ÉCOLE SUPÉRIEURE D'ÉLECTRICITÉ

Modélisation des systèmes

Cours de troisième année, option SI

Frédéric Boulanger

2009-11-20

Table des matières

1 Modélisation et conception **1**
1.1 Modéliser ou concevoir ? . . . 1
1.2 Modèles de calcul . . . 1

2 Le logiciel enfoui **3**
2.1 Qu'est-ce que le logiciel enfoui . . . 3
2.2 Les problèmes de l'enfoui . . . 4
2.2.1 Exactitude temporelle . . . 4
2.2.2 Le parallélisme . . . 4
2.2.3 La vivacité . . . 5
2.2.4 La réactivité . . . 5
2.2.5 L'hétérogénéité . . . 6
2.3 Limitations des méthodes de génie logiciel . . . 6
2.3.1 La définition des interfaces . . . 7
2.3.2 Procédures et objets . . . 8
2.3.3 La conception de matériel . . . 8
2.3.4 Les systèmes d'exploitation temps-réel . . . 8
2.3.5 Les modèles objet temps-réel . . . 9

3 Modélisation par acteurs **11**
3.1 Origine des acteurs . . . 11
3.2 Syntaxe abstraite, syntaxe concrète . . . 12
3.3 Sémantique . . . 12

4 Modèles de calcul **15**
4.1 Modèles de calcul et conception . . . 15
4.2 Modèles à flots de données . . . 15
4.2.1 Les réseaux de processus . . . 19
4.3 Les modèles déclenchés par le temps . . . 21
4.3.1 Le modèle réactif synchrone . . . 21
4.3.2 Les modèles à événements discrets . . . 22
4.3.3 Communication par rendez-vous . . . 23
4.4 Modèles à temps continu . . . 23

5 Automates finis **27**
5.1 Domaine d'application . . . 27
5.2 Automates et acteurs . . . 27
5.3 Exemple d'automate . . . 28
5.4 Simulation et bisimulation . . . 29
5.4.1 Relations entre comportements . . . 30
5.5 Composition d'automates et hypothèse synchrone . . . 30
5.5.1 Automates à sorties déterminées par l'état . . . 33
5.5.2 Sémantique constructive de la rétroaction . . . 33
5.5.3 Recherche exhaustive . . . 34

6 Esterel **35**
6.1 Introduction . . . 35
6.2 Le système ABRO . . . 36
6.3 Traces d'exécution . . . 36
6.4 L'automate d'ABRO . . . 37
6.5 ABRO en Esterel . . . 37
6.5.1 Le principe WTO . . . 38
6.6 Signaux valués . . . 39
6.7 Signaux et variables . . . 39
6.8 Préemption faible et immédiate . . . 40
6.9 Test du statut d'un signal . . . 42
6.10 Causalité constructive . . . 43

7 Systèmes mixtes et modaux **45**
7.1 Introduction . . . 45
7.2 Modèles mixtes . . . 45
7.3 Modèles modaux . . . 46
7.4 Automates temporisés . . . 47

Chapitre 1

Modélisation et conception

1.1 Modéliser ou concevoir ?

Modéliser (*to model*) consiste à construire une représentation formelle d'un système ou d'un sous-système. Un modèle peut-être une construction mathématique, c'est-à-dire une ensemble d'assertions sur les propriétés du système. Un modèle peut aussi être constructif, auquel cas il définit une procédure calculatoire qui imite un ensemble de propriétés du système. Les modèles constructifs sont généralement utilisés pour décrire la réponse d'un système aux stimuli que lui fournit son environnement. Un modèle constructif est aussi appelé un modèle *exécutable*.

Les modèles qui ne sont pas constructifs ont aussi leur utilité. Un modèle peut en effet être simplement explicatif : il explique ce que l'on observe, et on peut alors croire ou non que ce modèle décrit la réalité. Certains modèles explicatifs sont plus intéressants car il permettent de prédire ce que l'on observera, ce sont les modèles prédictifs. Un modèle prédictif non exécutable, par exemple la théorie newtonienne de la gravitation, peut être rendu exécutable en lui adjoignant une procédure de calcul des propriétés du système à partir des assertions que le modèle fait sur ces propriétés. Dans le cas général, on n'obtient qu'une approximation de la valeur des propriétés (par intégration numérique d'équations différentielles dans notre exemple), mais cette approximation peut être suffisante, même pour envoyer des hommes sur la Lune…

Concevoir (*to design*) consiste à définir un système ou un sous-système. Cela implique en général de définir un ou plusieurs modèles du système et de raffiner les modèles jusqu'à ce que les fonctionnalités désirées soient obtenues en tenant compte des contraintes de l'implémentation. On peut dire qu'un système est entièrement conçu lorsque le modèle qu'on en a est directement exécutable par la plateforme cible que l'on a choisie.

La conception et la modélisation sont donc étroitement liées, puisque la conception entraîne la modélisation. Dans certaines circonstances, des modèles peuvent être immuables : il décrivent des sous-systèmes, des contraintes ou des comportements qui sont imposés. Par exemple, un système d'asservissement de position peut utiliser un modèle du dispositif mécanique à asservir. Ce modèle n'entre pas dans le processus de conception mais est une contrainte pour la conception de l'asservissement.

Les modèles exécutables sont parfois appelés *simulations*, ce qui est approprié quand le modèle exécutable est clairement distinct du système qu'il modélise. Toutefois, dans de nombreux systèmes électroniques, un modèle qui était au départ une simulation finit par devenir une implémentation logicielle du système. La distinction entre le modèle et le système proprement dit devient alors floue. Cette situation se rencontre très fréquemment dans le domaine des systèmes enfouis.

1.2 Modèles de calcul

Un modèle prédictif est construit selon un *modèle de calcul* (*model of computation*) qui est l'ensemble des lois qui gouvernent l'interaction des composants dans le modèle. La théorie newtonienne de la gravitation est ainsi un modèle de calcul des interactions entre corps dues à leur masse. La relativité générale d'Einstein en est un autre.

Le jeu de règles utilisé pour calculer le comportement des composants d'un modèle est *un modèle d'exécution* du modèle de calcul. Un modèle de calcul peut avoir plusieurs modèles d'exécution : il peut exister plusieurs jeux de règles qui donnent les mêmes comportements des composants. Par exemple, pour calculer le mouvement de planètes soumises à la gravitation newtonnienne (modèle de calcul), on peut utiliser la méthode d'intégration d'Euler ou celle de Runge-Kuta (modèles d'exécution) pour intégrer numériquement les équations différentielles du système.

Le choix d'un modèle de calcul dépend du type de modèle que l'on construit. La possibilité de transformer un modèle en implémentation dépend fortement du modèle de calcul utilisé. Certains modèles de calcul sont bien adaptés à la génération de composants matériels, alors que d'autres seront plus facilement implémentés en logiciel. Pour les systèmes enfouis, les modèles les plus utiles sont ceux qui supportent le temps et le parallélisme. Les systèmes enfouis sont en effet constitués de composants qui fonctionnent simultanément et ont de nombreuses sources simultanées de stimuli. De plus, ces systèmes fonctionnent dans un environnement où le temps a un sens et où la date à laquelle une donnée est produite est au moins aussi importante que la valeur de cette donnée.

Chapitre 2

Le logiciel enfoui

2.1 Qu'est-ce que le logiciel enfoui

Le logiciel enfoui (*embedded software*) est le logiciel qui se trouve dans des dispositifs qui ne sont pas principalement des ordinateurs. Ce type de logiciel est très répandu dans les automobiles, les téléphones, les équipements électroniques grand-public, les jouets, les avions, les systèmes de sécurité, les imprimantes, les photocopieurs, les thermostats etc. Il est probable qu'une personne qui utilise des objets techniques interagit plus avec du logiciel enfoui qu'avec du logiciel conventionnel. Une caractéristique clef du logiciel enfoui est qu'il interagit avec le monde physique et a donc des contraintes temporelles auxquelles échappe le logiciel « de bureau ».

Une autre caractéristique de ce logiciel est qu'il est *réactif*, c'est-à-dire qu'il doit réagir à son environnement au rythme de cet environnement. Ceci le différencie des systèmes interactifs qui réagissent à leur environnement, mais à leur propre rythme, et des systèmes transformationnels qui reçoivent leurs données, effectuent un calcul et fournissent un résultat.

Les théories de l'informatique ont généralement comme présupposé que le logiciel n'est que la réalisation de fonctions mathématiques sous formes de procédures de calcul. Ces fonctions font correspondre un jeu de résultats à un jeu d'arguments, et le mécanisme utilisé pour faire le calcul est le plus souvent ignoré. Il n'est pris en compte que lorsqu'on s'intéresse à la finitude de la procédure de calcul ou à sa complexité.

Malheureusement, cela ne fonctionne pas bien avec le logiciel enfoui car son rôle principal n'est pas de transformer des données mais d'interagir avec le monde physique. Ce type de logiciel ne s'exécute pas sur des ordinateurs au sens habituel du terme, mais dans des dispositifs où la présence de logiciel n'est pas immédiate : téléphones, automobiles, chaînes Hi-Fi, jouets etc.

Le logiciel dont le rôle est d'interagir avec le monde physique doit être considéré comme ayant certaines propriétés du monde physique : il prend du temps pour s'exécuter et consomme de l'énergie. Contrairement aux algorithmes classiques, il ne doit pas « terminer » (il s'agirait d'une panne). On est donc loin des procédures idéalisées d'Alan Turing, même si elles conservent un intérêt pour modéliser certains aspects du logiciel enfoui.

Cet aspect « physique » du logiciel embarqué l'a placé un peu à part des progrès effectués en termes d'abstraction et de méthodes de conception au cours de la deuxième moitié du XX^{e} siècle. Au lieu d'utiliser des modèles objets, des systèmes de types polymorphes et des gestionnaires de mémoire automatiques, les concepteurs de systèmes enfouis programment encore trop souvent des DSP [1] en C, parfois encore en langage d'assemblage.

Ces concepteurs de systèmes enfouis sont rarement des informaticiens. Ils sont en général experts dans le domaine d'application du système et connaissent bien les architectures cibles du système. La tâche de l'informaticien est de mettre au point de meilleures abstractions pour la conception de ces systèmes. Les experts des systèmes enfouis sont souvent sceptiques devant un système de mémoire virtuelle qui va perdre un temps précieux lors d'un défaut de page, et ce à un moment imprévisible ; ou encore devant un ramasse-miette qui va bloquer une tâche importante le temps de lui trouver assez de mémoire pour tourner. D'un autre côté,

1. *Digital Signal Processor*, Processeur de traitement numérique du signal

ils ont à gérer des systèmes de plus en plus complexes, qui fonctionnent en réseau et doivent incorporer dynamiquement des mises-à-jour ou des personnalisations.

Certains systèmes enfouis, notamment dans les transports et dans le nucléaire, doivent subir une procédure de certification avant de pouvoir être utilisés en production. Cette procédure repose sur la confiance que les experts de la commission ont dans les méthodes utilisées par les concepteurs du système. Dans ce cadre, les méthodes formelles, en éliminant systématiquement les erreurs liées à l'intuition, ont leur place, mais pour tout ce qui ne se prête pas facilement à la formalisation, un certain nombre de méthodes ou « bonnes pratiques » permettent d'éviter les erreurs classiques et donnent une meilleure lisibilité au projet, ce qui permet à la commission d'évaluer plus facilement la robustesse du système et augmente ainsi sa confiance.

Les problèmes propres au logiciel enfoui nécessitent soit une adaptation des méthodes du logiciel classique, soit des fondements entièrement nouveaux qui tiennent compte de la « physicalité » et mènent à la robustesse.

2.2 Les problèmes de l'enfoui

Un aperçu naïf du logiciel enfoui pourrait laisser croire qu'il s'agit simplement de logiciel qui tourne sur des plates-formes aux ressources limitées. En fait, l'exactitude temporelle, le parallélisme, la vivacité, la réactivité et l'hétérogénéité doivent faire partie des abstractions utilisées dans le modèle de conception de ces systèmes. Elles sont essentielles pour que le système obtenu ait le comportement attendu car il n'est pas suffisant d'assurer la bonne correspondance entre arguments et résultats de la fonction du système.

2.2.1 Exactitude temporelle

Le temps n'apparaît pas dans les théories du calcul. Un calcul pur ne prend pas de temps et n'a aucune relation avec le temps. Les systèmes dits « temps-réel » se contentent généralement d'attribuer à chaque tâche un simple nombre : sa priorité. Dans le meilleur des cas, cette priorité peut varier selon la proximité de l'échéance de la tâche, ce qui permet de prendre en compte l'urgence, mais pas vraiment le temps lui-même. Même les logiques temporelles traitent principalement le temps de façon qualitative (« peut-être », « toujours ») et non quantitative.

Le véritable problème est que le calcul prend du temps, mais que l'évaluation de ce temps est difficile, et est rendue encore plus difficile par les progrès accomplis en architecture des ordinateurs. En effet, une grande part des gains de performance obtenus dans les processeurs modernes vient de l'exploitation de phénomènes statistiques : caches, pré-exécution spéculative d'instructions, prédiction de branchements etc. Ces techniques augmentent les performances moyennes mais rendent difficile la prédiction des temps de réponse courts et nuisent donc à la fiabilité des systèmes enfouis. C'est pourquoi elle ne sont pas utilisées dans les DSP et les micro-contrôleurs. Toutefois, ces techniques ont eu un tel impact sur les performances des processeurs que l'on voit mal comment les systèmes enfouis pourraient continuer à les ignorer, et cela ne pourra se faire que lorsque l'on saura gérer le temps dans le logiciel.

Un autre aspect de l'exactitude temporelle est que, même avec des machines infiniment rapides, le logiciel enfoui doit gérer le temps puisque les processus physiques avec lesquels il interagit évoluent avec le temps.

2.2.2 Le parallélisme

Le monde physique avec lequel interagissent les systèmes enfouis est intrinsèquement parallèle : les différents objets, et même les différents composants d'un objet évoluent concurremment et ne sont pas traités à tour de rôle par un « processeur physique ». De plus, un système enfoui interagit en général avec plusieurs autres systèmes en plus du ou des processus

physiques qu'il gère. Il peut être relié à un réseau, il doit réagir aux commandes d'un éventuel utilisateur ou encore émettre et recevoir des données simultanément.

Les outils informatiques de gestion du parallélisme sont généralement assez primitifs. Les threads, processus, sémaphores et autres moniteurs qui sont classiquement utilisés pour gérer le parallélisme sont d'un niveau d'abstraction si faible qu'il est difficile de construire des systèmes complexes et fiables avec eux. Des règles empiriques du genre « Toujours acquérir les mutex dans le même ordre » prévalent chez ceux qui sont bien obligés de les utiliser, mais ces règles ne garantissent rien de plus qu'un cierge déposé à la chapelle des sondes spatiales perdues.

Un des principaux problèmes des systèmes enfouis est de réconcilier la nature séquentielle du logiciel et le parallélisme du monde physique. Les mécanismes classiques de threads, sémaphores, rendez-vous, moniteurs, fournissent une bonne base mais sont inefficaces en eux-mêmes car leur composition dans des systèmes complexes devient rapidement impossible à comprendre.

Une autre approche du parallélisme, qui semble mieux convenir aux systèmes enfouis, est celle des langages synchrones comme Lustre, Esterel et Signal. Dans cette approche, le parallélisme est utilisé pour son pouvoir expressif lors de la conception des systèmes, mais il est ensuite compilé en un programme purement séquentiel : toutes les actions sont séquencées statiquement. Ce type d'approche donne des programmes très fiables dont on peut même prouver formellement certaines propriétés. Toutefois, l'ordonnancement statique qui élimine les mauvaises surprises du parallélisme n'est possible que si la structure du système est connue à la compilation et ne change pas au cours de l'exécution du système. Cette contrainte est trop forte pour certains systèmes en réseau qui doivent s'adapter dynamiquement à la configuration qu'ils découvrent au cours de leur fonctionnement.

De plus, l'approche réactive synchrone s'appuie sur l'hypothèse que les calculs sont instantannés, ce qui lui permet justement de résoudre le parallélisme en une séquence unique d'actions, mais ce faisant, elle renonce à traiter le problème du temps pris par les calculs.

