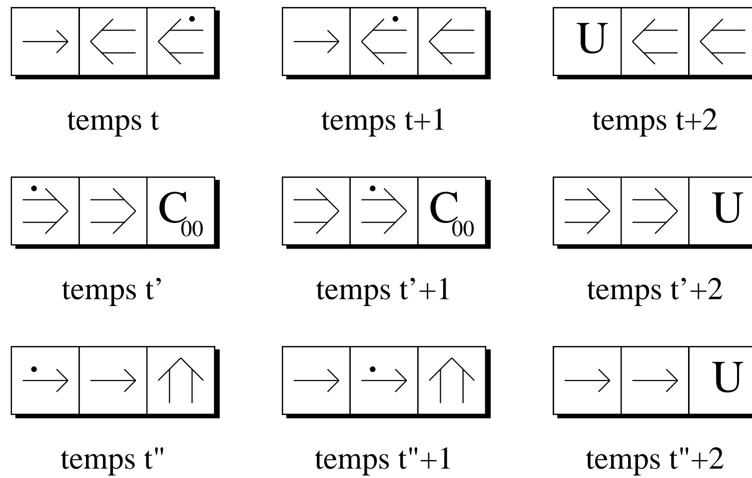


FIG. 2.6: Exemples de destruction de cellules

2.3 Réalisation câblée

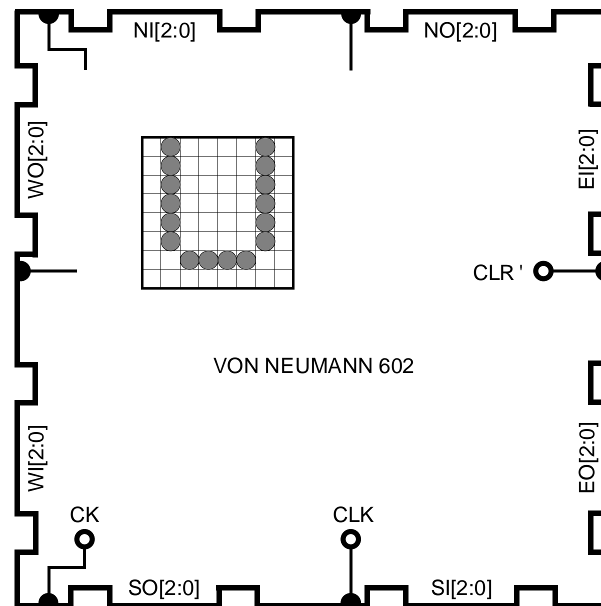


Fig. 2.7: Le biodule 602 dit “de von Neumann”

Les biodules 602 se présentent sous la forme d’un boîtier de 8x8x4cm. Pour réaliser un mini-réseau cellulaire il suffit de les assembler. Un biodule est formé de deux modules logiques indépendants :

- **la partie de calcul :** qui mémorise l’état courant de la cellule, communique avec les cellules voisines et calcule son état futur ;
- **le module d’affichage :** qui reçoit de la partie de calcul cinq bits codant l’état courant et s’occupe de le représenter sur un affichage à LED de 8x8 points.

Il est intéressant de remarquer que la logique de calcul est purement combinatoire ; les transitions sont donc calculées en un seul coup d’horloge.

Le simulateur se contentera d’implémenter le comportement visible des biodules (en cela, il ne s’agit donc que d’un simulateur de l’automate de von Neumann), avec une petite caractéristique supplémentaire. En effet, dans son raisonnement initial, John von Neumann se base sur le postulat que l’automate évolue dans sur un tissu composé de cellules à l’état quiescent, dans un espace infini. Ceci n’est pas réalisable physiquement. De plus, pour économiser le matériel, certains biodules n’évoluant pas ne sont pas assemblés. Nous faisons donc une distinction entre cellule à l’état quiescent (cellule présente physiquement dans le montage, mais inutilisée) et cellule “vide”. La cellule “vide” ne réagit jamais quel que soit le comportement de ses voisines ; c’est en fait l’environnement du montage biodule.

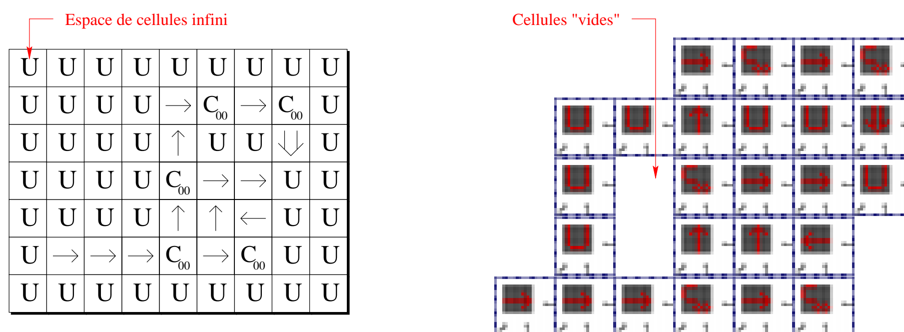


FIG. 2.8: Représentation théorique et réalisation pratique

Chapitre 3

Développement

Il n'est pas difficile d'avoir une idée. Le difficile, c'est de les avoir toutes.
– Alain.

3.1 Considérations préléminaires

Comme mentionné dans l'introduction, le développement du simulateur a été effectué en JAVA, car le but principal du mini-projet est de fournir l'application aux étudiants sous la forme d'une *applet* disponible par le biais du *World Wide Web*.

Toutefois, JAVA est un langage qui évolue rapidement. Notre choix s'est porté sur la version 1.1 du langage. Son principal avantage, outre la correction de nombreux défauts présents dans la version 1.0, est de fournir une gestion robuste des événements, ce qui simplifie considérablement le développement de l'interface utilisateur.

JAVA 1.1 n'est, dans l'état actuel des choses, pratiquement pas supporté par les principaux navigateurs. Mais cette version du langage est naturellement appelée à s'imposer dans l'avenir proche, étant donnés ses avantages. Il vaut donc mieux implémenter directement l'application en JAVA 1.1, plutôt qu'en JAVA 1.0, pour devoir ensuite en réécrire une grande partie. Nous utilisons pour visualiser l'*applet* le navigateur *Netscape Navigator 4.04*, compatible avec JAVA 1.1.2.