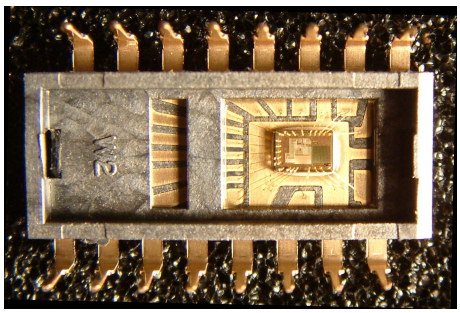
2.2.3 La vivacité

Dans les systèmes enfouis, la vivacité est une propriété critique. Les programmes ne doivent pas terminer ou se bloquer en attente d'événements qui n'auront jamais lieu. Dans le modèle de Turing, tous les programmes qui ne terminent pas sont considérés globalement comme des programmes défectueux. Pour le logiciel embarqué, ce sont les programmes qui terminent qui sont défectueux. L'« étreinte fatale » ou *dead lock* peut être considérée comme une forme de terminaison prématurée pour ces systèmes et doit être absolument évitée. De même que l'indécidabilité de l'arrêt d'une machine de Turing empêche de détecter systématiquement les programmes qui ne s'arrêtent pas, elle empêche aussi d'identifier systématiquement les programmes qui s'arrêtent.

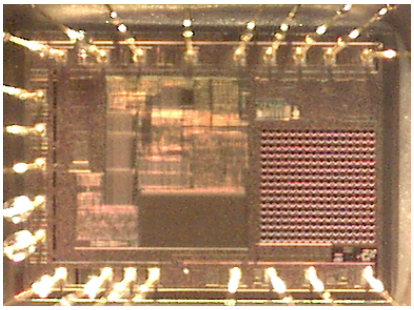
L'exactitude d'un logiciel enfoui n'est donc pas l'exactitude du résultat final, tout d'abord parce qu'il n'y a en général pas de résultat *final* puisque le système fonctionne en continu ; et ensuite parce que ce système doit produire un flux de réponses partielles qui sont non seulement correctes mais arrivent de plus à temps. On voit donc bien qu'un modèle dans lequel l'exactitude du système est uniquement liée à l'exactitude du résultat d'une fonction est assez mal adapté à la description d'un système qui interagit continuellement avec des processus physiques grâce à des capteurs, des actuateurs, tout cela en temps et en heure, en respectant des contraintes de consommation d'énergie, et en étant capable de récupérer après une panne.

2.2.4 La réactivité

Un système réactif est un système qui réagit continuellement à son environnement, et ceci au rythme de cet environnement. Cette définition est à opposer à celle des systèmes interactifs, qui réagissent à leur environnement à leur propre rythme, et à celle des systèmes transformationnels, qui ne sont que des procédures de calcul finies.



Boîtier complet



Détail de la puce

FIGURE 2.1 – Exemple de SoC : souris optique

Les systèmes réactifs ont des contraintes temps-réel et sont souvent critiques pour la sécurité. Au contraire des systèmes transformationnels, les systèmes réactifs ne terminent pas (sauf en cas de panne).

L'approche synchrone est particulièrement bien adaptée à la conception des systèmes réactifs. Des langages comme Esterel, Lustre et Signal ont été utilisés pour des applications où la validation formelle est importante, comme les systèmes critiques de contrôle d'un avion ou d'une centrale nucléaire. Ces langages traitent le parallélisme en analysant statiquement la structure du programme et en effectuant un ordonnancement statique. Toutefois, cette analyse statique permet difficilement de gérer la modularité, et ne permet pas de coder des architectures logicielles adaptatives.

2.2.5 L'hétérogénéité

L'hétérogénéité est une caractéristique intrinsèque des systèmes enfouis, à la fois du point de vue des modèles de calcul que des technologies d'implémentation. Ces systèmes sont souvent composés de sous-systèmes matériels et de sous-systèmes logiciels, et le logiciel enfoui interagit avec du matériel qui a été conçu spécifiquement pour ce système. Ainsi, un SoC (*System on Chip* : système intégré sur une puce) regroupe sur un même morceau de silicium des capteurs, des circuits spécifiques au système, un processeur, de la mémoire vive et une mémoire morte contenant le code que le processeur doit exécuter.

Les systèmes enfouis utilisent aussi conjointement différentes façons de gérer les événements. Certains événements surviennent de façon irrégulière dans le temps (alarmes, actions de l'utilisateur etc.) tandis que d'autres sont réguliers (échantillonnage périodique des entrées). Ces événements ont des tolérances différentes en matière d'exactitude temporelle.

La quête d'un modèle universel capable de traiter au mieux tous les aspects de tous les systèmes ne génère que des guerres entre défenseurs de tel ou tel langage, que tout le monde utiliserait s'il en comprenait la beauté et les subtilités... La modélisation de systèmes enfouis doit supporter l'hétérogénéité, c'est-à-dire autoriser l'utilisation conjointe de plusieurs modèles de calcul dans un modèle d'un système. Ceci ne signifie pas qu'un modèle qui gère à la fois le temps continu et les événements discrets est une mauvaise chose. C'est simplement un modèle de calcul de plus, qui peut avoir son utilité, mais qui ne peut prétendre à l'universalité.

2.3 Limitations des méthodes de génie logiciel

La construction de systèmes logiciels enfouis complexes pourrait bénéficier des technologies à base de composants. Idéalement, ces composants sont réutilisables et représentent une certaine expertise dans un domaine d'application. La composition de tels composants doit avoir un sens, et idéalement, la composition de composants produit un nouveau composant qui peut lui-même être utilisé dans de nouvelles compositions. Pour cela, ces composants doivent être des abstractions du logiciel métier qu'ils encapsulent. Ils doivent masquer les

détails, et ne laisser apparaître que les interfaces externes essentielles, avec une sémantique bien définie.

2.3.1 La définition des interfaces

Le génie logiciel a connu de grand progrès grâce à la banalisation de l'approche objet. La conception par objets est une *technique de composition*, au sens où elle permet de décomposer un système complexe en morceaux plus petits qui masquent leur propre complexité derrière une interface.

La notion d'interface n'est pas nouvelle, et la plus répandue des techniques de composition s'appuyant sur des interfaces est celle qui consiste à définir des procédures. Une procédure est un calcul fini qui reçoit un nombre prédéfini d'arguments et produit un résultat final. Les bibliothèques de procédures forment des lots de composants qui peuvent être réutilisés, échangés et même vendus. De telles bibliothèques ont fourni une abstraction efficace pour des fonctionnalités complexes. On peut considérer que l'approche objet pousse la notion de procédure un peu plus loin en masquant non seulement le code de la procédure mais aussi la structure des données derrière l'interface.

Les procédures sont toutefois peu adaptées aux systèmes enfouis. Il est par exemple assez curieux de définir un codeur de voix pour un téléphone cellulaire sous la forme d'un calcul fini. Ce type de calcul est non borné et transforme un flux non borné de données en un autre flux non borné. Dans la pratique, un tel codeur sera vendu comme un processus à exécuter sur un DSP particulier, mais il n'y a pas de mécanisme largement accepté permettant de déterminer si ce codeur peu partager les ressources de la plate-forme cible avec d'autres calculs.

Les processus et leur version allégée, les threads, sont les briques de base de la conception des logiciels parallèles. Un processus peut être vu comme un composant, et un système d'exploitation multi-tâches comme un environnement d'exécution qui coordonne ces composants et fournit les mécanismes d'interactions entre processus (moniteurs, sémaphores etc). L'interface d'un composant-processus est alors une séquence ordonnée d'actions externes.

Mais comme nous l'avons vu, les processus et les threads ne se prêtent pas bien à la composition : la composition parallèle de deux processus n'est en général pas un processus (elle ne présente pas *une* séquence ordonnée d'actions externes comme interface). Pire, cette composition n'est même pas un composant que l'on peu caractériser facilement. C'est pour cette raison que les programmes qui utilisent plusieurs processus ou threads sont si difficiles à concevoir. On ne peut parler de façon précise que des processus individuels, mais pas de leur composition, ce qui empêche de morceler la difficulté du programme complet.

Une autre voie à explorer est la conception par objets. La définition d'interfaces dans l'approche objet fonctionne bien grâce aux systèmes de typage qui la supportent. Les systèmes de typage contribuent en pratique de façon très importante à la qualité du logiciel. Les langages à objets, avec leurs types de données abstraits définissables par l'utilisateur, ainsi que les relations (héritage, polymorphisme) entre types, ont eu un énorme impact sur la réutilisabilité et la qualité du logiciel (il suffit de voir la bibliothèque de classes Java standard, et les nombreuses bibliothèques de classes et de templates C++). Combinés avec les design patterns et la modélisation par objets, les systèmes de typage fournissent un moyen d'aborder des structures logicielles autres que les procédures ou les lignes de code.

Toutefois, la programmation par objets concerne principalement la structure statique du logiciel. Elle concerne la syntaxe des programmes procéduraux et n'exprime rien sur leur dynamique ou leur parallélisme (même si des efforts sont faits en ce sens dans UML). Ainsi, le type d'un objet n'indique en rien que sa méthode `initialize()` doit être appelée avant sa méthode `fire()`. Les propriétés temporelles d'un objet (la méthode `x()` doit être appelée toutes les 10 ms) ne font pas partie du type de l'objet non plus. Pour que le logiciel enfoui puisse bénéficier d'une technique de composition, cette technique devra inclure les propriétés dynamiques dans la définition des interfaces.

2.3.2 Procédures et objets

Un mécanisme d'abstraction simple de ce type en matière de logiciel est la notion de procédure (ou de méthode en termes d'objets). Nous avons vu que les procédures, à cause de leur nature finie, ne sont pas bien adaptées à la description des interactions avec le monde physique (qui ne suit pas le modèle démarrage, exécution, terminaison, retour des résultats).

L'approche objet masque de plus la structure des données pour parvenir à l'abstraction des données en plus de l'abstraction des calculs. Toutefois, les objets sont des entités passives, qui ne rendent des services que sur invocation externe de leurs méthodes. La notion d'« objet actif » est quelque chose de conçu après coup, sans définir réellement un modèle de calcul avec une sémantique claire. C'est ainsi que malgré les nombreux services qu'elle a rendu dans la conception de grands systèmes logiciels, l'approche objet a peu à offrir pour régler les problèmes spécifiques du logiciel enfoui.

Une technologie à base de composants pour les systèmes enfouis doit plutôt considérer des processus que des fonctions. Mais elle doit permettre la composition de ces processus et être capable de prévoir leur comportement temps-réel.

2.3.3 La conception de matériel

La conception de matériel est beaucoup plus en prise avec le monde physique que la conception de logiciel, et il est intéressant de regarder quelles abstractions fonctionnent dans ce domaine. Par exemple, les systèmes synchrones permettent de concevoir des systèmes matériels de grande taille, complexe, et modulaires. Son application au logiciel a donné les langages synchrones, qui sont effectivement appropriés à la conception de systèmes enfouis.

Les modèles utilisés pour le matériel sont habituellement construits avec des langages de description comme VHDL ou Verilog. Ces langages fournissent un modèle à événements discrets du calcul, qui fait du temps un concept partagé par tous les composants. La conception synchrone est une façon particulière d'utiliser ces langages.

Du point de vue du calcul, la distinction entre matériel et logiciel se fait uniquement sur le degré de parallélisme et sur le rôle du temps. Ainsi, une application qui utilise beaucoup de parallélisme et est très liée au temps sera plus facile à exprimer en termes de matériel, même si son implémentation finale se fait en logiciel. Réciproquement, une application purement séquentielle sans contrainte de temps sera plus facile à exprimer en termes de logiciel, quelle que soit son implémentation finale.

2.3.4 Les systèmes d'exploitation temps-réel

La plupart des systèmes enfouis ont à effectuer des calculs en temps-réel, et certains ont des contraintes de temps dures, par exemple lorsqu'il s'agit de traiter des flux de données ou d'effectuer un traitement du signal. Dans les ordinateurs généralistes, ce type de fonctionnalités est traité par des périphériques spécialisés (carte son ou carte graphique). Dans les systèmes enfouis, ces tâches sont en compétition pour l'acquisition des ressources de la plate-forme. Dans les systèmes enfouis interconnectés en réseau, le problème devient plus difficile car la combinaison de tâches en compétition n'est pas connue à l'avance.

Beaucoup de ces systèmes utilisent un système d'exploitation temps-réel qui offre des services d'ordonnancement de tâches appropriés aux systèmes temps-réel en plus des services classiques d'un système d'exploitation, comme les entrées-sorties. L'ordonnancement peut s'appuyer sur des priorités ou sur la proximité d'une échéance. Le problème est que le choix des priorités fait souvent des hypothèses sur la périodicité des activations des tâches, ou bien impose d'itérer jusqu'à ce qu'une implémentation particulière fonctionne correctement, ce qui ne donne évidemment pas des systèmes robustes.

Le principal problème de l'ordonnancement est que la plupart des techniques ne supportent pas la composition. Même si certaines propriétés sont garanties pour un composant donné, il n'y a pas de méthode systématique pour garantir certaines propriétés de la composi-

tion de deux composants, sauf dans des cas triviaux. Un exemple typique des problèmes de l'ordonnancement à base de priorité est connu sous le nom d'« inversion de priorité ».

L'inversion de priorité se produit quand plusieurs processus interagissent, par exemple en utilisant un moniteur pour acquérir une ressource partagée. Supposons qu'un processus de priorité faible ait acquis la ressource et soit préempté par un processus de priorité moyenne. Ensuite, un processus de priorité élevée préempte le processus de priorité moyenne et tente d'acquérir la ressource. Il est alors bloqué puisque la ressource est utilisée par le processus de faible priorité, qui est lui même bloqué par le processus de priorité moyenne. Ainsi, le processus de priorité élevée ne peut pas s'exécuter tant que le processus de priorité moyenne n'a pas terminé et autorisé le processus de priorité faible à poursuivre son exécution et à libérer la ressource.

Bien qu'il existe des palliatifs à l'inversion de priorité (par exemple l'héritage de priorité, qui consiste à attribuer temporairement à une tâche qui détient une ressource la priorité la plus élevée des tâches qui attendent cette ressource), ce problème est le symptôme d'un défaut fondamental : dans les systèmes à base de priorités, les processus interagissent à la fois via l'ordonnanceur et via le mécanisme d'exclusion mutuelle (les moniteurs). Ces deux mécanismes d'interaction n'ont, mis ensemble, pas de sémantique cohérente pour la composition.

2.3.5 Les modèles objet temps-réel

La pratique du temps-réel a été étendue au logiciel distribué à composants sous la forme de CORBA temps-réel et de la modélisation temps-réel à objets (*Real-time Object-Oriented Modeling*). CORBA (*Common Object Request Broker Architecture*) est une approche à objets distribuée qui s'appuie sur les appels de procédure distants (*Remote Procedure Calls*). CORBA propose d'autres services, parmi lesquels un service d'événements qui s'appuie sur une sémantique de publication-souscription (*publish-subscribe*). CORBA temps-réel étend CORBA en associant des priorités au traitement des événements et en permettant l'ordonnancement des tâches de façon à traiter les événements à temps. CORBA temps-réel s'appuie toutefois sur les mêmes abstractions qui ont montré leur faiblesse pour le logiciel embarqué. Le programmeur doit par exemple spécifier le temps d'exécution des procédures dans le pire des cas, selon que la procédure est en cache ou non. Or ce type d'information est difficile à connaître de manière exacte. L'ordonnancement temps-réel est ensuite effectué selon divers paramètres comme la périodicité, et ajusté en fonction de paramètres dont le sens est moins précis comme l'« importance » et la « criticité ». Globalement, ces paramètres sont des estimations qui ne permettent pas de déterminer le comportement du système autrement qu'en expérimentant.

Chapitre 3

Modélisation par acteurs

3.1 Origine des acteurs

La notion d'acteur a été introduite dans les années 1970 par Carl Hewitt au MIT pour décrire le concept d'agent raisonnant autonome [1]. Gul Agha a ensuite fait évoluer ce terme pour décrire un modèle formel de parallélisme dans lequel les acteurs ont chacun un flot de contrôle (*thread*) indépendant et communiquent par échange asynchrone de messages. La notion d'acteur que nous utilisons ici est liée à celle d'Agha, mais si les acteurs sont concurrents, ils ne disposent pas nécessairement de leur propre flot de contrôle, et s'ils communiquent par échange de messages, ce n'est pas nécessairement de manière asynchrone.

Les composants d'un modèle ont une interface qui masque leur état interne et leur comportement. Cette interface restreint la façon dont un composant peut interagir avec son environnement. Un composant peut donc être considéré comme un objet. Pour traiter le parallélisme, il est plus pratique de doter chaque composant d'une activité propre, concurrente de celle des autres composants. On parle alors d'*acteurs*.

Dans une modélisation par objets, les composants interagissent en transférant le contrôle grâce à des appels de méthodes. Dans une modélisation par acteurs, ils interagissent en s'échangeant des messages grâce à des canaux de communication. Les acteurs interagissent uniquement avec les canaux auxquels ils sont connectés, jamais directement avec d'autres acteurs.

L'interface d'un acteur comporte des ports, qui représentent les points de communication entre l'acteur et son environnement, et des paramètres qui permettent de configurer le comportement de l'acteur. Ces paramètres peuvent aussi évoluer, et faire partie de l'état observable de l'acteur.