L'implantation sous la forme d'une *applet* pose quelques problèmes supplémentaires : certaines possibilités du langage JAVA nous sont interdites. Ceci est imposé pour des raisons de sécurité. Tout ce qui concerne la manipulation de fichiers, en particulier, est interdit dans une *applet*. Nous devrons donc dans un premier temps renoncer à pouvoir charger et sauvegarder des configurations de biodules, ce qui est un handicap majeur pour ce type d'application. Nous verrons toutefois qu'il est possible de contourner le problème.

3.2 Présentation des fonctionnalités

Il faut développer les fonctions usuelles des simulateurs de systèmes synchrones, à savoir :

- la simulation automatique continue d'une configuration ;
- la simulation pas-à-pas pour permettre l'observation des transitions intéressantes ;
- le retour à l'état initial et l'édition de la configuration.

A partir de ces trois fonctions de base, on conçoit aisément quelques extensions utiles :

- la possibilité de modifier l'état de la configuration pendant la simulation, de manière à observer les effets engendrés ;
- la possibilité d'arrêter la simulation sur un état précis (point d'arrêt) ou à un coup d'horloge précis.

Enfin, certaines caractéristiques particulièrement intéressantes de l'automate cellulaire de von Neumann, concernant sa construction et la réplication, nous conduisent à ajouter trois fonctions de plus :

- l'introduction d'une séquence d'excitations dans la configuration, c'est-à-dire d'une série d'impulsions normales ou spéciales auxquelles la configuration va réagir ;
- la récupération d'une séquence d'excitations générée par l'automate ;
- la génération de la séquence d'excitations de construction d'une configuration.

Muni de ces "entrées/sorties", l'automate devient capable de traiter de l'information et de produire un résultat.

3.2.1 Fonctions d'édition

Elles se résument essentiellement à une boîte à outils contenant l'ensemble des états valides pour un biodule 602. Pour construire un automate, il suffit de sélectionner les états nécessaires (confluents, transmissions...) et de les placer sur la grille d'édition (voir paragraphe 3.3). Un outil "gomme" sert à retirer des logidules du système.

Trois fonctions supplémentaires seront implémentées :

- un outil "propriétés" permettant d'obtenir des informations sur une cellule, et le cas échéant, d'en modifier certaines ;
- une réinitialisation du simulateur, c'est-à-dire la possibilité de reprendre l'édition à zéro ;
- la manipulation de fichiers, mais uniquement le chargement.

3.2.2 Fonctions de simulation

Elles servent à étudier le comportement d'un réseau et agissent donc au niveau global. Une fois l'édition d'un réseau réalisée, il devra être possible d'en effectuer la simulation ; le résultat doit être visible en temps réel (pour des vitesses raisonnables, de l'ordre de la dizaine de Hertz, compte tenu de la lenteur relative du langage JAVA en *applet*), tout en conservant l'interaction avec l'utilisateur.

Pendant la simulation, l'utilisateur aura la possibilité de faire varier la vitesse de la simulation ou même de passer en mode "pas-à-pas", dans lequel il commande directement l'évolution du système. La fonction "pas-à-pas" sera du reste être disponible à tout moment, pour effectuer un pas unique de simulation.

Naturellement, le réseau à étudier peut être complexe. Certaines configurations, pendant leur évolution, possèdent des transitions intéressantes à étudier en détail, à un moment précis. Il est donc utile de pouvoir stopper la simulation ou entrer en mode "pas-à-pas" suite à un passage dans un état particulier ou à un moment particulier. On rajoutera donc un ensemble

de fonctions destinées à la manipulation de points d'arrêts dans le mécanisme de simulation ; plus précisément, une option permettra à l'utilisateur de faire fonctionner le système depuis le temps zéro (état initial) jusqu'à un temps t spécifié. Un des intérêts de ce mécanisme est d'autoriser un retour rapide vers un état qui avait été atteint dans une simulation antérieure. Cette fonctionnalité devra être désactivable si l'utilisateur souhaite poursuivre une simulation jusqu'à son terme.

Un autre type de point d'arrêt très utile est celui déterminé par l'état d'une cellule du système. Il sera dans cette optique possible de placer un point d'arrêt sur chacune des cellules du réseau, par exemple "état confluent C_{10} atteint". Très exactement, il sera possible de stopper l'ensemble de la simulation lorsqu'une cellule aura atteint un état précis.

Nous avons parlé plusieurs fois de temps zéro de la simulation ; étant donné qu'il est permis d'intervenir avec les outils d'édition sur n'importe quelle configuration du système (voir paragraphe 3.3), il n'est pas judicieux d'imposer le temps zéro de la simulation de manière arbitraire : c'est beaucoup trop restrictif (ex. : au moment du chargement de la configuration initiale ; quid des modifications réalisées entre temps ?). Une fonction supplémentaire permettra donc à l'utilisateur de fixer le moment zéro à sa guise, plusieurs fois de suite si nécessaire, et d'y revenir.

3.2.3 Manipulation de séquences d'excitations

Dans son ouvrage sur les automates universels [1], John von Neumann décrit les organes nécessaires à la construction d'un automate. Il décrit en particulier deux types de "bras constructeurs" (simple et double) utilisés pour le placement des cellules de l'automate dans l'espace de construction.

Pour la réalisation physique de l'automate, la solution retenue est le bras simple. Jean-Luc Beuchat et Jacques-Olivier Haenni [2][3] ont conçu un système séquentiel qui calcule la séquence d'excitations à introduire à l'une des extrémités du réseau pour reconstruire un automate complet (ou tout du moins un organe complet), selon le principe du bras constructeur de von Neumann.

Le simulateur doit pouvoir reproduire le mécanisme de construction utilisé en pratique. En fournissant une séquence adéquate au simulateur et un "point d'introduction" dans le réseau, on doit pouvoir étudier le comportement du bras constructeur avec les outils de simulation décrits plus haut.

Le même principe est applicable à la transmission de séquences de données à un automate existant, de manière à ce qu'il les traite ad hoc et fournisse un résultat. Ce résultat pourra à son tour être récupéré au niveau de ce que nous appelerons des "points d'extraction" (fonctionnement inverse du point d'introduction).

Enfin, puisqu'il est possible de reconstituer un automate complet à partir d'une séquence d'excitations, il doit être possible de produire la séquence correspondant à un réseau existant. Ceci a été réalisé avec l'aide de Jean-Luc Beuchat et Jacques-Olivier Haenni. Remarquons que de telles séquences peuvent atteindre des longueurs considérables.

Le simulateur nous permet alors d'étudier les réseaux de biodules 602 aussi bien du point de vue "microscopique" (fonctionnement des cellules), que du point de vue "macroscopique" (fonctionnement d'un organe tel que décodeur ou pulseur).

3.3 L'interface utilisateur : critères

L'affichage du réseau de biodules se fait dans le navigateur lui-même sous la forme d'une grille dans laquelle on peut placer et retirer les biodules. Etant donné le nombre relativement petit de fonctionnalités offertes, il nous a paru intéressant de donner à l'utilisateur la possibilité d'accéder à la majorité de ces fonctionnalités directement, sans passer par des menus. L'utilisateur va donc se retrouver aux commandes du simulateur avec un "tableau de bord" complet et directement accessible.

Le problème principal qui se pose est de faire la différence entre la partie "édition" du programme et la partie "simulation". Il est trop fastidieux de séparer complètement les deux parties pour l'utilisateur, car elles sont beaucoup plus liées qu'elles ne le semblent de prime abord : en effet, il est courant d'éditer un petit bout de réseau, puis de le tester, puis de refaire quelques modifications, et ainsi de suite. De plus, modifier le comportement de l'automate en cours de route, par exemple pour tester l'influence d'une excitation sur l'ensemble du système, revient en fait à éditer la configuration du réseau.

Nous avons donc choisi de proposer à l'utilisateur une interface permettant à la fois d'éditer et de simuler le réseau. Toutefois, il ne sera pas possible de modifier l'état d'une cellule en même temps que l'on simule le comportement de l'ensemble. Mais il est tout à fait possible, sans passer d'un mode "simulation" à un mode "édition" de modifier une cellule : il suffit d'arrêter, même momentanément (mode pas-à-pas), la simulation en cours, de faire la modification voulue, et de relancer la simulation, tout cela dans la même interface.

Trois aspects sont à étudier pour l'interface :

- la vision globale du système ;
- l'édition de la configuration ;
- le contrôle de la simulation.

3.3.1 Vision au niveau global

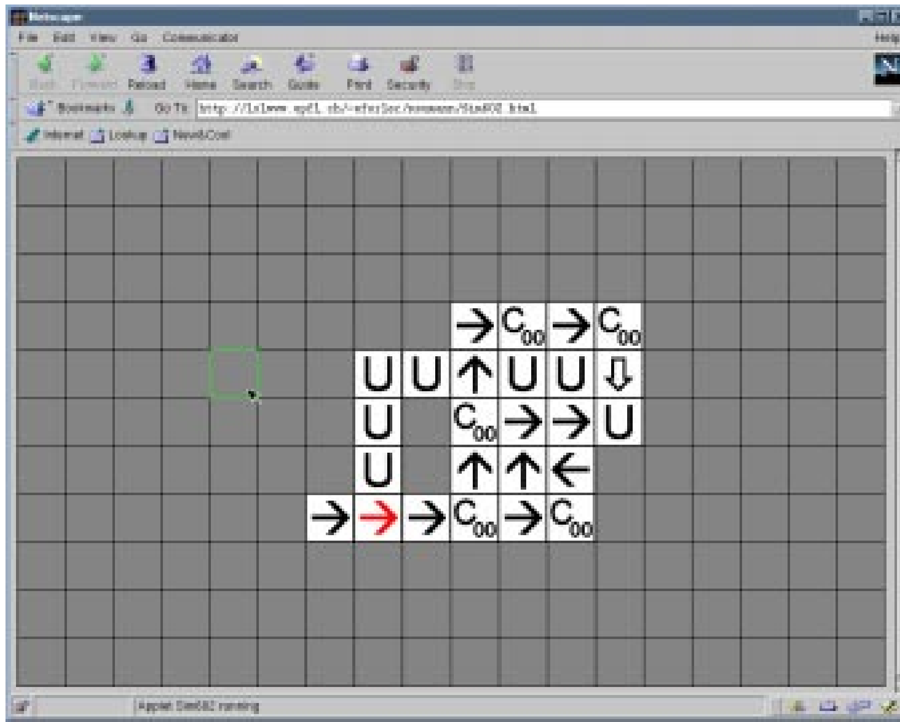


FIG. 3.1: Aspect général de l'interface

La fenêtre du navigateur ne montre que le réseau de biodules. Le nombre de biodules total peut être beaucoup plus grand que le nombre de biodules effectivement visibles dans le navigateur. La grille d'affichage s'adapte automatiquement à la taille de la fenêtre. Pour résoudre le problème de visibilité, un système de navigation a été rajouté dans la fenêtre de contrôle de simulation (voir paragraphe 3.3.3). Il permet de se déplacer sur la totalité de l'espace et d'en afficher une partie dans le navigateur. Lorsque l'utilisateur est autorisé à modifier la configuration du réseau (en gros, lorsqu'il n'y a pas de simulation en cours), la case pointée par le curseur de la souris est encadrée en vert. L'outil courant peut ainsi être appliqué à cette case par simple clic.

Les outils et le contrôle de la simulation sont disponibles dans deux fenêtres flottantes, indépendantes du navigateur. Cela rend plus flexible à la fois l'utilisation et la programmation de l'ensemble.

3.3.2 Interface d'édition

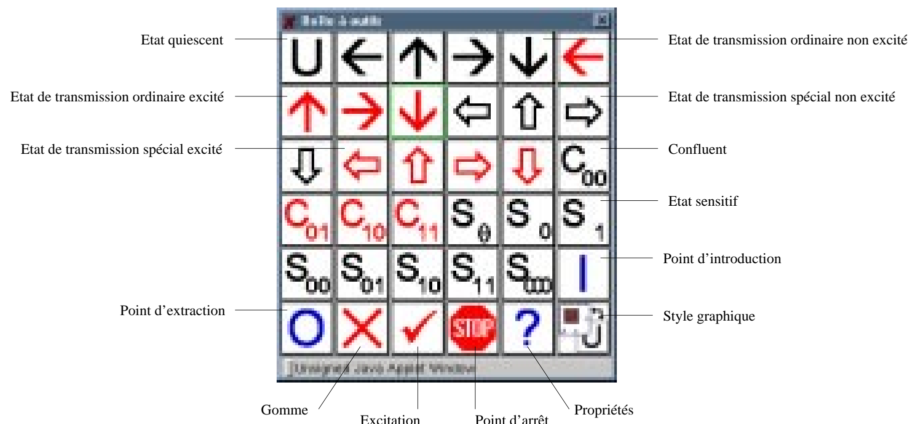
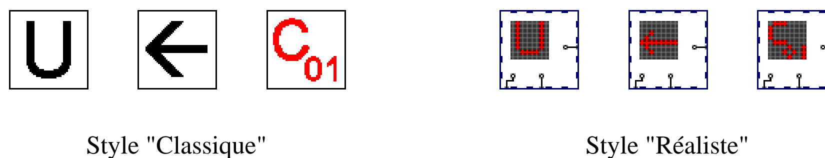