De même qu'un acteur définit une interface, un modèle peut définir une interface qui devient alors une abstraction hiérarchique du modèle. Cette interface a des ports et des paramètres distincts de ceux des acteurs du modèle. Les ports externes peuvent être connectés à d'autres ports externes du modèle ou à des ports d'acteurs qui composent le modèle. Les paramètres externes peuvent être utilisés pour calculer les valeurs des paramètres des acteurs du modèle.

La modélisation par acteurs s'est répandue par l'utilisation d'outils comme Simulink (The Mathworks), Labview (National Instruments), SPW (*the Signal Processing Worksystem* de Cadence), ou Cocentric System Studio (Synospsys). Elle gagne en légitimité grâce aux efforts de l'OMG dans UML-2. Les langages de conception de matériel, comme VHDL, Verilog et SystemC, s'appuient tous sur les acteurs.

Le projet Ptolemy [2] offre la plus grande variété de modèles de calcul pour la modélisation par acteurs.

Gul Agha avance qu'aucun modèle du parallélisme ne peut ou ne doit autoriser l'expression de toutes les abstractions de la notion de communication. Au lieu de cela, les acteurs doivent se composer selon une méthode d'interaction, et chaque méthode d'interaction particulière correspond à un modèle de calcul. Agha considère le passage de message comme l'équivalent

1. "Viewing control structures as patterns of passing messages", *Journal of Artificial Intelligence*, 8(3):323–363, June 1977.

2. http ://ptolemy.eecs.berkeley.edu

du `goto` du fait de son manque de structure, mais suppose que l'on peut construire les autres modes d'interaction sur le passage de message.

3.2 Syntaxe abstraite, syntaxe concrète

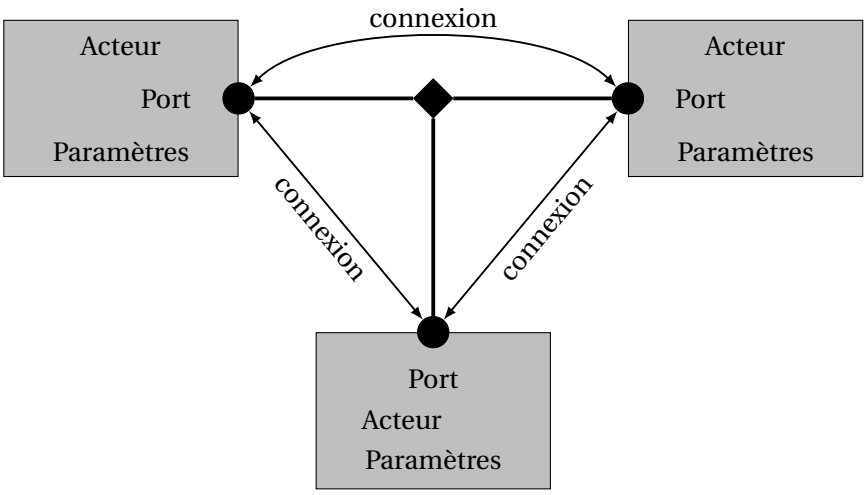


FIGURE 3.1 – Exemple de syntaxe abstraite

Les notions de modèle, d'acteur, de port, de paramètre et de canal constituent la syntaxe abstraite de la modélisation par acteurs. Cette syntaxe peut être représentée concrètement de différentes façons : graphiquement, au format XML ou encore sous la forme d'un programme utilisant une API donnée (par exemple en SystemC).

Une syntaxe abstraite détermine comment un modèle peut être décomposé en composants interconnectés, sans se soucier de la façon dont ce modèle est représenté sur papier ou dans un fichier (ce qui est le rôle de la syntaxe concrète), et sans se préoccuper de la signification des connections entre composants, ou même de ce qu'est un composant.

Les représentations graphiques d'un modèle sont les plus faciles à appréhender par un être humain. Ces syntaxes visuelles peuvent elle-mêmes être très variées, allant du schéma-bloc au diagramme états-transitions.

La modélisation par acteurs n'exige pas de syntaxe visuelle et peut s'effectuer selon des syntaxes purement textuelles. D'ailleurs, les schémas sont souvent remplacés par du texte dans les langages de description de matériel tels que VHDL et Verilog.

La structure syntaxique d'un modèle à acteurs donne très peu d'information sur sa sémantique qui est relativement orthogonale à la syntaxe et est déterminée par le modèle de calcul utilisé. Le modèle de calcul donne les règles d'exécution du modèle : quand les acteurs effectuent-ils le calcul de leur comportement interne, quand mettent-ils à jour leur état interne, quand communiquent-ils. Le modèle de calcul détermine aussi la nature des communications entre acteurs.

3.3 Sémantique

La sémantique est le sens que l'on donne aux composants et à leurs interactions. On peut par exemple considérer qu'un composant est un processus, et que les connexions entre composants représentent les communications entre ces processus. On peut aussi considérer qu'un composant est un état, et que les connexions entre composants représentent les transitions entre états. Ces modèles sémantiques sont appelés *modèles de calcul* (*Models of Computation*).

Considérons par exemple une famille de modèles de calcul dans lesquels les composants sont des producteurs ou des consommateurs de données. Dans ce contexte, les ports sont considérés soit comme des entrées, soit comme des sorties.

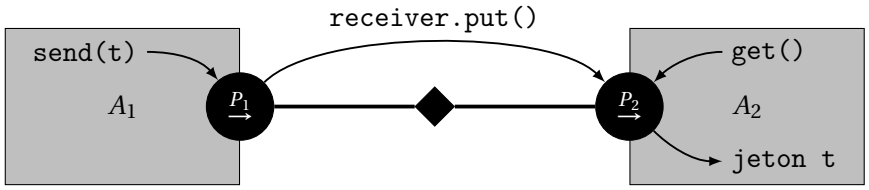


FIGURE 3.2 – Système consommateur-producteur

Sur le diagramme de la figure 3.2, deux acteurs s'échangent des données. Lorsque le producteur veut envoyer une donnée au consommateur, il envoie un jeton (qui encapsule la donnée) en invoquant la méthode `send` de son port de sortie. Ceci invoque la méthode `put` du port de destination. Le composant consommateur récupère la donnée en invoquant la méthode `get` de ce même port. Ce mécanisme est complètement polymorphe, au sens où le comportement des méthodes `put` et `get` est totalement non spécifié et dépend du modèle de calcul.

Pour la plupart des utilisateurs d'outils de conception, le problème de la sémantique du modèle de calcul ne se pose pas. Pour eux, Simulink n'a pas de sémantique, il est « comme ça ». Un des enjeux de la modélisation des systèmes enfouis est d'inventer ou d'identifier les modèles de calcul qui ont des propriétés adaptées au système à modéliser.

Chapitre 4

Modèles de calcul

4.1 Modèles de calcul et conception

Un modèle de calcul peut être considéré comme l'ensemble des « lois de la physique » qui gouvernent les interactions entre composants. Nous avons vu que les modèles de calcul utiles pour la conception de systèmes enfouis doivent supporter le parallélisme, et qu'il n'existe malheureusement pas d'équivalent pour le parallélisme de ce qu'est le modèle de von Neumann pour le code séquentiel. Le modèle de von Neumann s'appuie sur la notion d'état global du système, état qui est modifié par chacune des actions entreprises séquentiellement.

Dans un système parallèle, il est difficile de définir et de manipuler une notion d'état global à cause des nombreuses modifications partielles qui ont lieu simultanément.

Une approche classique des modèles de calcul consiste à étendre ou à définir un nouveau langage pour définir un modèle de calcul. Le problème de cette approche est qu'elle oblige à réécrire le code existant et/ou à concevoir tout le système avec l'unique modèle supporté par le langage.

Une autre approche consiste à utiliser des modèles de calcul pour coordonner des programmes écrits dans des langages classiques. C'est ce que fait le langage de spécification de systèmes SystemC. Cette approche permet de découpler le choix du langage de programmation et celui du modèle de calcul. Il devient ainsi possible de concevoir un système dont certaines parties sont écrites en VHDL pour pouvoir être implémentées en matériel, d'autres en Java de façon à pouvoir être réutilisées telles qu'elles sur d'autres plateformes, et d'autres enfin en C de façon à obtenir de meilleures performances.

On peut ainsi considérer les modèles de calcul comme des Design Patterns plutôt que comme des caractéristiques de langages de programmation. La modélisation d'un système peut faire appel à différents modèles de calcul, de même qu'on peut utiliser plusieurs design patterns dans un même programme.

Il existe de nombreux modèles de calcul, chacun traitant le parallélisme et le temps de diverses façon. Nous présentons ici quelques uns des modèles de calcul les plus significatifs.

4.2 Modèles à flots de données

Dans les modèles à flots de données (*data flow*), les acteurs sont des opérateurs qui effectuent un calcul atomique déclenché par la disponibilité des données. Les connexions entre acteurs représentent le flot des données d'un producteur à un consommateur. SPW (*Signal Processing Worksystem*) de Cadence, et LabVIEW de National Instruments sont des exemples d'outils de modélisation commerciaux qui utilisent des modèles à flots de données.

Le modèle à flots de données synchrones (*Synchronous Data Flow*) est une restriction des modèles à flots de données. Son intérêt est que dans ce modèle, la vivacité du système et la finitude des tampons de communication sont des propriétés décidables. Un modèle SDF peut être ordonnancé statiquement, ce qui facilite la génération de code et donc le passage du modèle à l'implémentation.

Dans le modèle SDF, chaque port consomme ou produit un nombre constant d'échantillons de données à chaque activation de l'acteur qui le possède. Chaque acteur est donc activé à chaque fois que tous ses ports d'entrée contiennent le nombre voulu d'échantillons de

données. Le problème de l'ordonnanceur est donc de trouver une séquence d'activations des différents acteurs qui laisse le nombre d'échantillons de données constant. On choisira une des séquences les plus courtes pour définir une itération du système.

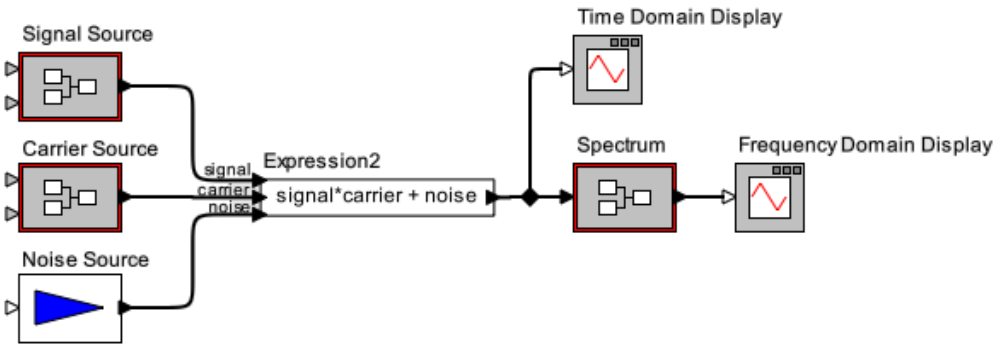


FIGURE 4.1 – Modèle SDF simple

La figure 4.1 montre un modèle qui utilise le modèle de calcul SDF pour estimer le spectre du signal obtenu en modulant une sinusoïde avec un autre signal sinusoïdal et en ajoutant du bruit.

Sur la gauche, on trouve de haut en bas : l'acteur qui génère le signal modulant, l'acteur qui génère la porteuse et l'acteur qui génère le bruit. Les sorties de ces 3 acteurs sont connectées à l'entrée d'un acteur `Expression` dont le comportement consiste à calculer l'expression qui lui est donnée en paramètre, en fonction de ses entrées. La sortie de `Expression` est reliée à un afficheur et à un acteur qui calcule le spectre de fréquence du signal, ce spectre étant lui-même fourni à un dernier acteur qui l'affiche.

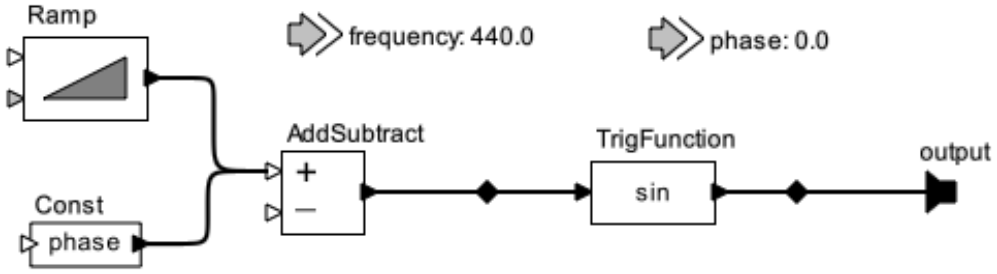


FIGURE 4.2 – Modèle interne des sinusoïdes

Dans ce modèle, les acteurs qui génèrent les sinusoïdes et l'acteur qui calcule le spectre d'un signal sont composites. Cela signifie que leur comportement est décrit sous forme d'un assemblage d'autres acteurs. Si l'on observe l'intérieur d'un acteur qui génère une sinusoïde, on obtient le modèle de la figure 4.2. Ce modèle construit un signal croissant régulièrement en fonction de la fréquence et de la phase de la sinusoïde. Il applique ensuite la fonction `sin` à ce signal pour obtenir la sinusoïde voulue.

Les systèmes que nous venons de voir sont simples car ils ne contiennent pas de boucle, et chacun des ports d'un acteur consomme ou produit une donnée à chaque fois que l'acteur est activé.

La figure 4.3 montre un modèle qui contient une boucle autour de l'acteur `SketchedSource`. Les dépendances entre ports montrent clairement que le générateur de sinusoïde doit être activé en premier afin que le multiplieur dispose de données. Mais le multiplieur dépend aussi de la sortie de `SketchedSource`, donc il faudrait aussi l'activer avant le multiplieur. Toutefois, d'après les connexions entre ports, `SketchedSource` dépend lui-même de la sortie du multiplieur et doit donc être activé après lui. Tel quel, ce réseau d'acteurs devrait donc être rejeté puisqu'aucun ordre d'activation n'est compatible avec les dépendances entre ports.

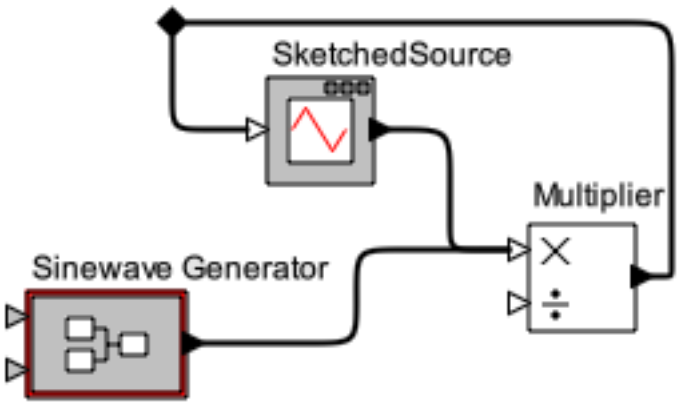


FIGURE 4.3 – Modèle SDF avec boucle

Il existe une propriété du port de sortie de l'acteur `SketchedSource` qui permet à l'ordonnanceur d'accepter ce modèle. Cette propriété stipule que le port contient une valeur initiale, qui peut donc être fournie avant même que l'acteur ait réagit. L'ordre retenu pour l'activation des acteurs de ce modèle sera donc : sinusoïde, multiplieur, puis `SketchedSource`, ce dernier ayant besoin de la valeur de sortie du multiplieur pour l'afficher.

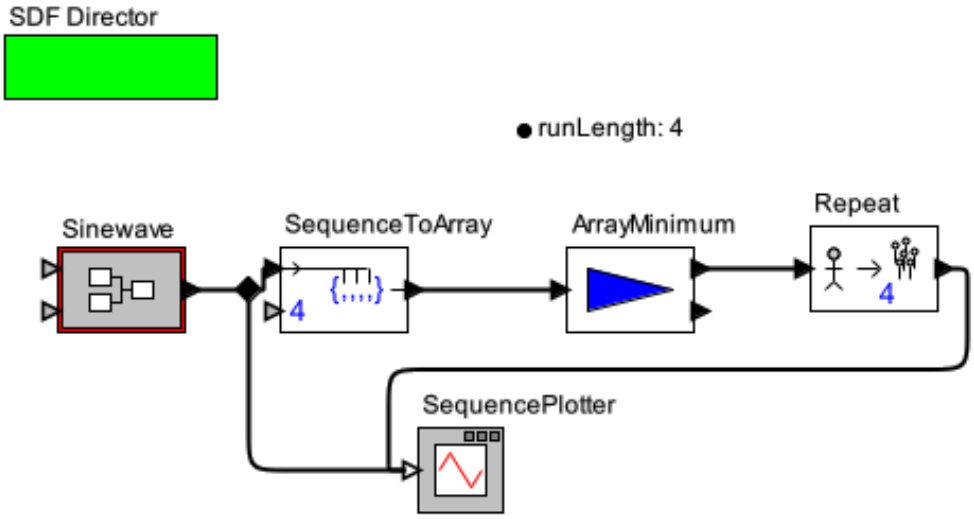


FIGURE 4.4 – Modèle SDF à taux multiples

Dans les modèles SDF que nous venons d'étudier, tous les acteurs consomment et produisent un échantillon sur chaque port à chaque activation. Il existe des modèles dans lesquels certains ports consomment ou produisent plus d'un échantillon, on les appelle des modèles à taux multiples (*multi rate models*).