FIG. 3.2: Boîte à outils pour l'édition

Les outils d'édition sont regroupés dans une boîte à outils : on y trouvera les différents états de l'automate (soit 29 outils pour le biodule 602), plus les fonctionnalités suivantes :

- l'outil "point d'introduction" ;
- l'outil "point d'extraction" ;
- l'outil "gomme" ;
- l'outil "excitation", qui permet d'appliquer une excitation à un état de transmission ou à un confluent ;
- l'outil "point d'arrêt" ;
- l'outil "propriétés", qui fournit une liste des caractéristiques de la celule sélectionnée, avec possibilité d'en modifier certaines (suivant le type de cellule).

Sur la figure 3.2, on peut constater qu'un dernier outil a été rajouté ; C'est un outil purement esthétique, qui permet en fait de choisir, le cas échéant, entre plusieurs "styles graphiques" (*skins*).



Style "Classique" Style "Réaliste"

FIG. 3.3: Les styles graphiques standard pour les cellules

Enfin, il faut savoir que la boîte à outils est dynamique dans le sens que son contenu est automatiquement modifié en fonction des caractéristiques du biodule utilisé ; Par exemple, si on implémente l'extension de Pesavento (voir chap. 5) à 32 états, la boîte à outils en tient compte. Les détails de ce mécanisme sont expliqués dans le paragraphe 4.2

3.3.3 Interface de simulation

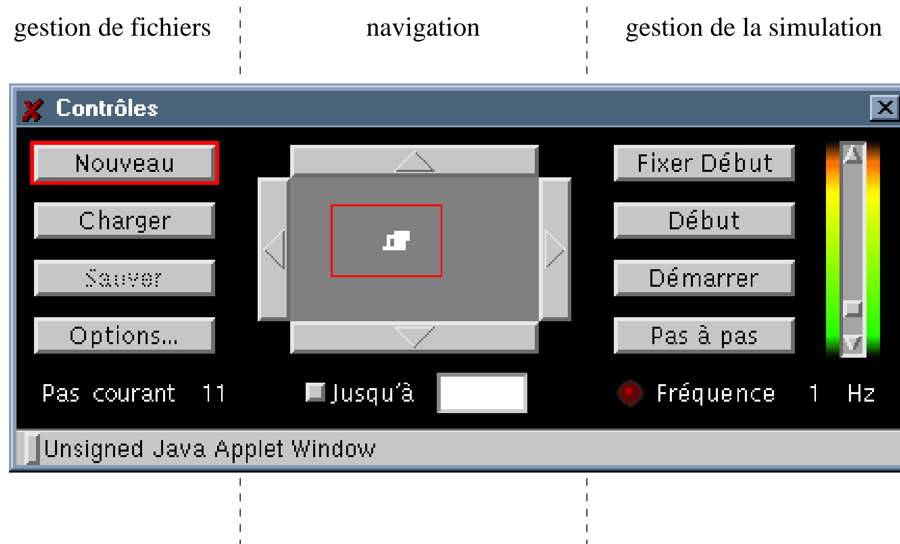


Fig. 3.4: Contrôles de simulation

Elle se compose de trois parties distinctes :

- **La partie gestion de fichiers**, à gauche ; le bouton “Nouveau” efface complètement la simulation en cours, après confirmation. Le bouton “Charger” permet d'accéder à une liste de configurations prédéfinies (pulseur, décodeur, bascule, etc.). Les boutons “Sauver” et “Options...” sont présents pour d'éventuelles extensions futures, mais n'ont pas d'effet particulier pour le moment.
- **La partie navigation**, au milieu ; quatre boutons permettent de faire défiler la partie visible du réseau (symbolisée par un rectangle rouge) dans les quatre directions. Il est également possible de cliquer directement sur la partie centrale (réseau à échelle réduite) pour centrer la vision sur une zone particulière.
- **La partie gestion de la simulation**, à droite ; les boutons “Fixer Début” et “Début” permettent respectivement de fixer le temps zéro de la simulation et d'y revenir. Le bouton “Démarrer” lance la simulation ; la fréquence d'horloge se règle avec le curseur vertical et peut varier entre 0.1 et 10 Hz. Le bouton “Pas-à-Pas” effectue un pas de simulation. Il est possible de fixer une limite supérieure (facultative) au nombre de pas à effectuer (champ “Jusqu'à”).

Chapitre 4

Implémentation

La peur de l'ennui est la seule excuse du travail.
– Renard.

4.1 L'automate de von Neumann en JAVA

4.1.1 Comportement d'une cellule

La cellule a été implémentée à l'aide de deux classes, `Biodule` et `Biodule602`. `Biodule` est une classe abstraite qui définit les propriétés communes à tout type de cellule, en particulier la couche d'interaction avec l'extérieur. `Biodule602` est une classe dérivée de `Biodule`.

Chaque cellule possède quatre pointeurs vers ses cellules voisines. Elle dispose également d'une série de d'indicateurs (*flags*) qui lui permettent de savoir si elle doit tenir compte de l'information venant de chacune de ses voisines et si elle doit émettre de l'information vers celles-ci. Ces indicateurs sont mis à jours par les méthodes `setInputs` et `setOutputs`.

L'état futur est calculé dans la méthode `nextState`, en fonction de l'état courant et de deux informations supplémentaires, fournies par les méthodes `getExcitation` et `kill`.

La méthode `kill` détermine, pour un état de transmission uniquement, si celui-ci doit être détruit (passage à l'état quiescent). La méthode `getExcitation` est plus complexe ; le résultat obtenu est un ensemble de caractéristiques de l'excitation reçue par la cellule, selon un code sur 4 bits :

- bit 0 : vaut 1 si parmi les signaux reçus, il y en a au moins un qui provient d'un état de transmission ordinaire, 0 sinon ;
- bit 1 : vaut 1 si parmi les signaux reçus, il y en a au moins un qui provient d'un état de transmission spécial, 0 sinon ;
- bit 2 : vaut 1 si parmi les signaux reçus, il y en a au moins un qui provient d'un état confluent C_{1x}, 0 sinon ;
- bit 3 : vaut 1 si tous les états de transmission ordinaires voisins (dirigés vers la cellule courante) ont transmis un signal, 0 sinon. Ceci sert au comportement en porte ET pour les confluents.