Dans le modèle de la figure 4.4, une sinusoïde est affichée (pour servir de référence), et est aussi découpée en paquets d'échantillons. Pour chaque paquet, on prend la valeur minimale que l'on répète autant de fois qu'il y a d'échantillons dans le paquet, et on affiche le résultat pour le comparer au signal d'origine.

Lorsque la taille des paquets est 1, les deux signaux sont identiques et tous les acteurs sont activés une fois par itération du système. Si l'on choisit une taille de paquets de p échantillons, la sinusoïde devra avoir produit p échantillons avant que l'on puisse activer `SequenceToArray`. Ce dernier produira alors un échantillon contenant les p échantillons qu'il vient de recevoir. Dans cet échantillon, `ArrayMinimum` choisira la valeur minimale et la produira en sortie. Enfin, `Repeat` répétera cette valeur r fois pour la fournir à l'afficheur. La figure 4.5 montre ce que

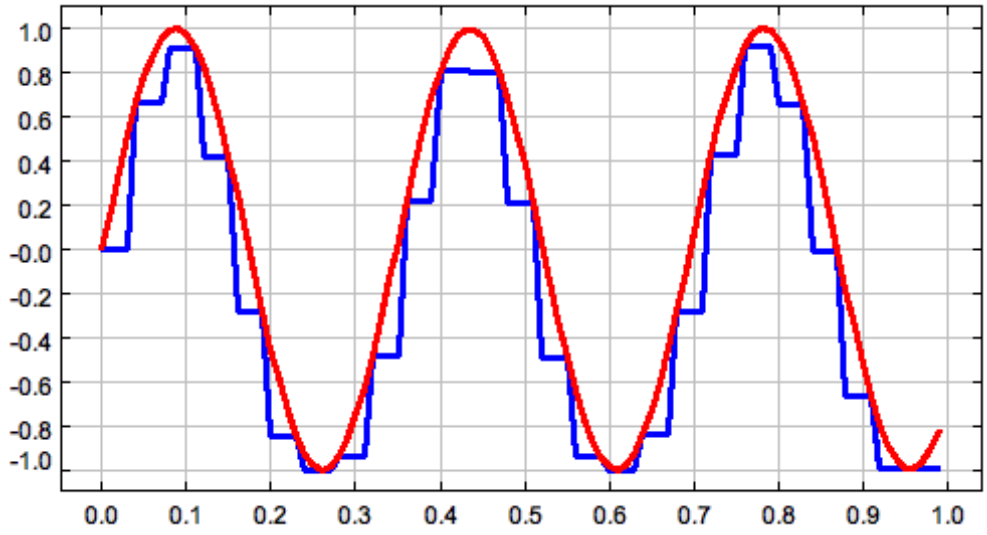


FIGURE 4.5 – Sortie du modèle SDF de la figure 4.4

produit ce modèle sur 25 itérations pour une sinusoïde à 230 Hz échantillonnée à 8 kHz et un paramètre `runLength` égal à 4 (le nombre d'échantillons produits est donc 100).

Si l'on note n_{sin} le nombre d'activation de la sinusoïde, n_{array} celui de `SequenceToArray`, n_{min} celui de `ArrayMinimum`, n_{repeat} celui de `Repeat`, et n_{plot} celui de l'afficheur, on a :

$$\begin{aligned} n_{\text{sin}} &= p \times n_{\text{array}} \\ n_{\text{array}} &= n_{\text{min}} \\ n_{\text{min}} &= n_{\text{repeat}} \\ r \times n_{\text{repeat}} &= n_{\text{plot}} \\ n_{\text{sin}} &= n_{\text{plot}} \end{aligned}$$

On en déduit que :

$$\begin{aligned} n_{\text{sin}} &= n_{\text{plot}} \\ &= p \times n_{\text{narray}} \\ &= r \times n_{\text{repeat}} \end{aligned}$$

$$\begin{aligned} n_{\text{array}} &= n_{\text{min}} \\ &= n_{\text{repeat}} \end{aligned}$$

Ce système ne peut avoir de solution que si $p = r$, et dans ce cas, à chaque itération du système, la sinusoïde sera activée p fois pour produire p échantillons, puis `SequenceToArray` sera activé et consommera les p échantillons présent sur son port d'entrée et produira un échantillon vectoriel sur sa sortie. `ArrayMinimum` sera alors activé, consommera l'échantillon vectoriel présent sur son entrée et produira la valeur minimale de ce vecteur sur sa sortie. `Repeat` sera lui aussi activé, consommera l'échantillon présent sur son entrée et produira p échantillons identiques sur sa sortie (car $p = r$). Enfin, l'afficheur sera activé et consommera p échantillons sur chacune des voies de son port d'entrée (il s'agit en effet d'un port multiple, qui se comporte de façon analogue à un bus).

Le nombre d'échantillons de signal produits par le système dépend donc de la valeur de p puisqu'à chaque itération, le système produit p échantillons. Les connexions entre la sinusoïde et l'afficheur et entre `Repeat` et l'afficheur (ou les ports d'entrée de l'afficheur) doivent être capable de stocker p échantillons en attente de traitement. Il est donc possible

de dimensionner les tampons de communication et d'ordonnancer le système uniquement d'après sa topologie, ce qui facilite la génération de code efficace.

Pour être complet, notons qu'en plus de l'échantillon de données courant, il est possible d'accéder à un nombre borné d'échantillons précédents, ce qui permet d'exprimer des comportements sous forme d'équations aux différences finies. Cette possibilité doit être prise en compte dans le dimensionnement des tampons de communication.

4.2.1 Les réseaux de processus

Une manière simple de gérer le parallélisme est de considérer les composants comme des processus (ou des threads) qui communiquent par passage asynchrone de messages via des files. Les files permettent à un émetteur de ne pas avoir à attendre que le destinataire soit prêt à recevoir le message.

Les réseaux de processus de Kahn (KPN, pour *Kahn Process Networks*) sont un cas particulier de réseaux de processus dans lesquels un processus ne peut pas savoir si une file de message est vide ou non, et est bloqué en attente lorsqu'il cherche à obtenir un message à partir d'une file vide. Ces restrictions garantissent que le réseau est déterministe, c'est-à-dire que ce qu'il calcule ne dépend que de sa topologie et est indépendant de l'implémentation qui en est faite.

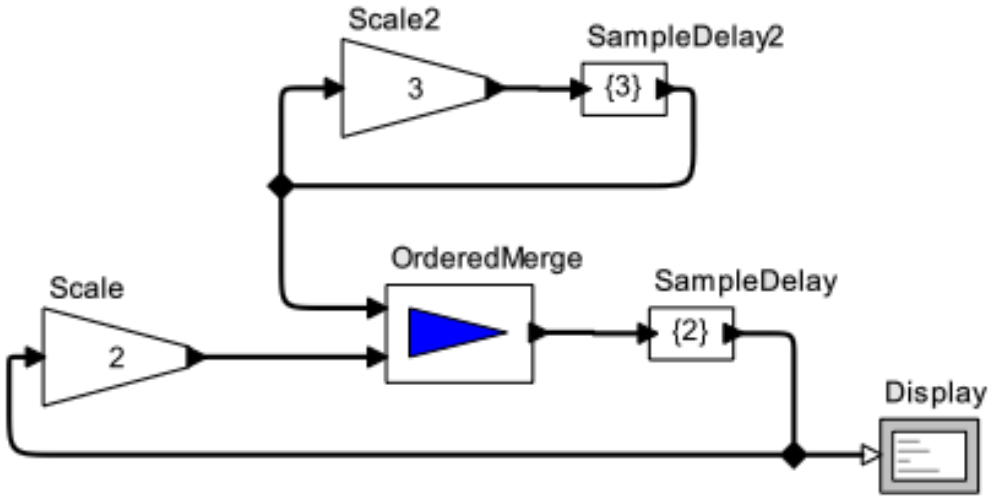


FIGURE 4.6 – Exemple de modèle PN

On peut considérer les modèles de calcul à flots de données comme un cas particulier des réseaux de processus dans lequel les processus sont construits comme des séquences d'activations atomiques des acteurs.

Les réseaux de processus sont particulièrement bien adaptés au traitement du signal. Leur couplage est faible, un processus n'étant bloqué que lorsque les données qu'il doit traiter ne sont pas disponibles, ce qui facilite leur parallélisation et leur répartition. Leur principal défaut est qu'ils sont mal adaptés à la spécification de logiques de contrôle complexes, la logique de contrôle y étant limitée au routage de données.

La figure 4.6 montre un modèle PN simple qui calcule, en ordre croissant, tous les entiers qui sont le produit d'une puissance de 2 et d'une puissance de 3.

La boucle du haut produit les puissances de 3 à partir de 3, et la boucle du bas produit les puissances de 2 et leur produit par les puissances de 3 reçues de la boucle du haut. L'acteur `OrderedMerge` insère les puissances de 3 à leur place dans la séquence d'entiers produite par la boucle inférieure.

Le fonctionnement de ce réseau est infini, mais différents ordonnancement des processus sont possibles. Certains permettent de garantir une taille bornée pour les files de communication, d'autres non. On peut en effet faire fonctionner la boucle supérieure indéfiniment avant

de traiter l'infinité de puissances de 3 dans la boucle inférieure, ce qui nécessite une file infinie pour l'entrée supérieure du `OrderedMerge`. Dans la pratique, on préférera ne produire que les données nécessaires pour éviter le blocage d'autres processus, c'est-à-dire n'activer la boucle du haut que lorsque l'acteur `OrderedMerge` est en attente de données. Dans le cas général, il n'est pas toujours facile d'assurer qu'un réseau qui peut s'exécuter avec des files de taille bornée le fera.

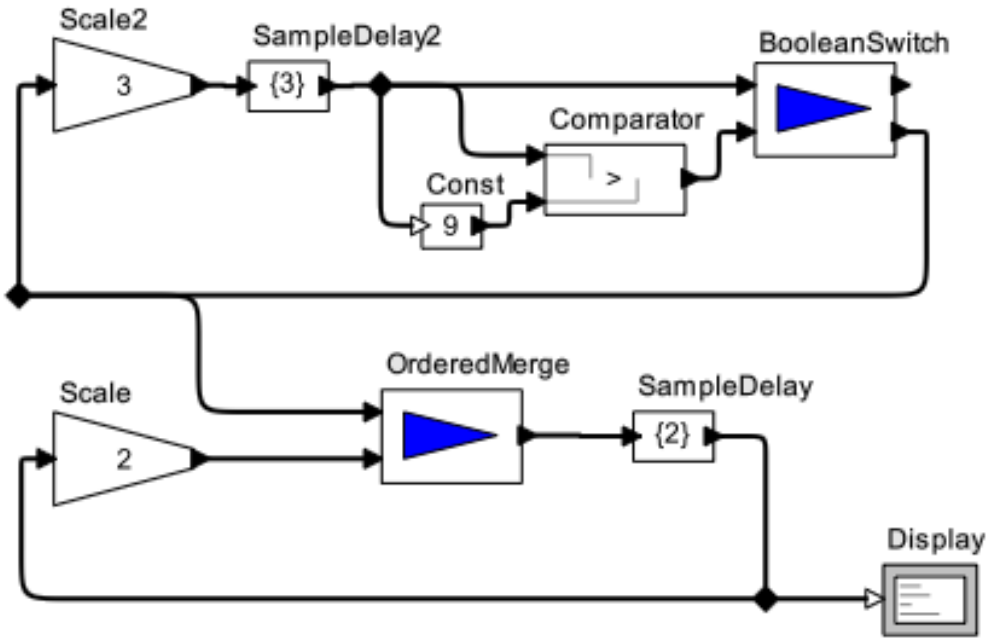


FIGURE 4.7 – Exemple de modèle PN

Pour illustrer les possibilités limitées de spécification du contrôle dans les réseaux de processus, la figure 4.7 montre un modèle qui limite le calcul des multiples de puissances de 2 et de 3 à la valeur maximale 9. Pour cela, la boucle supérieure est ouverte dès qu'un entier supérieur strict à 9 est va être produit. Le multiplieur par 3 ne recevant plus de données, il est bloqué. De même, l'acteur `OrderedMerge` ne recevant plus rien sur son entrée supérieure, se bloque. Tous les acteurs étant bloqués, aucune donnée ne pourra plus être produite et le système termine.

La table suivante montre la sortie du système sur la première ligne, et les entrées supérieures et inférieures de l'acteur `OrderedMerge` sur la deuxième et la troisième ligne. Lorsque plusieurs données sont disponibles, elles sont séparées par une virgule. Lorsqu'aucune ne sera plus disponible, le symbole $\perp$ est utilisé :

2	3	4	6	8	9	$\perp$
	3, 9	9	9	9	9	$\perp$
	4	4, 6	6, 8	8, 12	12, 16	12, 16, 18

Au départ, la valeur 2 est présente puisqu'elle est fournie par le retard. La valeur 4 devient alors présente sur l'entrée basse de l'`OrderedMerge`. D'autre part, la boucle du haut produit 3 et 9 avant de s'ouvrir et de se bloquer. Ces deux valeurs sont donc disponibles sur l'entrée haute du `OrderedMerge` qui doit donc choisir entre 3 et 4. Il choisit la plus petite valeur, 3, qui est donc produite en sortie, et génère un 6 sur l'entrée basse. On a alors le choix entre 9 et 4, donc 4 est produit, et 8 est ajouté à la file de communication de l'entrée basse etc. Lorsque la donnée 9 est consommée, un 9 est produit en sortie, et 18 est ajouté à la file d'attente de l'entrée basse du `OrderedMerge`. Comme l'entrée supérieure n'est plus connectée à rien (le `BooleanSwitch` est orienté vers le haut), aucune donnée ne parviendra plus sur ce port, et le système complet est bloqué.

4.3 Les modèles déclenchés par le temps

Les systèmes déclenchés par le temps (*Time-triggered*) sont pilotés par des horloges. Une horloge est un signal qui se répète indéfiniment à intervalles réguliers. Ce type d'activation régulière permet d'ordonnancer statiquement les calculs et les communications entre composants répartis. Pour modéliser du matériel, des simulateurs simulent des calculs qui se produisent à chaque tic d'horloge, ce qui correspond bien au fonctionnement synchrone du matériel. Dans d'autres systèmes, chaque composant est un processus qui se met en attente du tic d'horloge ou d'une condition sur ses entrées (qui sont synchrones avec les tics). Ce type d'approche a donné SystemC, un langage de spécification pour la conception de matériel au niveau système.

4.3.1 Le modèle réactif synchrone

Dans le modèle réactif synchrone (SR, pour *Synchronous Reactive*), les connexions entre composants représentent des valeurs qui sont alignées sur les tics d'une horloge globale, comme pour les systèmes pilotés par le temps. Toutefois, le modèle SR ne suppose pas que tous les signaux ont une valeur à chaque tic de l'horloge, ni que les tics sont régulièrement espacés dans le temps. Ceci permet de bien traiter les systèmes dans lesquels les événements surviennent de façon irrégulière. Chaque tic de l'horloge correspond à un instant où le système doit réagir à son environnement. Cette horloge est déterminée de manière externe au modèle synchrone. Dans la pratique, elle peut être déterminée par un gestionnaire d'interruptions ou être le résultat d'un algorithme plus complexe qui détermine quand le système doit réagir en fonction de l'évolution de son environnement.

Les acteurs représentent des relations entre leurs entrées et leurs sorties à chaque tic. Comme toutes les entrées ne sont pas nécessairement connues, ces relations sont en général partielles (sinon, on dit que l'acteur est strict, c'est-à-dire qu'il doit connaître toutes ses entrées pour calculer une quelconque de ses sorties), avec des restrictions qui garantissent le déterminisme. Des langages comme Lustre, Esterel et Signal utilisent ce modèle, et des techniques de compilation sophistiquées permettent d'éliminer le parallélisme pour générer du code purement séquentiel.

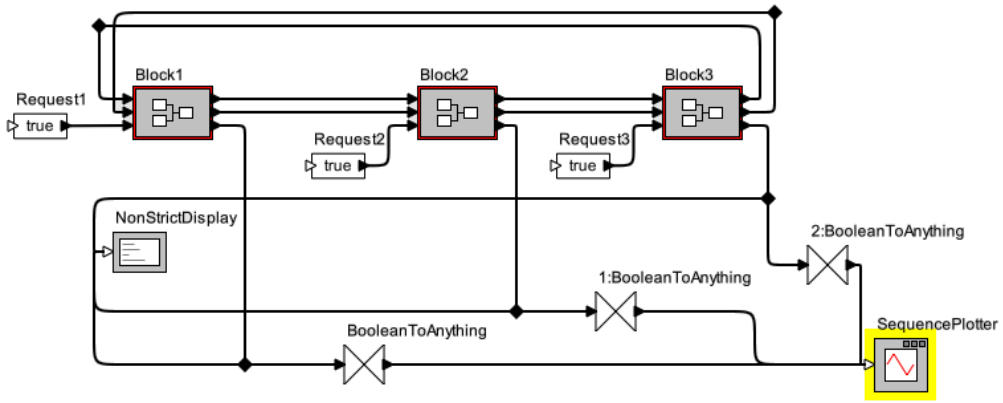


FIGURE 4.8 – Modèle SR d'arbitrage à jeton

La figure 4.8 montre un modèle d'un système d'arbitrage à jeton qui autorise l'accès à une ressource à un seul bloc à la fois, mais de façon équitable.

La figure 4.9 montre le modèle interne d'un arbitre. L'entrée `R` indique une requête d'accès à la ressource, et la sortie `G` (*grant*) indique que la ressource est allouée à cet arbitre pour cet instant. `PI` et `PO` servent à propager le jeton au travers des arbitres qui n'ont pas de requête d'accès à la ressource, et `TI` et `TO` (*Token In* et *Token Out*) font circuler le jeton entre les arbitres de façon à obtenir un arbitrage équitable.

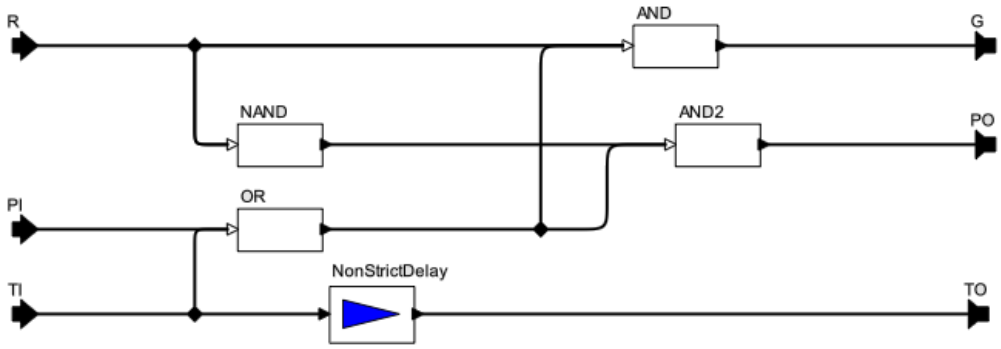


FIGURE 4.9 – Modèle SR d'un arbitre

Le schéma global comporte des boucles entre les blocs, et le modèle interne d'un arbitre ne contient pas de retard susceptible de briser cette boucle. Le système fonctionne car cette boucle est causale, c'est-à-dire qu'il est possible de trouver une et une seule valeur des données qui vérifie les relations imposées par les acteurs. De plus, cette valeur peut être construite par recherche de point fixe (l'assurance de son existence n'est pas une garantie de sa constructibilité).

Dans notre exemple, le jeton est initialement dans le premier arbitre. L'acteur `NonStrictDelay` de cet arbitre va donc fournir le jeton sur sa sortie `TO`, sans avoir besoin de connaître son entrée (c'est pour cela qu'il est dit « non strict »). Le deuxième arbitre va donc recevoir `true` sur son entrée `PI`, ce qui permet à sa porte `OR` de produire `true` sans même connaître son autre entrée. L'entrée `R` étant à `true`, on peut alors calculer `G` qui vaut `true` et `PO` qui vaut `false`. `TO` vaut `false` puisque cet arbitre ne possédait pas le jeton initialement, mais il le possède maintenant puisque `TI` valait `true` à cet instant.

Pour calculer le comportement des acteurs dans ce modèle de calcul, on part donc d'un état où les données disponibles sur les ports sont inconnues. On fait réagir les acteurs à cet état, et certains vont être capable de produire des données sur leurs sorties. On arrive ainsi à un état où certaines données sont connues, et on itère le procédé jusqu'à ce que toutes les données soient connues. Le comportement du système est ainsi défini par le point fixe de cette itération. Ce point fixe existe si tous les acteurs ont un comportement strictement monotone : si on les fait réagir avec plus de données connues, il déterminent strictement plus de données, et lorsque la valeur d'une donnée à été déterminée pour un instant, les activations suivantes de l'acteur au même instant ne changent pas cette donnée.

4.3.2 Les modèles à événements discrets

Dans les modèles de calcul à événements discrets, chaque événement a une valeur et une étiquette temporelle. Les connexions entre acteurs représentent des suites d'événements positionnés sur une ligne temporelle globale. Les acteurs sont des processus qui traitent les événements et en produisent. Contrairement au modèle réactif synchrone, les événements produits en réaction à un événement ne le sont pas nécessairement au même instant. De plus, chaque événement ayant une étiquette temporelle, le temps séparant deux instants a une signification, alors que dans l'approche réactive synchrone, il n'existe rien entre deux instants de l'horloge d'activation.

Les modèles à événements discrets (DE, pour *Discrete Events*), sont utilisés pour la modélisation de réseaux de communication et pour la modélisation de matériel. Des langages comme VHDL ou Verilog ont un modèle de calcul à événements discrets.

La limite des modèles à événements discrets est la notion de ligne de temps globale. Cette notion sur-spécifie les systèmes pour lesquels maintenir une horloge globale est difficile, par exemple les circuits VLSI de grande taille ayant des horloges très rapides, ou les environnement

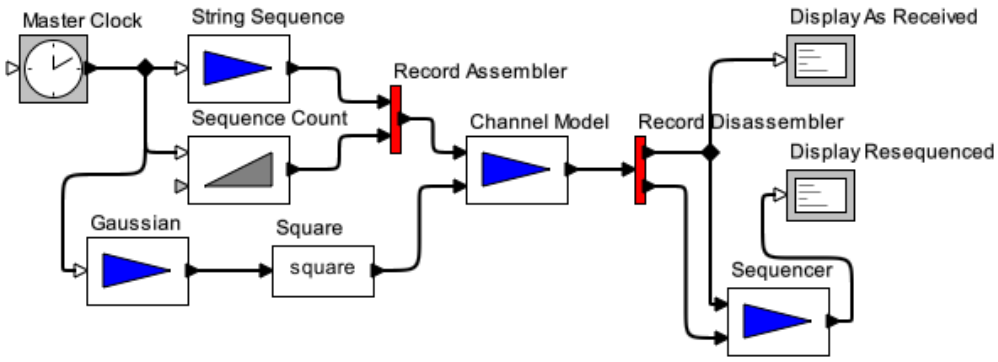


FIGURE 4.10 – Modèle DE d'un canal de transmission

répartis sur un réseau. Un autre gros défaut est que ce modèle est relativement coûteux à implémenter en logiciel, et que les simulateurs n'ont donc pas de très bonnes performances.

La figure 4.10 montre un modèle DE d'un canal de communication qui transmet une séquence de chaînes de caractères avec des délais aléatoires selon une distribution de Rayleigh. Les chaînes de caractères n'apparaissent donc pas nécessairement dans le bon ordre à la sortie du canal. Pour remettre les chaînes en ordre, on les étiquette avec le numéro dans la séquence (acteur `Record Assembler`), puis, à la sortie du canal, on utilise le numéro dans la séquence pour sortir les chaînes dans le bon ordre (acteur `Sequencer`).

4.3.3 Communication par rendez-vous

La communication par rendez-vous est un mode de communication synchrone entre processus. Lorsque deux processus communiquent, ils le font de façon atomique en une action instantanée appelée « rendez-vous ». Contrairement à la communication asynchrone des réseaux de processus, la communication par rendez-vous synchronise les processus car l'émetteur est lui aussi bloqué jusqu'à ce que le récepteur arrive au point de rendez-vous. Deux exemples classiques de modèles à rendez-vous sont CSP (*Communicating Sequential Processes*) de Hoare, et CCS (*Calculus of Communicating Systems*) de Milner. Ce modèle de calcul est celui utilisé par des langages comme Lotos et Occam.

Les modèles à rendez-vous sont adaptés à la modélisation de systèmes où le problème principal est le partage de ressources (clients et serveurs de bases de données, multiplexage de ressources matérielles). Leur principal inconvénient est qu'il est difficile d'obtenir des modèles déterministes en utilisant ces modèles de calcul.

Les modèles à rendez-vous et les réseaux de processus n'ont pas de notion globale du temps (c'est d'ailleurs ce qui les rend utiles pour la modélisation des systèmes répartis). Les messages échangés par les processus ne sont que partiellement ordonnés, alors qu'ils le seraient totalement s'ils étaient placés sur une ligne de temps globale. Ceci peut être gênant lorsqu'on veut utiliser ces modèles conjointement à des modèles qui ont une notion globale du temps. Des extensions des modèles à rendez-vous et des réseaux de processus permettent de gérer un temps global.

4.4 Modèles à temps continu

Les systèmes physiques peuvent en général être modélisés par un jeu d'équations différentielles. Un modèle à temps continu considère les composants comme des relations entre des fonctions du temps continu, qui sont elles-mêmes représentées par les connexions entre les composants. Un modèle exécutable doit fournir une procédure de calcul de ces fonctions, en général par intégration numérique selon divers algorithmes.

Les modèles à temps continu sont particulièrement utiles pour modéliser l'environnement avec lequel un logiciel enfoui va interagir. La modélisation conjointe du logiciel et de son

environnement permet de mieux appréhender le fonctionnement du système complet et est supportée par des environnements de modélisation par acteurs comme Simulink, Saber, VHDL-AMS et Ptolemy II.

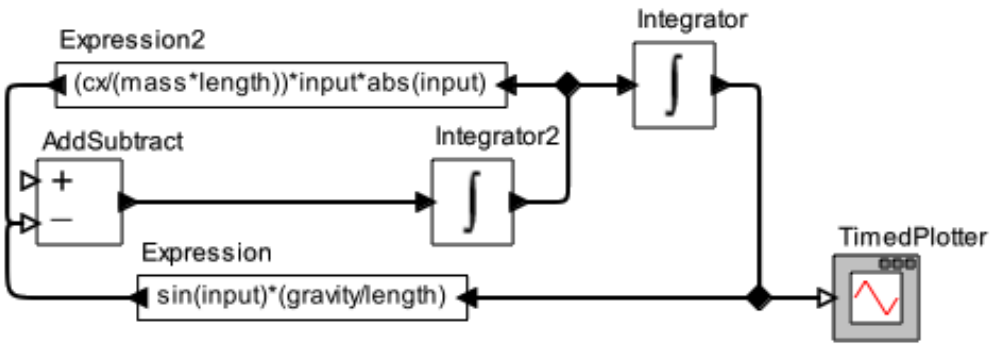


FIGURE 4.11 – Exemple de modèle CT

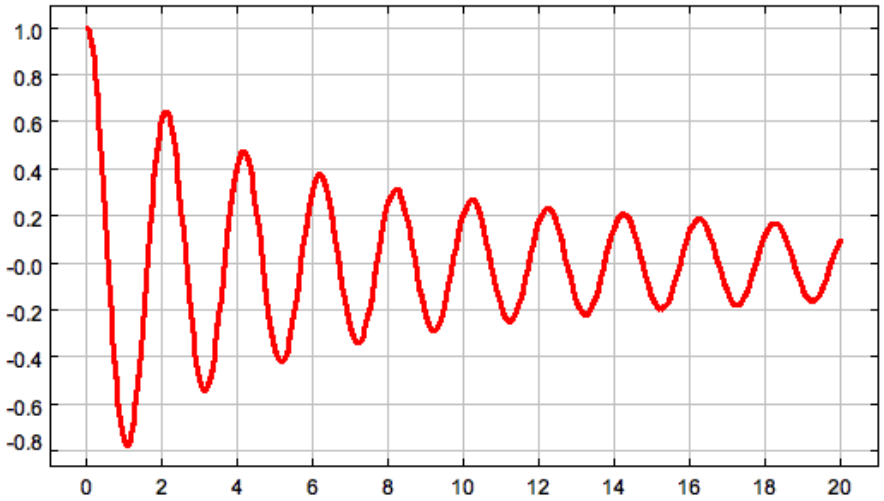


FIGURE 4.12 – Évolution de l'angle du pendule avec le temps

La figure 4.11 montre un modèle à temps continu d'un pendule. Ce système est régit par l'équation :

$$\ddot{\theta} = -\frac{g}{l}\sin\theta - \frac{C_x}{ml}\dot{\theta}|\dot{\theta}|$$

qui est exprimée graphiquement dans le modèle à l'aide d'acteurs « intégrateurs » et d'acteurs « expression mathématique ». Dans cette expression, θ est l'angle que fait le pendule avec la verticale, C_x est le coefficient de traînée aérodynamique du pendule, g est l'accélération de la pesanteur, m la masse du pendule, et l sa longueur. L'évolution du système avec un angle initial de 1 radian et une vitesse nulle est donnée sur la figure 4.12.

Pour être utile pour la modélisation conjointe d'un système et de son environnement, un modèle de calcul à temps continu doit être capable de générer précisément des événements discrets qui correspondent aux acquisitions de données du système, ainsi que de transformer les événements discrets qui correspondent aux actions du système en signaux continus.

La résolution numérique d'un système d'équations différentielles vise en effet une certaine précision sur la valeur des fonctions calculées. La précision voulue détermine la taille du pas d'intégration. Mais lorsque le système est sensible au franchissement d'un seuil (par exemple si une interruption est générée lorsqu'un signal dépasse une certaine valeur), la date à laquelle l'événement se produit devient importante. En effet, si la fonction considérée varie peu, ou varie très régulièrement, l'intégrateur d'équations différentielles peut utiliser un pas

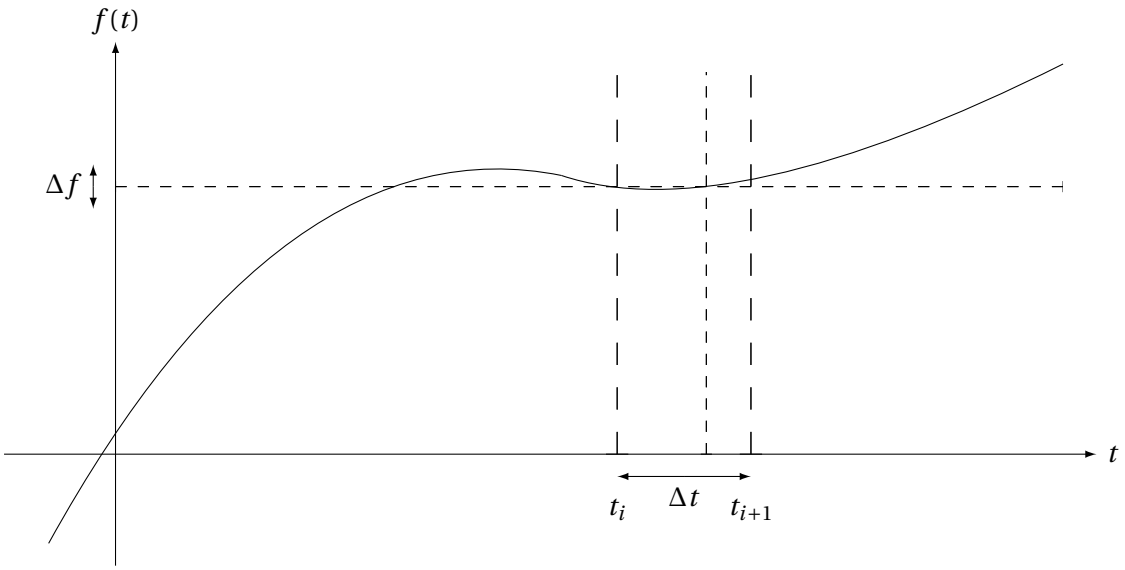


FIGURE 4.13 – Précision sur la valeur/sur la date

d'intégration grand pour obtenir une précision suffisante, ce donne une grande incertitude sur la date de l'événement.

Lorsqu'un événement est produit au cours d'un pas d'intégration, il est donc nécessaire de rejouer ce pas d'intégration en le divisant jusqu'à déterminer suffisamment précisément la date de l'événement.

Chapitre 5

Automates finis

5.1 Domaine d'application

Les automates, ou machines à états (*state machines*), permettent de modéliser explicitement l'espace d'état d'un système. Lorsque cet espace d'état est fini, on parle d'automates finis. Les automates sont particulièrement bien adaptés à la modélisation de la logique de commande, notamment pour les systèmes critiques. En effet, les automates se prêtent à des méthodes formelles d'analyse qui permettent de valider certaines propriétés. Enfin, il est facile d'implémenter un automate aussi bien en logiciel qu'en matériel.