L'état futur est placé dans la variable `futureState`. Toutefois, ceci ne signifie pas que la cellule a changé d'état. Le changement n'a lieu qu'à l'appel de `transition`, qui réalise le basculement (il s'agit essentiellement d'une mise à jour de la variable `currentState`).

Le mécanisme d'héritage employé permet de définir facilement de nouveaux types de cellules, utilisables directement par le simulateur. Par exemple, pour implémenter l'extension de Pesavento, il suffit d'écrire les méthodes de calcul correspondantes et de les incorporer dans une classe dérivée de `Biodule`.

4.1.2 Comportement du réseau

Dans le simulateur (classe `BdSimulator`), les cellules sont réparties dans une matrice de 50x50 = 2500 éléments (taille arbitraire). La puissance de l'allocation dynamique et du ramasse-miettes en JAVA permet de gérer la création de cellules et leur destruction pratiquement sans aucune programmation supplémentaire. A chaque création, on appelle le constructeur de la classe `Biodule602` ; pour la destruction, il n'y a rien à faire.

Outre le constructeur, on utilise également la méthode `connect` qui réalise les connexions de la cellule avec chacune de ses voisines, par emploi de `Biodule.setNeighbour`. L'ensemble est réalisé au sein de `addBiodule`.

Lors de la simulation, les transitions sont réalisées séquentiellement :

```
/* Calcul des états futurs     */

for (int i=0;i<MAXSIZE;i++)
  for (int j=0;j<MAXSIZE;j++)
    if (machine[i][j]!=null) machine[i][j].compute();

/* Transition (coup d'horloge) */

for (int i=0;i<MAXSIZE;i++)
  for (int j=0;j<MAXSIZE;j++)
    if (machine[i][j]!=null) machine[i][j].transition();
```

où `machine[i][j]` est la cellule aux coordonnées (i,j). Ce mécanisme est implanté dans la méthode `calculateAll`.

Bien que JAVA soit capable de manipuler les processus légers (*threads*), nous avons constaté que leur utilisation pour calculer les états futurs en parallèle n'était pas une bonne stratégie ; en effet, lorsque le nombre de cellules devient important (quelques dizaines), la seule gestion des processus nécessite plus de temps que les calculs eux-mêmes !

Par contre, nous avons implanté un tel processus pour la simulation de l'horloge et la synchronisation du calcul des transitions et de l'affichage. Ainsi, `calculateAll` est appelée à intervalle régulier (selon une fréquence à choix, voir paragraphe 3.3.3), et précis.

Ce même processus léger s'occupe également du suivi des points d'arrêt par appel à la méthode `Biodule.breakPointReached` dans une boucle, selon le même principe que pour le calcul des transitions.

Quant aux points d'introduction et d'extraction, qui se présentent à l'utilisateur comme des cellules au même titre que celles de l'automate, il n'est pas question de les implémenter comme des états supplémentaires dans `Biodule`. Les classes dérivées de `Biodule` ne doivent gérer que les règles véritables de transition, sans adjonction.

Le programme utilise une technique inspirée de la réalisation matérielle (paragraphe 3.2.3) pour le point d'introduction ; on va en fait simuler une cellule dans un état de transmission adéquat (ordinaire, spécial, excité ou non), variable au cours du temps, pour diffuser la séquence. Concrètement, il s'agit de forcer l'état de la cellule à chaque coup d'horloge, grâce à la méthode `Biodule.setState`.

Pour le point d'extraction, il s'agit de récupérer l'état courant d'une cellule. L'état quiescent se prête bien au problème, puisqu'il passe dans l'état sensitif S_θ dès réception d'une excitation ordinaire ou spéciale (voir paragraphe 2.2.5) : c'est celui que l'on simulera, avec retour de l'état S_θ vers l'état quiescent à chaque excitation.

La synchronisation des points d'introduction et d'extraction avec l'horloge est réalisée par le processus léger mentionné précédemment.

4.2 L'interface utilisateur

4.2.1 La boîte à outils

La boîte à outils est décomposée en deux classes, `ToolBox` et `Tool`. Lors de la construction de la boîte à outils, on donne en paramètre le type de cellule (`Biodule`) qui sera utilisé (dans le cas actuel, il s'agit de `Biodule602`). Il faut également passer un tableau contenant les ressources graphiques pour les icônes. La fenêtre est alors composée en fonction des outils de base (gomme, point d'arrêt, etc.) et des états possible de la cellule. Chaque élément de la fenêtre est un objet de la classe `Tool`.

Lorsqu'un clic souris a lieu sur l'un des outils, celui-ci avertit la boîte qu'une modification doit être faite (méthode événementielle `ToolBox.mouseClicked`). La boîte se charge de la mise à jour en changeant les propriétés des outils concernés (méthode `Tool.setState`) et le cas échéant, de l'appel à la fenêtre de propriétés.

4.2.2 La fenêtre de contrôle

La classe `ControlBox` gère la fenêtre de contrôle de simulation. Elle fait assez peu de choses en elle-même ; Elle se contente essentiellement de transmettre les événements générés par les éléments graphiques qui la composent (boutons...) aux classes responsables des actions correspondantes. Ceci est réalisé par la méthode événementielle `actionPerformed`. Les seuls événements gérés localement sont ceux liés au mouvement sur la configuration et à la fréquence de simulation, ainsi que l'appel des fenêtres de chargement et d'options.

4.2.3 La grille d'édition

La grille d'édition est gérée dans la classe `BdSimulator`, qui est dérivée de `Applet`. C'est le corps principal du programme ; il contient le moteur de simulation mentionné plus haut, ainsi que la plupart des méthodes influentes :

– `activateBiodule` : réponse à la commande "excitation" dans la boîte d'outils ;
– `locateInput` : réponse à la commande "point d'introduction" ;
– `locateOutput` : réponse à la commande "point d'extraction" ;
– `nextSkin` : réponse à la commande "Style graphique" ;
– `eraseAll` : réponse à la commande "Nouveau" dans la fenêtre de contrôle ;

– `setFrequency` : réponse à un changement de la fréquence de simulation ;
– `saveState` : réponse à la commande “Fixer Début” ;
– `restoreState` : réponse à la commande “Début”.