Un inconvénient des automates est leur faible expressivité : il s'agit d'une modélisation d'assez bas niveau, et de petites modifications de la spécification d'un système peuvent entraîner de grands changements dans la structure d'un automate. Un autre inconvénient est que le nombre d'états d'un automate peut devenir très grand, même pour un système de complexité modérée.

En combinant les automates (qui sont purement séquentiels) avec des modèles qui supportent le parallélisme, on peut modéliser des systèmes complexes avec des automates ayant un nombre d'états gérable. Par exemple, les StateCharts, qui sont maintenant utilisés par UML pour modéliser les aspects dynamiques d'un objet, combinent les automates avec un modèle réactif synchrone. Lorsque l'on combine des automates avec des équations différentielles, on obtient un modèle de système hybride.

5.2 Automates et acteurs

Du point de vue de la modélisation par acteurs, chaque état de l'automate est représenté par un acteur, et les transitions entre états par des relations. Chaque acteur a un port d'entrée qui est une extrémité des transitions qui mènent à cet état, et un port de sortie qui est une extrémité des transitions qui partent de cet état. Chaque transition a une garde, qui indique quand la transition doit être effectuée, et une action, qui est exécutée lorsque la transition est suivie. Le comportement de l'acteur qui représente un état peut être non défini, auquel cas l'acteur ne fait que représenter un élément particulier de l'espace d'état du modèle ; ou peut être défini par un autre modèle, que l'on nomme le « raffinement » de l'état. Le raffinement d'un état définit le comportement de l'automate lorsqu'il se trouve dans cet état. Si ce raffinement est lui-même un automate, on a un automate hiérarchique.

Dans un automate hiérarchique H, le comportement associé à un état e peut être un autre automate S. Lorsqu'une transition mène dans l'état e, il y a deux possibilités :

1. l'automate S redémarre dans son état initial
2. l'automate S redémarre dans l'état où il était la dernière fois que l'on était dans l'état e

Dans le premier cas, on dit que la transition qui mène à l'état e se fait en mode réinitialisation (*reset mode*), alors que dans le second cas, elle de fait en mode historique (*history mode*).

De même, lorsqu'une transition quitte l'état e, l'automate S peut lui-même réagir à l'événement qui provoque la transition ou non. On dit d'une transition qui quitte un état dont le raffinement est un automate qu'elle *préempte* l'automate. Si on laisse l'automate du raffinement réagir à l'événement qui déclenche la transition, la préemption est faible, sinon, il s'agit de préemption forte.

5.3 Exemple d'automate

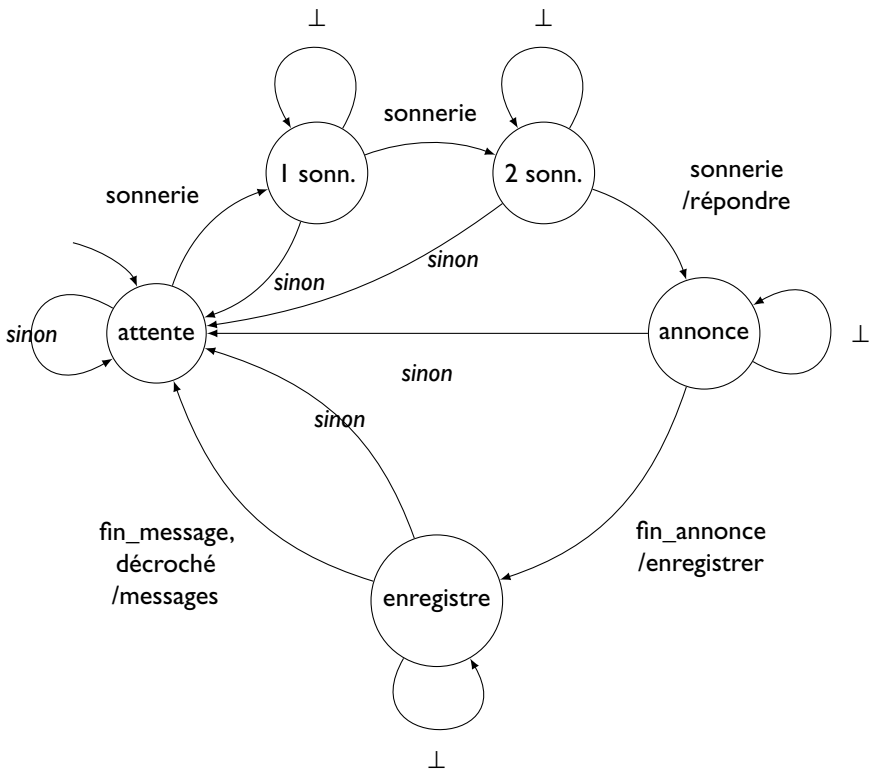


FIGURE 5.1 – Automate d'un répondeur téléphonique

La figure 5.1 montre un automate qui modélise le fonctionnement d'un répondeur téléphonique. Les entrées du système sont :

sonnerie : signal d'appel

décroché : le combiné est décroché

fin_annonce : la lecture de l'annonce est terminée

fin_message : la fin du message est détectée (présence de la tonalité par exemple)

Les sorties sont :

répondre : répondre à l'appel et commencer à lire l'annonce

enregistrer : commencer à enregistrer le message

messages : un ou plusieurs messages ont été enregistrés

Les états du système sont :

attente : état de repos du système

1 sonn. : on a compté une sonnerie

2 sonn. : on a compté 2 sonneries

annonce : on a compté 3 sonnerie et on a lancé la lecture de l'annonce

enregistre : l'annonce est terminée, on a lancé l'enregistrement du message.

Les transitions marquées par le symbole ⊥ sont prises lorsqu'aucune entrée n'est présente. Les transitions marquées par *sinon* sont prises lorsqu'une entrée autre que celles pour lesquelles existe une transition explicite est présente.

Les transitions ⊥ qui bouclent sur un état sans produire de sortie ne sont en général pas indiquées sur le diagramme d'un automate. Ces transitions ne sont utiles que lorsqu'on compose un automate avec un autre, l'un des automates étant parfois amené à ne rien faire (*to stutter* : bégayer, balbutier) pendant que l'autre réagit à une entrée.

5.4 Simulation et bisimulation

Deux automates ayant les mêmes alphabets d'entrée et de sortie peuvent être équivalents au sens où il produisent la même séquence de sortie lorsqu'on leur présente la même séquence d'entrée.

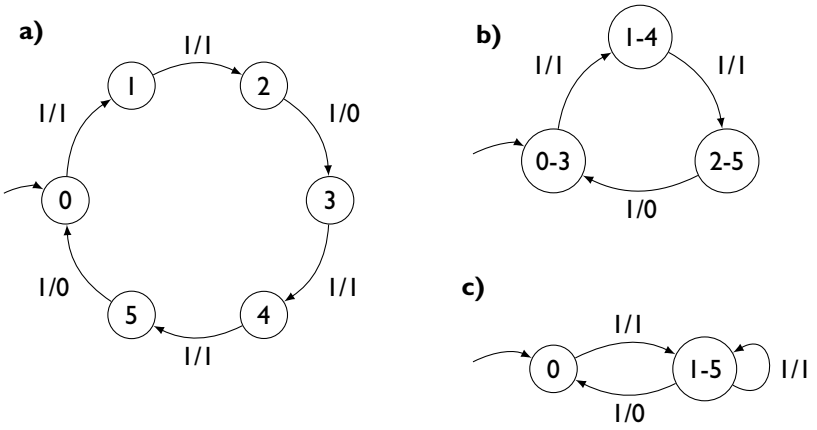


FIGURE 5.2 – Automates et simulation

La figure 5.2 montre trois automates. L'automate a a plus d'états que les autres, mais son comportement est identique à celui de b : les deux produisent une suite répétitive de deux 1 suivis d'un 0 quand ils reçoivent une suite de 1. L'automate c est non déterministe car deux transitions partent de l'état 1-5 avec pour garde 1. Il a plus de comportements possibles que a et b : il peut produire n'importe quelle suite commençant par 1 et telle qu'un 0 est toujours suivi d'un 1. L'automate c est donc plus général, ou plus abstrait que a et b.

On dit que l'automate c simule les automates a et b, et que l'automate b bisimule a, ou est bisimilaire à a. La bisimulation est une forme d'équivalence entre automates, alors que la simulation est une forme d'abstraction.

Un automate a simule un automate b s'il peut imiter le comportement de b pour toute séquence de symboles d'entrée. On place les deux automates dans leur état initial, et on fait réagir b à un symbole d'entrée. Si b est non déterministe, n'importe quelle transition possible peut être suivie. L'automate a doit alors réagir au même symbole d'entrée, de façon à produire le même symbole de sortie que b. Si a est non déterministe, n'importe quelle transition qui convient peut être choisie.

On dit que a bisimule, ou est bisimilaire à b si a simule b et b simule a.

Ainsi, pour montrer que l'automate c de la figure 5.2 simule l'automate b, on place les deux dans leur état initial. On fait réagir b à un symbole d'entrée, qui peut être ici $\perp$ ou 1. S'il s'agit de $\perp$, b ne fait rien, et c peut faire de même, ce qui sera toujours le cas dans cet exemple, nous ne considérerons donc plus $\perp$ comme entrée possible. S'il s'agit de 1, b produit 1 et passe dans l'état 1-4, et c ne peut que produire un 1 et passer dans l'état 1-5. Lorsque b reçoit de nouveau un 1, il produit un 1 et passe dans l'état 2-5, ce que c peut reproduire en restant dans l'état 1-5. Lorsque b reçoit à nouveau un 1, il produit un 0 et passe dans l'état 0-3, ce que c peut à nouveau reproduire en passant dans l'état 0. Nous sommes revenus au point de départ, c est donc capable d'imiter le comportement de b. L'ensemble de paires d'états $(0-3,0)(1-4,1-5)(2-5,1-5) \subset E_b \times E_c$ est la relation de simulation de b par c. Cette relation établit une correspondance entre les états des deux machines.

On peut de même montrer que b simule a, la relation de simulation (suggérée par les noms donnés aux états de b) étant :

$$(0,0-3)(1,1-4)(2,2-5)(3,0-3)(4,1-4)(5,2-5)$$

Si l'on note I_a l'état initial de l'automate a, et $((x,e),(x',o))$ la transition de garde e qui joint l'état x à l'état x' en produisant le symbole o, une relation de simulation $S_{a,b}$ de a par b doit vérifier les propriétés suivantes :

1. $(I_a, I_b) \in S_{a,b}$
2. si $(x, y) \in S_{a,b}$, alors pour tout symbole d'entrée e et pour toute transition $((x, e), (x', o))$ il existe une transition $((y, e), (y', o))$ telle que $(x', y') \in S_{a,b}$.

La première propriété indique que b doit être capable d'imiter a à partir des états initiaux des deux automates. La deuxième propriété correspond aux règles d'imitation de a par b.

La relation de simulation est trivialement réflexive. Elle est aussi transitive, mais elle n'est pas symétrique. Par exemple, l'automate c de la figure 5.2 simule l'automate b, mais la réciproque est fausse : partant de l'état initial, on fait réagir c à un 1, il produit un 1 et passe dans l'état 1-5. b ne peut imiter ce comportement qu'en passant dans l'état 1-4. Lorsqu'on fait ensuite réagir c à un 1, il peut produire un 0 et passer dans l'état 0, ce que b ne peut imiter. b ne simule donc pas c.

5.4.1 Relations entre comportements

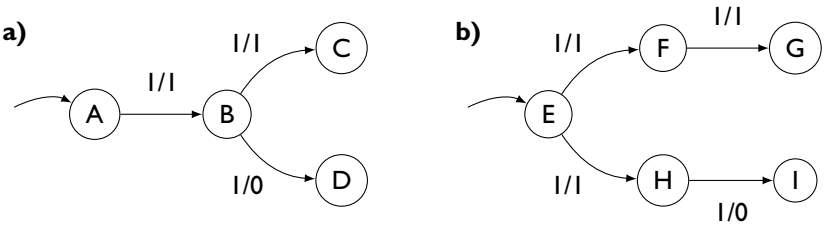


FIGURE 5.3 – Deux automates de mêmes comportements mais non bisimilaires

Une relation de simulation établit une correspondance entre deux automates, dont l'un est généralement beaucoup plus simple que l'autre. Cette relation permet de déterminer des propriétés de l'automate complexe en analysant l'automate le plus simple. En effet, si l'on appelle *comportement* d'un automate a une paire de suites $((e_i), (o_i))_{0 \leq i \leq n}$ telle que a puisse produire la séquence (o_i) lorsqu'il reçoit la séquence (e_i), alors si b simule a, les comportements de a sont inclus dans ceux de b.

$$b \text{ simule } a \quad \Rightarrow \quad C_a \subset C_b$$

Une conséquence de ceci est que si un comportement est impossible pour b, il l'est aussi pour a :

$$b \text{ simule } a \qquad \Rightarrow \qquad \big((e, o) \notin C_b \quad \Rightarrow \quad (e, o) \notin C_a \big)$$

Ainsi, si l'on prouve une propriété pour tous les comportements d'une abstraction de l'automate d'un système, cette propriété est vraie pour le système lui-même. De façon similaire, si l'on prouve qu'un comportement est impossible pour une abstraction de l'automate d'un système, on sait que ce comportement est impossible pour le système.

La réciproque de la relation entre simulation et inclusion des comportements est fausse : si $C_a \subset C_b$, on n'a pas nécessairement b simule a. Le contre-exemple de la figure 5.3 le montre : les automates a et b ont tous deux pour comportements $((1,1);(1,1))$ et $((1,1);(1,0))$, mais b ne simule pas a. En effet, partant des états initiaux, on fait réagir a qui produit un 1 et passe dans l'état B. b peut imiter ce comportement de deux façons, en allant dans l'état F ou dans l'état H. Si elle va dans l'état F et que a produit ensuite un 0 en allant en D, b ne peut pas l'imiter. De même, si b va en H et que a produit un 1 en allant en C, b ne peut pas l'imiter. b ne simule donc pas a, bien que leurs comportements soient les mêmes.

5.5 Composition d'automates et hypothèse synchrone

Lorsque l'on compose des automates, certaines sorties d'un automate peuvent être des entrées d'un autre. Comment ces entrées sont elles prises en compte ? Une façon de procéder

est de considérer que les sorties d'un automate sont simultanées avec ses entrées. C'est ce qu'on appelle l'*hypothèse synchrone*. Cette hypothèse fait que les sorties d'un système peuvent dépendre d'elles-mêmes si elles sont rebouclées sur les entrées (rétroaction).

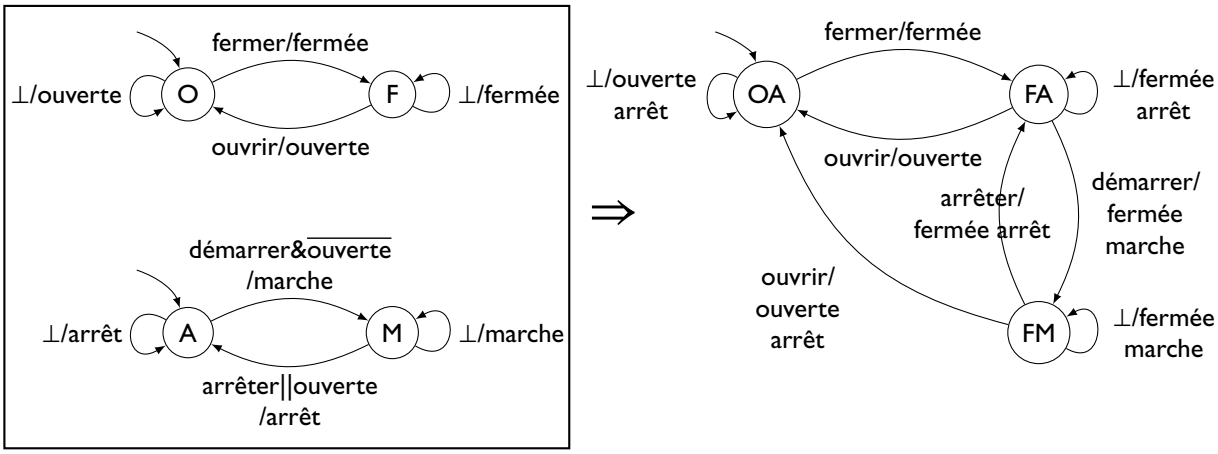


FIGURE 5.4 – Composition synchrone de deux automates

Considérons par exemple le système de la figure 5.4, où une porte peut être ouverte ou fermée et un moteur arrêté ou en marche. Ce système peut-être modélisé par deux automates, l'un représentant le comportement de la porte, l'autre celui du moteur. Lorsqu'on construit l'automate qui correspond au comportement de ces deux automates réunis selon l'hypothèse synchrone, on se rend compte qu'aucun état ne correspond à la situation où la porte est ouverte et le moteur en marche. En fait, cet état existe, mais il n'est pas atteignable depuis l'état initial car l'état M n'est atteignable que par une transition dont la garde est démarrer&$\overline{\text{ouverte}}$, et la transition qui permet de rester dans l'état O produit le symbole ouverte. La garde de la transition qui mène de OA à OM est donc en contradiction avec l'action de la transition. Cette garde ne peut jamais être satisfaite et l'on ne peut donc pas aller de OA à OM. Cette contradiction est la conséquence de l'hypothèse synchrone : l'action de la transition produit des symboles qui sont synchrones avec ceux qui déclenchent la transition, le symbole ouverte produit par la réaction de *stuttering* de l'automate de la porte doit être pris en compte pour suivre ou non la transition de A vers M. On montre de même que l'état OM n'est atteignable ni à partir de FA, ni à partir de FM.

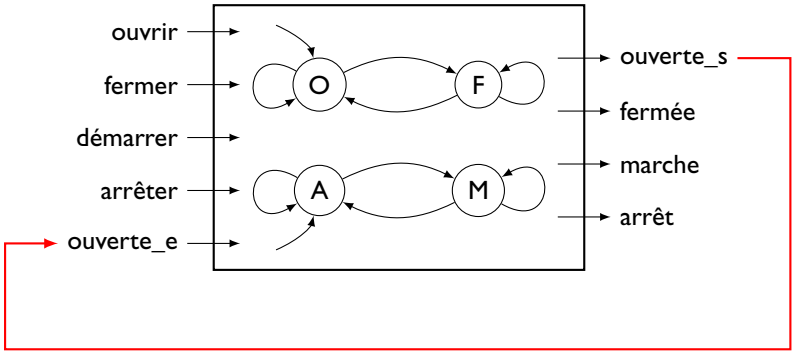


FIGURE 5.5 – Système en boucle ouverte et rétroaction

Cet exemple est un cas particulier de *rétroaction*, la sortie ouverte du système composé des deux automates étant utilisée comme entrée du même système. La forme la plus simple de rétroaction est l'équation $f(x) = x$. Une solution de cette équation, si elle existe, est appelé *point fixe* de f. Le comportement du système de la figure 5.4 est un point fixe du comportement du même système en boucle ouverte. On obtient le système en boucle ouverte en renommant les entrées et/ou les sorties de façon à ce qu'aucune entrée n'ait le même nom qu'une sortie. La figure 5.5 montre le système en boucle ouverte, l'entrée ouverte étant renommée ouverte_e

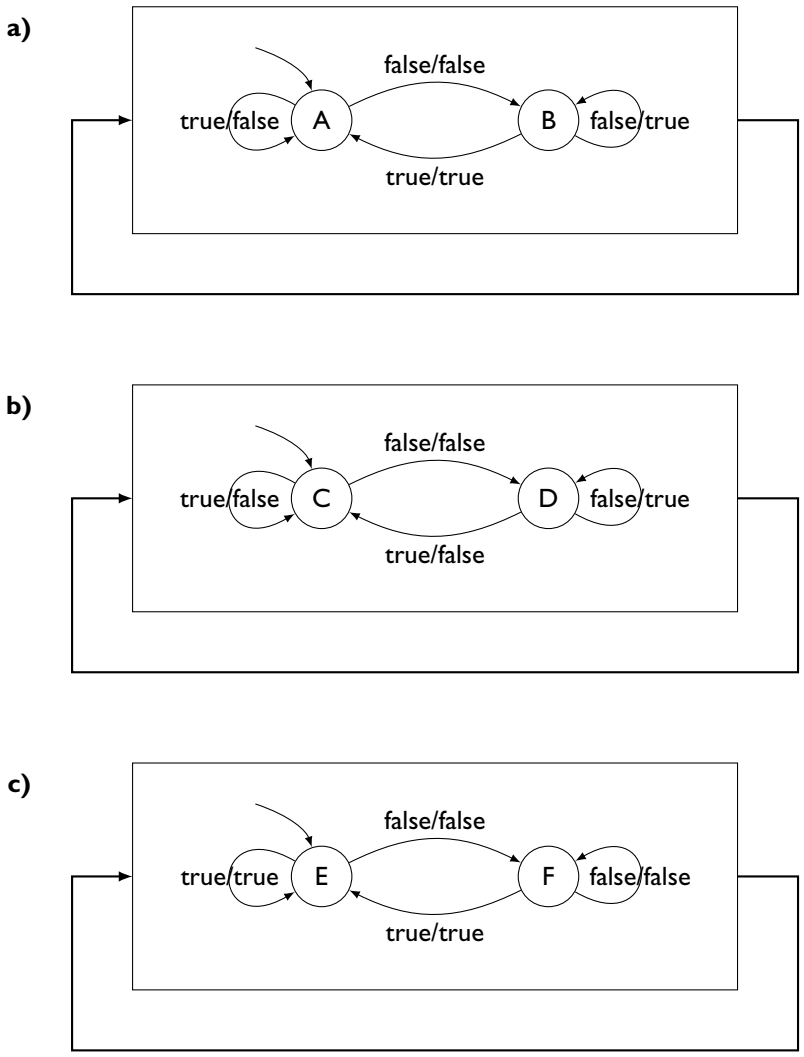


FIGURE 5.6 – Systèmes rétroactifs

et la sortie de même nom étant renommée ouverte_s. La flèche rouge indique la rétroaction, qui identifie la sortie ouverte_s à l'entrée ouverte_e.

Le problème qui se pose pour un système avec rétroaction est de savoir si son comportement est bien défini, c'est-à-dire si un point fixe existe et est unique. En effet, si aucun point fixe n'existe, le système n'a pas de comportement, et si le point fixe n'est pas unique, il peut en adopter plusieurs et n'a donc pas un comportement bien défini.

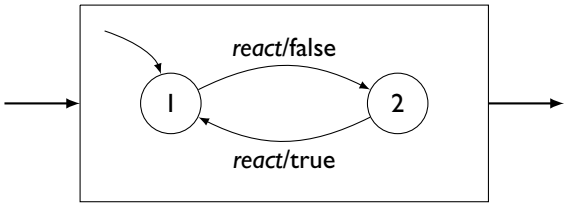


FIGURE 5.7 – Automate équivalent au système a de la figure 5.6

Parmi les systèmes rétroactifs de la figure 5.6, seul le a est bien défini, et l'automate équivalent est donné sur la figure 5.7. Cet automate ayant un alphabet d'entrée vide puisque la boucle du système a a été supprimée, on doit lui ajouter un symbole d'entrée virtuel nommé *react* afin de pouvoir le faire réagir.

Le système b n'a pas de point fixe dans l'état D car toutes les transitions qui quittent cet état ont un symbole de sortie différent de leur garde. Au contraire, le système c a deux points

fixes dans chacun de ses états, car deux transitions dont l'action est égale à la garde partent de chacun d'eux.

5.5.1 Automates à sorties déterminées par l'état

L'automate a est particulier car toutes les transitions qui quittent un de ses états ont la même action. La sortie d'un tel automate est donc déterminée par son état. Les automates dont la sortie est déterminée par l'état sont intéressants car ils donnent toujours un système bien défini lorsqu'ils sont dans une boucle de rétroaction, quels que soient les autres automates dans la boucle. En effet, puisque leur sortie ne dépend que de leur état, il n'est pas nécessaire de connaître leurs entrées pour déterminer leur sortie. On peut donc partir de leur sortie pour déterminer tous les symboles de la boucle en étant sûr que, quelque soit le symbole trouvé pour l'entrée de cet automate, la sortie que l'on a utilisé est la bonne.

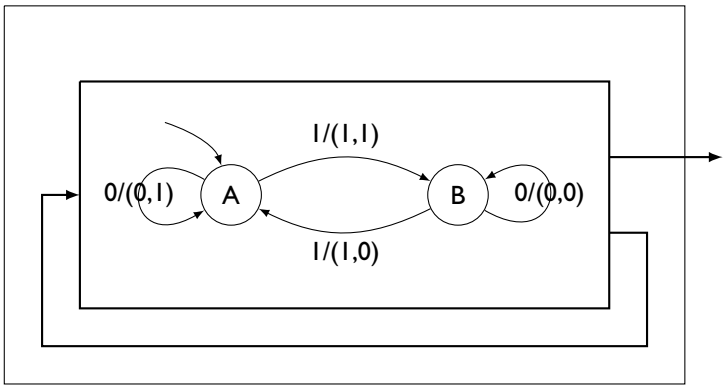


FIGURE 5.8 – Rétroaction avec sorties non déterminées par l'état

5.5.2 Sémantique constructive de la rétroaction

Dans la réalité, les systèmes considérés sont beaucoup plus complexes que ceux de nos exemples, et contiennent de nombreux automates et boucles de rétroaction. De plus, ces boucles ne contiennent pas nécessairement un automate dont les sorties sont déterminées par l'état. Il existe une méthode qui permet en général (mais pas toujours) de trouver le point fixe d'un système. Cette méthode est constructive, c'est-à-dire qu'appliquée mécaniquement, elle calcule le point fixe, ou indique qu'il n'existe pas ou n'est pas unique, en un nombre fini d'étapes.

Cette méthode consiste à considérer tous les signaux non spécifiés comme inconnus (on leur donne la valeur $\perp$). Partant des signaux d'entrée connus, on essaye de faire réagir les automates, dans l'ordre que l'on veut, afin de déterminer d'autres signaux. On s'arrête lorsque tous les signaux sont déterminés (on a alors trouvé l'unique point fixe), ou lorsqu'on n'a déterminé aucun nouveau signal en passant en revue tous les automates du système (il n'y a alors pas de point fixe, ou il n'est pas unique).

Si l'on considère l'exemple de la figure 5.8, on part en ne connaissant ni la valeur de la sortie rebouclée ni la valeur de la sortie du système. Comme il n'y a qu'un seul automate dans le système, on l'examine et l'on constate que dans l'état A, on ne peut pas déterminer sa sortie. L'automate n'est donc pas à sortie déterminée par l'état. Par contre, on remarque que quelle que soit la transition suivie depuis A, la valeur de la sortie rebouclée (deuxième élément de l'action) est 1. Le signal rebouclé passe donc de $\perp$ à 1. On réitère l'opération : l'entrée de l'automate est maintenant connue et vaut 1, l'automate effectue donc la transition vers B et produit la sortie $(1,1)$. La sortie du système est donc elle aussi déterminée et le point fixe est trouvé.

On procède de même dans l'état B : quelle que soit la transition suivie, la sortie bouclée vaut 0, l'automate suit donc la transition qui boucle sur l'état B et produit la sortie $(0,0)$. Le

comportement de ce système est donc de produire un 1 lors de sa réaction initiale, puis des 0 à chaque réaction suivante.

5.5.3 Recherche exhaustive

La méthode constructive de détermination du comportement d'un système bouclé ne fonctionne malheureusement pas dans tous les cas. Il existe des systèmes bouclés bien définis pour lesquels cette méthode échoue.

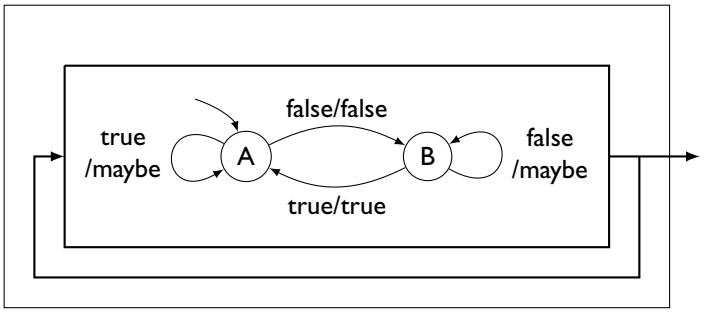


FIGURE 5.9 – Système sans sémantique constructive

Si on donne à la composition synchrone d'automate une sémantique constructive, ces systèmes n'ont pas de sens. On peut toutefois leur donner un sens hors de la sémantique constructive, en examinant toutes les transitions possibles dans chaque état et pour toutes les entrées. Si dans chaque cas, une seule transition permet d'éviter les contradictions, le système est bien défini.

Le système de la figure 5.9 n'a pas de sens selon la sémantique constructive, car partant de l'état initial, il est impossible de donner une valeur à la sortie de l'automate. Par contre, si on examine les transitions possibles à partir de l'état initial, on constate que seule la transition menant à l'état B est cohérente puisqu'elle produit false et que sa garde est false. De même, dans l'état B, seule la transition qui mène à A est cohérente et produit true. Il n'y a donc qu'un seule transition possible dans chaque état (l'alphabet d'entrée est ici vide), le système est donc bien formé.

La recherche exhaustive ne peut fonctionner que si le nombre d'états est fini et si le nombre de transitions partant de chaque état est fini. Dans la pratique, ces nombres doivent rester petits pour que l'algorithme termine en un temps raisonnable.

Les langages synchrones comme Esterel utilisent la sémantique constructive pour la composition synchrone. Certains systèmes bien définis sont donc rejetés par le compilateur. La contrepartie est que le compilateur trouve le point fixe ou rejette le programme en un temps raisonnable.

Chapitre 6

Esterel

6.1 Introduction

Esterel est un langage synchrone impératif. Il s'appuie sur l'hypothèse synchrone et la transmission instantanée des signaux pour donner un sens à la composition parallèle de processus. La nature impérative d'Esterel — le fait qu'il permet d'exprimer ce qui doit être exécuté étape par étape — le distingue des deux autres langages synchrones, Lustre et Signal, qui gèrent des flots de données.

Esterel est né au début des années 1980 au Centre de Mathématiques Appliquées de l'École des mines de Paris, au départ pour résoudre le problème de la programmation d'une petite voiture autonome qui devait participer à une course. Esterel a ensuite évolué, sous l'impulsion de Gérard Berry, pour devenir un langage mûr, avec une sémantique précise. À la même époque, Paul Caspi et Nicolas Halbwachs mettaient au point Lustre à Grenoble, et Paul Le Guernic et Albert Benveniste créaient Signal à Rennes.

Argos, une variante synchrone des StateCharts par Florence Maraninchi, a été créée en 1984 à Grenoble, et Charles André l'a étendu pour obtenir les SyncCharts, qui ont toute la puissance d'Esterel, mais avec une syntaxe graphique.

Esterel, comme les autres langages synchrones, est destiné à la conception de systèmes réactifs enfouis. Il permet d'associer parallélisme, déterminisme et gestion stricte du temps.

Le parallélisme permet d'exprimer plus naturellement le comportement d'un système. Toutefois, de nombreuses approches du parallélisme mènent au non déterminisme, c'est-à-dire au fait que, soumis aux mêmes entrées, le système n'adopte pas nécessairement le même comportement et ne produit pas les mêmes sorties. En associant parallélisme et déterminisme, les langages synchrones facilitent la compréhension, la mise au point et le test des systèmes.

La gestion stricte du temps dans les langages synchrones permet d'exprimer la simultanéité de deux événements. Par exemple, si l'on veut produire une sortie `minute` toutes les 60 occurrences d'une entrée `seconde`, `minute` ne sera pas produit en même temps que `seconde` dans un langage de programmation classique, alors qu'en Esterel, les deux signaux seront synchrones :

```
% Calcule les minutes et les heures à partir des secondes
module horloge:
  input seconde;
  output minute, heure;

  [ every 60 seconde do
      emit minute
    end every
  ] || [
    every 60 minute do
      emit heure
    end every
  ]
end module
```

Dans cet exemple, deux boucles s'exécutent en parallèle et communiquent par émission instantanée de signaux. À l'instant même de la 3600e occurrence du signal `seconde`, les signaux `minute` et `heure` seront émis simultanément.

6.2 Le système ABRO

Considérons la spécification suivante :

> Émettre le signal `O` aussitôt que les signaux `A` et `B` ont été émis.
> Ré-initialiser ce comportement à chaque fois que le signal `R` est émis.

Si ce système était réalisé sous la forme d'un circuit séquentiel, les signaux seraient véhiculés par des fils sous forme de tensions. D'un point de vue logique, les fils transportent des valeurs booléennes appelées `0` et `1`, ou `false` et `true`. En Esterel, on utilise la notion de présence ou d'absence d'un signal, indépendamment de sa valeur. Lorsque seule l'absence ou la présence d'un signal est significative (il n'a pas de valeur), on dit que le signal est *pur*. On dénote la présence ou l'absence d'un signal par les symboles `present` et `absent`, réservant `0` et `1` aux entiers, et `false` et `true` aux booléens. Un signal peut ainsi avoir une valeur entière (ou booléenne) en plus de son statut absent/présent.