L'affichage des cellules, le mouvement dans le réseau et le raffraichissement sont réalisés directement par la méthode `paint`, commune à toutes les *applets* et responsable du contexte graphique.

4.3 Structure globale, interactions entre classes

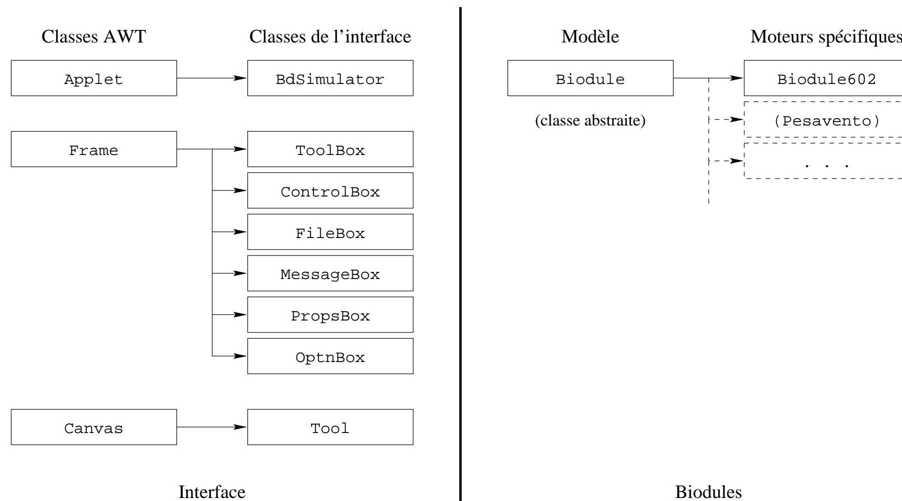


FIG. 4.1: Classes composant l'application et héritage

L'application a été décomposée en une dizaine de classes. La plupart d'entre elles s'occupent de la gestion de l'interface. Ici intervient un aspect très intéressant de JAVA 1.1 : la manipulation des événements (souris, clavier...). Avec JAVA 1.1, il est maintenant possible de faire réagir automatiquement les classes de l'AWT en fonction d'événements très divers, envoyés par des objets précis (impossible en JAVA 1.0 sans la création de méthodes spécifiques). Comme nous l'avons déjà vu, ce concept a été largement exploité pour mettre en œuvre les interactions nécessaires.

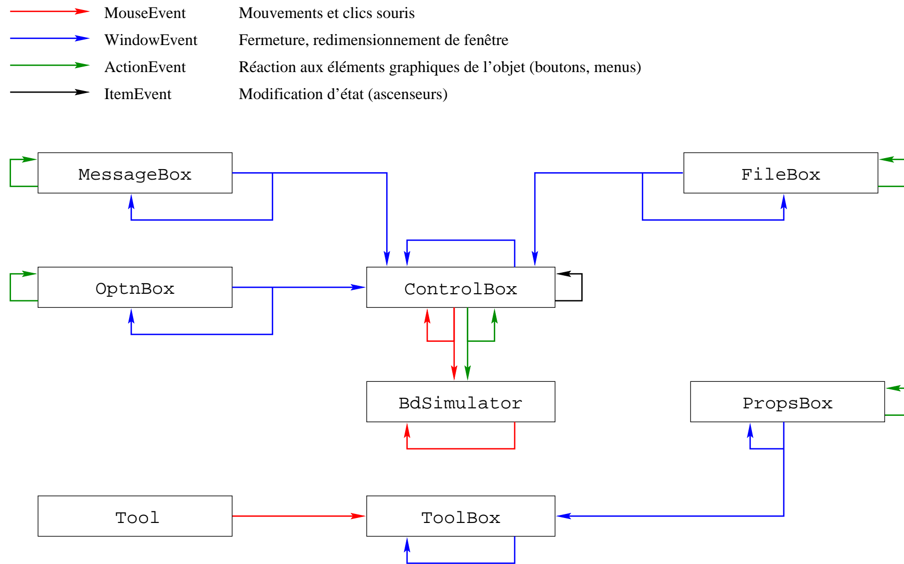


FIG. 4.2: Gestion des événements dans le simulateur

La majeure partie des événements est envoyée à la fenêtre de contrôle de simulation (voir paragraphe 3.3.3). C'est en effet elle principalement qui régit le comportement de l'application, même si le moteur de simulation se trouve dans la classe `BdSimulator`. Toutes les fenêtres gèrent les événements envoyés par leurs éléments graphiques. Les interactions entre `BdSimulator` et `ControlBox` correspondent aux mouvements dans la grille d'édition et au contrôle de la configuration.

Chapitre 5

Limitations et extensions envisageables

Un travail réglé et des victoires après des victoires, voilà sans doute la formule du bonheur.
– Alain.

5.1 Limitations

Les limitations actuelles de l'application sont essentiellement dues à des contraintes sur le langage JAVA :

- difficultés avec la manipulation de fichiers dans une *applet* ;
- lenteur relative du langage, interprété dans le navigateur, qui peut réduire l'ergonomie de l'interface, et surtout, limite les capacités de calcul ;
- JAVA est un standard émergeant ; à ce titre, ses spécifications ne sont pas finalisées, et le langage n'est actuellement pas aussi portable qu'on le prétend ;

Au niveau de la simulation, la taille maximale du réseau est de 50x50 = 2500 cellules. Mais ceci est une contrainte arbitraire, qui peut être aisément modifiée, sous réserve de capacités de mémoire et de calcul suffisantes.

5.2 Extensions

5.2.1 Gestion des fichiers

Du fait des limitations imposées par le langage JAVA dans la programmation des *applets*, il ne nous a pas été possible de mettre en oeuvre une sauvegarde des réseaux réalisés avec le simulateur. Néanmoins, il existe des solutions à ce problème ; celles-ci nécessitent des développements importants qui sortent du cadre même du simulateur. En effet, les *applets* ne gèrent pas les accès directs à la mémoire de masse, mais par contre, elles possèdent de nombreuses facilités pour l'échange de données via un réseau informatique, et via le *World Wide Web* en particulier. La solution serait donc de créer un serveur de fichiers, responsable de la gestion et de l'archivage des données, et accessible par l'*applet* via un protocole restant à définir. Naturellement, tous les problèmes habituels de confidentialité, protection, reconnaissance de propriété des données viennent s'ajouter au développement de cette partie du système.

5.2.2 Extension du biodule 602

Le biodule 602 réalise la cellule telle qu'imaginée par John von Neumann. Une extension du concept de von Neumann a été trouvée par U. Pesavento [4], qui a élargi le nombre d'états de l'automate de 29 à 32, rendant du même coup la création du constructeur universel beaucoup plus aisée (de l'ordre de 20'000 cellules au lieu des 200'000 initialement prévues). Le simulateur ayant été écrit en JAVA, de manière aussi modulaire que possible, le comportement d'un biodule basé sur les règles de Pesavento est implémentable sans difficulté majeure.

On peut utiliser la capacité de JAVA à charger dynamiquement des classes pour proposer à l'utilisateur de choisir un type de biodule à étudier plus particulièrement.

5.2.3 Entrées/sorties sur l'automate

Les concepts mis en œuvre pour l'introduction et la récupération de séquences binaires dans l'automate (voir chapitre 3) peuvent être étendus de manière à permettre plusieurs points d'introduction et plusieurs points d'extraction simultanés. Cette extension est intéressante dans le cadre de la simulation par l'automate de von Neumann de circuits logiques de complexité très variable (on peut relativement facilement simuler une combinaison de portes logiques et de bascules à plusieurs entrées avec cet automate).

Annexe A

Liste des classes et des méthodes

Cette liste est susceptible de modifications ultérieures. Seules les classes et méthodes essentielles sont actuellement mentionnées.

```
public abstract class Biodule {

  /* Constantes d'ordre général ------------------------------------------ */

  public static final int B_NORTH = 1;
  public static final int B_EAST  = 2;
  public static final int B_SOUTH = 3;
  public static final int B_WEST  = 0;

  public static final int B_NEAST = 4;
  public static final int B_NWEST = 5;
  public static final int B_SEAST = 6;
  public static final int B_SWEST = 7;

  public static final int B_NOBRK = 65535;


  /* variables fournies pour le confort ---------------------------------  */
  /* Il n'est pas obligatoire de les réutiliser                            */

  int currentState, futureState, bkptState;

  /* Methodes ------------------------------------------------------------ */

  public abstract int numberOfStates();
  /* Retourne le nombre d'états possible de la cellule                     */

  public abstract String skinFileName() [];
  /* Fournit un ou plusieurs noms de fichiers génériques pour le graphisme */
```

```
  public int getState();
  /* Renvoie l’état courant de la cellule                                    */

  public abstract Biodule getNeighbour(int direction);
  /* Renvoie la cellule voisine dans la direction spécifiée                  */

  public abstract void setNeighbour(int direction, Biodule cell);
  /* Fixe la cellule voisine dans la direction spécifiée                     */

  public abstract void compute();
  /* Un appel à cette methode provoque le calcul de l’état futur             */

  public abstract void transition();
  /* Un appel à cette methode provoque le basculement à l’état futur         */

  public void setState(int state);
  /* Force l’état de la cellule à state                                      */

  public abstract void excite();
  /* Excite une cellule, avec éventuel changement d’état à la clef           */

  public abstract boolean excited();
  /* Vrai si la cellule se trouve dans un état ‘‘excité’’                    */

  public abstract boolean inputsFrom(int direction);
  /* Vrai si la cellule utilise l’info venant de la direction spécifiée      */

  public abstract boolean outputsTo(int direction);
  /* Vrai si la cellule émet de l’info dans la direction spécifiée           */

  public abstract boolean link(int direction);
  /* Vrai si un lien est établi entre la cellule courante et sa voisine      */
  /* dans la direction spécifiée                                             */

  public void setBreakpoint(int state);
  /* Place un point d’arrêt                                                  */

  public int getBreakPoint();
  /* Renvoie le point d’arrêt courant                                        */

  public boolean breakPointReached();
  /* Vrai si le point d’arrêt a été atteint                                  */

}
```

```
public class Biodule602 extends Biodule {

  Biodule602 (int startState);
  /* Constructeur                                                           */

  Biodule602 ();
  /* Constructeur                                                           */

  private int getExcitation();
  /* Permet de déterminer le type d'excitation appliquée à la cellule      */

  private boolean kill();
  /* Vrai si la cellule a reçu un signal de destruction                     */

  public int nextState();
  /* Méthode interne de calcul de l'état futur                              */

  private int turnAbout(int direction);
  /* Renvoie la direction opposée à celle spécifiée                         */

  private void setOutputs(int state);
  /* Fixe les sorties en fonction de l'état de la cellule                   */

  private void setInputs(int state);
  /* Fixe les entrées en fonction de l'état de la cellule                   */

}
```

```
public class Tool extends Canvas {

  /* Méthodes ------------------------------------------------------------ */

  Tool (Image icon, int id);
  /* Constructeur                                                            */

  public boolean getState ();
  /* Vrai si l'outil est sélectionné                                         */

  public int getId();
  /* Renvoie l'identificateur numérique de l'outil                           */

  public boolean isEnabled();
  /* Renvoi vrai si l'outil est disponible                                   */

  public void setEnabled(boolean e);
  /* Rends l'outil disponible ou non                                         */

  public void setState(boolean c);
  /* Sélectionne ou désélectionne l'outil                                    */

  public void setIcon(Image i);
  /* Fixe l'icône représentative                                             */

}
```

```
public class ToolBox extends Frame implements MouseListener, WindowListener {

  /* Constantes d’ordre général ------------------------------------------- */

  public static final int T_Tools      = 7;     /* Nombre d’outils spéciaux */

  public static final int T_NOTHING    = 65535; /* Pas d’outil              */
  public static final int T_INPUT      = 65534; /* Point d’introduction     */
  public static final int T_OUTPUT     = 65533; /* Point d’extraction       */
  public static final int T_REMOVE     = 65532; /* Gomme                    */
  public static final int T_ACTIVATE   = 65531; /* Excitation               */
  public static final int T_BREAKPT    = 65530; /* Point d’arrêt            */
  public static final int T_PROPERTIES = 65529; /* Propriétés               */
  public static final int T_SKIN       = 65528; /* Styles graphiques        */
  public static final int T_LAST       = T_SKIN;/* Dernier outil            */

  /* Méthodes ------------------------------------------------------------- */

  ToolBox(Biodule b, Image ip[]);
  /* Constructeur                                                           */

  private void addTool(int id);
  /* Ajoute un outil dans la boîte à outils                                 */

  public int getActiveTool();
  /* Renvoie l’outil courant                                                */

}
```

```
public class BdSimulator extends Applet implements MouseListener,
                                                   MouseMotionListener,
                                                   ActionListener,
                                                   Runnable {

  /* Constantes d'ordre général ------------------------------------------- */

  /* Type de rafraichissement de la grille à réaliser                        */

  final static public int TARGET_CHANGED          = 1; /* Chgt de cible      */
  final static public int BIODULE_ADDED           = 2; /* Cellule ajoutée    */
  final static public int BIODULE_REMOVED         = 3; /* Cellule retirée    */
  final static public int BIODULE_CHANGED_STATE   = 4; /* Chgt d'état        */
  final static public int TARGET_LOST             = 5; /* Souris hors grille */
  final static public int ALL                     = 0; /* Raffr. complet     */

  final static public int MAXSIZE                 = 50; /* 50x50 cellules    */

  /* Méthodes ------------------------------------------------------------- */

  public void init();
  /* Initialisation de l'applet                                              */

  private void stopload();
  /* Notification de fin de chargement des ressources graphiques             */

  private Image loadImage(String animage);
  /* Chargement d'une image                                                  */

  private void activateBiodule(int xb, int yb);
  /* Excitation d'un biodule                                                 */

  private void locateInput(int xb, int yb);
  /* Positionnement du point d'introduction                                  */

  private void locateOutput(int xb, int yb);
  /* Positionnement du point d'extraction                                    */

  private void nextSkin();
  /* Changement de style graphique                                           */

  private void eraseAll();
  /* Remise à zéro (efface toute la grille)                                  */

  private void connect(int xb, int yb);
  /* Réalise une connexion de biodule avec ses voisins                       */
```

```
  private void calculateAll();
  /* Calcule l'état futur de tout le réseau                                    */

  private void setFrequency(float f);
  /* Règle la fréquence de simulation                                          */

  private void saveState();
  /* Mémorise l'état de la configuration (temps zéro)                          */

  private void restoreState();
  /* Restaure la configuration mémorisée (retour au temps zéro)                */

}
```

```
public class ControlBox extends Frame implements MouseListener,
                                                 WindowListener,
                                                 AdjustmentListener {

  /* Méthodes ------------------------------------------------------------ */

  public ControlBox(ActionListener al, Image pool[]);
  /* Constructeur                                                          */

  public void ledON();
  public void ledOFF();
  public void switchLed();
  /* Méthode de gestion du témoin ‘‘lumineux’’ de l’horloge                */

}
```

Bibliographie

[1] John von Neumann. *Theory of Self-Reproducing Automata*. University of Illinois Press, Urbana and London, 1966.

[2] Jean-Luc Beuchat, Jacques-Olivier Haenni. *Automate cellulaire autoréplicateur de von Neumann*. Laboratoire de Systèmes Logiques, EPFL, 1998.

[3] Jean-Luc Beuchat, Jacques-Olivier Haenni. *Logidule de von Neumann*, rapport interne. Laboratoire de Systèmes Logiques, EPFL, 1995.

[4] Umberto Pesavento. *Von Neumann's Universal Constructor*. 1995.

[5] Chris Langton. *Self-reproduction on a Cellular Automaton*. Physica 10D, 1984, pp. 135-144.

Table des figures

2.1 Etats de transmission ordinaires 4
2.2 Les états confluents (1) 5
2.3 Les états confluents (2) 5
2.4 Les états confluents (3) 5
2.5 Les états sensitifs 6
2.6 Exemples de destruction de cellules 7
2.7 Le biodule 602 dit “de von Neumann” 8
2.8 Représentation théorique et réalisation pratique 9

3.1 Aspect général de l'interface 14
3.2 Boîte à outils pour l'édition 15
3.3 Les styles graphiques standard pour les cellules 15
3.4 Contrôles de simulation 16

4.1 Classes composant l'application et héritage 20
4.2 Gestion des événements dans le simulateur 21

Index

A
Allocation dynamique 18
Applet
  Restrictions 10, 22
Automate universel 2

B
Bras constructeur 12

C
Classe
  Abstraite 17
  Dérivée 17
Contraintes 22

E
Evénements 20
Extensions 22

F
Fonctionnalités
  Edition 11
  Pas-à-pas 11
  Points d'arrêts 12
  Séquences binaires 12
  Simulation 11

I
Interface
  Critères 13
  Edition 15, 19
  Grille d'édition 13, 19
  Simulation 16
  Visualisation 14

J
JAVA 1.1.2 10

L
Limitations 22

O
Outils
  Gestion de fichiers 11
  Gomme 11
  Propriétés 11

P
Point d'extraction 12, 23
Point d'introduction 12, 23
Processus léger 18

S
Syst. et prog. génétiques 1
Système de navigation 14

T
Temps zéro 12

Table des matières

1 Introduction **1**
1.1 Présentation du document . . . 1
1.2 But du projet . . . 1

2 Le biodule 602 **2**
2.1 L'automate de von Neumann . . . 2
2.2 Etats et règle de transition détaillés . . . 3
2.2.1 L'état quiescent . . . 3
2.2.2 Les états de transmission ordinaires . . . 3
2.2.3 Les états de transmission spéciaux . . . 4
2.2.4 Les états confluents . . . 4
2.2.5 Les états sensitifs . . . 5
2.2.6 Destruction d'une cellule . . . 7
2.3 Réalisation câblée . . . 8

3 Développement **10**
3.1 Considérations préléminaires . . . 10
3.2 Présentation des fonctionnalités . . . 10
3.2.1 Fonctions d'édition . . . 11
3.2.2 Fonctions de simulation . . . 11
3.2.3 Manipulation de séquences d'excitations . . . 12
3.3 L'interface utilisateur : critères . . . 13
3.3.1 Vision au niveau global . . . 14
3.3.2 Interface d'édition . . . 15
3.3.3 Interface de simulation . . . 16

4 Implémentation **17**
4.1 L'automate de von Neumann en JAVA . . . 17
4.1.1 Comportement d'une cellule . . . 17
4.1.2 Comportement du réseau . . . 18
4.2 L'interface utilisateur . . . 19
4.2.1 La boîte à outils . . . 19
4.2.2 La fenêtre de contrôle . . . 19
4.2.3 La grille d'édition . . . 19
4.3 Structure globale, interactions entre classes . . . 20

5 Limitations et extensions envisageables **22**
5.1 Limitations 22
5.2 Extensions 22
5.2.1 Gestion des fichiers 22
5.2.2 Extension du biodule 602 23
5.2.3 Entrées/sorties sur l'automate 23

A Liste des classes et des méthodes **24**

Bibliographie **32**

Table des figures **33**

Index **34**