Un événement d'entrée pour le système est donné par l'état de chaque signal d'entrée. La valeur d'un signal ne peut changer que lorsqu'il est présent. Lorsque le système réagit à un événement d'entrée, il produit un événement de sortie, décrit par l'état de chacun des signaux de sortie.

Ainsi, la spécification informelle du système ABRO n'est pas précise car elle n'indique pas ce qu'il faut faire quand plusieurs signaux sont présents en même temps. Si `A` et `B` sont présents en même temps, il n'y a pas de problème : il faut émettre `O`. Mais si `R` est aussi présent ? On supposera tout d'abord qu'il faut lui donner priorité et ne pas émettre `O`. De même, comment réagir si `A` et `B` sont tous les deux présents au démarrage du système ? Dans un premier temps, on considérera que la première réaction du système sert à l'initialiser, et que son comportement ne démarre qu'ensuite, ce qui est assez classique.

6.3 Traces d'exécution

Une trace d'exécution est une suite de réactions, chaque réaction étant représentée par son événement d'entrée et son événement de sortie. Dans la syntaxe du simulateur `csimul` d'Esterel, l'événement d'entrée est préfixé par > et l'événement de sortie est préfixé par `Output:`. Seuls les signaux présents sont indiqués dans les événements. La trace suivante correspond donc à un comportement correct vis-à-vis de la spécification ABRO :

```
> ;
Output:
> A;
Output:
> B;
Output: O
> R;
Output:
> A B;
Output: O
> A B R;
Output:
```

La notion de trace d'exécution montre que le temps d'Esterel est un temps logique. Il correspond à une suite de réactions, aussi appelés instants ou ticks. L'événement de sortie et l'événement d'entrée ont lieu au même instant car la réaction du système ne prend pas de temps logique. C'est ce que l'on appelle le modèle synchrone parfait, ou modèle à retard nul.

6.4 L'automate d'ABRO

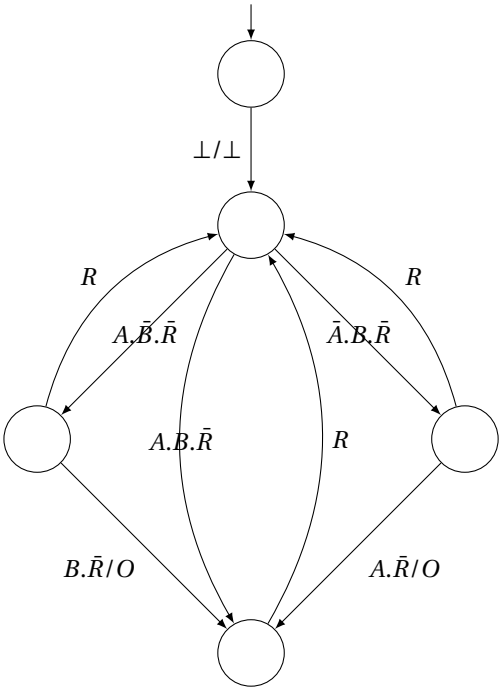


FIGURE 6.1 – L'automate du système ABRO

L'automate de la figure 6.1 correspond au comportement du système ABRO. Toutefois, chaque signal y apparaît plusieurs fois, en positif ou en négatif, et il n'est pas immédiat de comprendre ce qu'il fait. De plus, si on change la spécification pour définir un système ABCRO qui doit en plus attendre l'émission de `C` avant d'émettre `O`, on obtient un automate à 8 états en forme de cube. Avec n signaux à attendre, l'automate est un $n-$cube à 2^n sommets, ce qui devient très rapidement impraticable.

6.5 ABRO en Esterel

À l'opposé, le programme Esterel pour ABRO est simple :

```
module ABRO:
  input A, B, R;
  output O;

  loop
    [ await A || await B ];
    emit O
  each R
end module
```

Après la déclaration du module et des signaux d'entrée et de sortie, on trouve le corps du programme. Ce corps est écrit dans un style impératif en utilisant des flux de contrôle. Chaque instruction Esterel débute à un instant t, est active pendant un certain temps, et termine à un instant t'. Une instruction est instantanée si $t = t'$, ce qui correspond à un comportement combinatoire, abstraction faite des délais de propagation, dans l'optique d'une réalisation matérielle. Si $t' > t$, l'instruction prend du temps, ce qui correspond à un comportement séquentiel (le système change d'état).

L'instruction de retard `await A` prend toujours du temps et dure au moins un `A`. Si elle débute à l'instant t, elle termine au plus petit instant $t' > t$ tel que `A` est présent à t', et est donc

active entre les deux. En termes de flots d'exécution, le thread courant est bloqué sur `await A` jusqu'à ce que `A` survienne.

La mise en parallèle `await A || await B` exécute les deux `await` concurremment. Si la mise en parallèle débute à l'instant t, ses deux branches commencent exactement à l'instant t. Elle termine dès que la dernière des deux branches termine. Ici, `await A || await B` termine au moment exact ou `A` et `B` ont été émis. Les crochets `[` et `]` ne servent qu'à grouper les instructions. L'opérateur de séquencement `;` est en effet plus prioritaire que l'opérateur de mise en parallèle `||`.

L'opérateur de séquencement fait démarrer l'instruction suivante au moment même où la précédente termine. Ainsi, dans `p ; q`, `q` débute exactement à l'instant où `p` termine.

L'émission d'un signal est instantanée. Ici, `emit O` est exécuté exactement à l'instant où le dernier de `A` et `B` est émis, ce qui correspond assez bien à la notion de *aussitôt que* de la spécification informelle.

Enfin, la ré-initialisation par le signal `R` se fait grâce à l'instruction `loop ... each R` qui préempte son corps et le redémarre immédiatement à chaque occurrence de `R`. Le corps de la boucle démarre exactement au même moment que la boucle et s'exécute normalement jusqu'à l'occurrence suivante de `R`. Si le corps termine avant l'occurrence de `R`, la boucle attend, sinon, le corps de la boucle est terminé à l'instant même où `R` est présent, et il est immédiatement redémarré. La préemption du corps de la boucle est forte, ce qui signifie que le corps de la boucle ne reçoit pas le contrôle pour réagir à l'instant où il termine.

La préemption forte permet de gérer la priorité des signaux par l'imbrication des instructions : c'est l'instruction de préemption la plus externe qui est prioritaire. La gestion des priorités est donc exprimée par la structure du programme, et non par des numéros attribués aux différents threads.

Si l'on doit maintenant aussi attendre que `C` soit émis avant d'émettre `O`, la modification a apporter au code est mineure :

```
module ABCRO:
  input A, B, C, R;
  output O;

  loop
    [ await A || await B || await C ];
    emit O
  each R
end module
```

L'ampleur des changements dans le code est proportionnelle à celle des changements de la spécification, contrairement au cas de l'automate où elle était exponentielle.

6.5.1 Le principe WTO

Un principe qui permet de limiter le nombre d'erreurs et de faciliter la compréhension et la maintenance du code est *Write Things Once* (N'écrire les choses qu'une fois), ou WTO. Dans un programme Esterel, les répétitions observées dans l'automate sont remplacées par des éléments structurants comme les boucles. Les deux notations codent le même comportement, mais le programme Esterel a une sémantique plus abstraite. Le compilateur Esterel peut d'ailleurs traduire un programme Esterel en automate, même si ce n'est plus la technique utilisée par défaut actuellement.

Dans cet exemple, la mise en parallèle évite que la taille du code source augmente exponentiellement avec le nombre de signaux ; la mise en séquence permet de n'utiliser qu'une seule occurrence de `O` pour tous les cas de terminaison de l'attente des signaux ; la préemption permet de n'utiliser qu'une seule occurrence de `R` pour tuer le corps de la boucle quel que soit l'état dans lequel il se trouve. On remarquera que la mise en parallèle apparaît ici dans une séquence d'instructions qui est elle-même à l'intérieur d'une boucle. Ces notions sont en effet orthogonales en Esterel. Certains langages ne supportent la mise en parallèle qu'au niveau le plus élevé de la hiérarchie, perdant ainsi l'orthogonalité et nuisant au principe WTO.

6.6 Signaux valués

En plus de son attribut de présence, un signal peut avoir une valeur. La valeur d'un signal ne peut changer qu'aux instants où le signal est présent. On considère ici un système chargé de calculer une vitesse à partir de deux signaux purs `Centimetre` et `Seconde` :

> Compter le nombre de centimètres parcourus en une seconde, et émettre ce résultat grâce au signal `Vitesse` chaque seconde.

Dans un premier temps, nous supposerons que les signaux `Centimetre` et `Seconde` ne peuvent pas survenir au même instant. Le code du module `Vitesse` est alors :

```
module Vitesse:
  input Centimetre, Seconde;
  relation Centimetre # Seconde;
  output Vitesse : integer;

  loop
    var Distance := 0 : integer in
      abort
        every Centimetre do
          Distance := Distance + 1
        end every
      when Seconde do
        emit Vitesse(Distance)
      end abort
    end var
  end loop
end module
```

Dans les déclarations, le symbole # dans la section `relation` indique que les signaux `Centimetre` et `Seconde` sont exclusifs l'un de l'autre. L'environnement dans lequel s'exécute le module devra donc assurer la sérialisation de ces deux signaux.

Examinons la structure du corps de ce module en partant de l'extérieur. Tout d'abord, l'instruction `loop ... end loop` est une boucle infinie qui redémarre son corps aussitôt qu'il termine. Le mot-clef `var ... in` déclare une variable locale nommée `Distance`, de type `integer`, initialisée à 0.

L'instruction `abort` préempte son corps quand le signal `Seconde` est présent, de la même façon que `loop ... each`. La partie qui suit le `do` est optionnelle et indique les instructions à exécuter lorsque le corps est préempté. Ici, on émet la vitesse. Le corps du `abort` est une boucle `every` qui redémarre son corps à chaque occurrence du signal `Centimetre`. Ce corps affecte la somme de 1 et de la valeur de la variable `Distance` à la variable `Distance`. La somme et l'affectation ne prennent pas de temps, de sorte que le corps du `every` est instantané.

On remarque ici que, contrairement à un signal, une variable peut changer de valeur au cours d'un instant. Si nous choisissons d'utiliser un signal pour représenter la distance, nous ne pouvons pas écrire **emit** Distance(?Distance + 1) car ceci donnerait deux valeurs à `Distance` à certains instants, et le système n'admettrait alors pas de point fixe. Ce problème est résolu par l'utilisation de l'opérateur `pre` qui fournit la valeur d'un signal à l'instant précédent. Ainsi l'instruction **emit** Distance(**pre**(?Distance) + 1) est correcte.

6.7 Signaux et variables

Un signal est partagé par toutes les instructions qui sont dans sa portée lexicale, c'est-à-dire tout le module pour les signaux d'interface, et le corps de l'instruction `signal ... in` pour les signaux locaux. Un signal valué a un statut et une valeur unique à chaque instant. Le statut est volatile, mais la valeur est permanente : si la valeur ne change pas à un instant, elle reste identique à celle qu'avait le signal à l'instant précédent. La valeur d'un signal est changée par l'instruction `emit`, elle est lue grâce à l'opérateur `?`.

Une variable est modifiée par l'opérateur d'affectation `:=` qui est instantané. Sa valeur est obtenue en faisant référence à la variable par son nom. Contrairement à un signal, une variable peut prendre plusieurs valeurs successives au cours d'un instant. La portée d'une variable est le corps de l'instruction `var ... in`, qui peut contenir des instructions s'exécutant en parallèle. Il est donc nécessaire de préciser les règles qui assurent la consistance de la valeur d'une variable. Lorsqu'une variable est partagée par plusieurs instructions s'exécutant en parallèle :

- soit la variable est utilisée en lecture seule par toutes les instructions ;
- soit la variable est modifiée par une instruction et n'est utilisée ni en lecture ni en écriture par les autres.

Il est ainsi illégal d'écrire :

```
  x := 0;
  x := x + 1
||
  x := 1
```

puisqu'il n'y a pas moyen de donner un sens raisonnable à ce fragment de programme.

6.8 Préemption faible et immédiate

Si nous considérons maintenant que `Centimetre` et `Seconde` peuvent se produire en même temps, nous devons traiter ce cas particulier en associant le centimètre mesuré soit à la seconde qui se termine, soit à celle qui commence. Détaillons tout d'abord le sens exact de l'instruction `abort p when S` :

- Quand l'instruction débute, `p` débute immédiatement, et l'absence ou la présence de `S` est ignorée ;
- si `p` termine avant la prochaine occurrence de `S`, l'instruction `abort` toute entière termine au même moment ;
- si `S` survient alors que `p` n'est pas encore terminé, l'instruction `abort` termine immédiatement et `p` ne reçoit pas le contrôle à cet instant.

La première et la dernière clause font que l'instruction `abort` est retardée et forte. Retardée car la présence du signal `S` n'est pas prise en compte à l'instant où débute l'instruction, et forte car `p` ne reçoit pas le contrôle lors de la préemption. Il est possible de changer le comportement de `abort` à son commencement et à sa fin.

Pour rendre `abort` sensible à la présence de `S` dès son début, il suffit d'utiliser le mot-clef `immediate` dans la spécification du signal à attendre :

```
abort
  p
when immediate S
```

Dans ce cas, si `S` est présent au premier instant, `abort` termine instantanément, et `p` n'est pas exécuté du tout.

Pour que le corps du `abort` reçoive le contrôle lors de la préemption, il suffit le mot-clef `weak` (on a alors une préemption faible) :

```
weak abort
  p
when S
```

Si `p` est encore actif quand `S` survient, il termine mais s'exécute quand même à cet instant.

Enfin, le dernier cas possible permet de prendre en compte la présence de `S` à l'instant initial tout en donnant le contrôle à `p` lors de la préemption :

```
weak abort
  p
when immediate S
```

Si S survient à l'instant où débute le `abort`, ce dernier termine, mais p est quand même exécuté à cet instant.

L'instruction `loop p each S` n'est en fait pas une primitive et une abréviation pour :

```
loop
  abort
    p; halt
  when S
end loop
```

elle a aussi une forme immédiate dans laquelle le `when` est immédiat. De même, `every S do p end every` est une abréviation pour :

```
await S;
loop
  abort
    p; halt
  when S
end loop
```

qui a aussi une forme immédiate dans laquelle le `await` initial est immédiat.

Dans notre calcul de vitesse, si nous supprimons la relation d'exclusion entre `Centimetre` et `Seconde`, que ce passe-t-il ? Considérons l'instant auquel les deux signaux se produisent. Comme la préemption par `Seconde` est forte, l'instruction `every` qu'elle préempte n'est pas exécutée, ce qui fait que l'occurrence de `Centimetre` n'est pas comptée dans la seconde qui se termine. La boucle `loop` externe redémarre son corps, ce qui démarre immédiatement le `abort` et `every`, mais comme ce dernier est retardé, l'occurrence courante de `Centimetre` n'est pas prise en compte pour cette seconde qui débute. Le centimètre est donc perdu à chaque fois qu'il coïncide avec une seconde.

Si nous voulons compter le centimètre dans la seconde qui se termine, il suffit d'utiliser une préemption faible. Le `every` est alors exécuté lors de l'occurrence de `Seconde`, et si `Centimetre` est aussi présent, `Distance` est incrémentée, ce qui fait que la bonne vitesse est émise par **emit** Vitesse(Distance). Quand la boucle externe redémarre son corps, le `abort` et le `every` sont exécutés, mais comme le `every` est retardé, la présence de `Centimetre` est ignorée.

```
module Vitesse:
  input Centimetre, Seconde;
  output Vitesse : integer;

  loop
    var Distance := 0 : integer in
      weak abort
        % le centimètre est compté dans la seconde qui se termine
        every Centimetre do
          Distance := Distance + 1
        end every
      when Seconde do
        emit Vitesse(Distance)
      end abort
    end var
  end loop
end module
```

Si nous voulons compter le centimètre dans la seconde qui suit, on conserve la préemption forte, mais on utilise un `every` immédiat. Le centimètre n'est alors pas compté dans le `every` qui est préempté par `Seconde`, mais il l'est par le `every` qui est redémarré par la boucle `loop`.

```
module Vitesse:
  input Centimetre, Seconde;
  output Vitesse : integer;
```

```
    loop
      var Distance := 0 : integer in
        abort
          % le centimètre est compté dans la seconde qui débute
          every immediate Centimetre do
            Distance := Distance + 1
          end every
        when Seconde do
          emit Vitesse(Distance)
        end abort
      end var
    end loop
  end module
```

Enfin, si on utilise la préemption faible et non retardée, un centimètre qui coïncide avec une seconde est compté deux fois : une première fois dans le `every` qui reçoit le contrôle lors de la préemption faible, et une seconde fois par le `every` qui est redémarré par la boucle `loop` :

```
module Vitesse:
  input Centimetre, Seconde;
  output Vitesse : integer;

  loop
    var Distance := 0 : integer in
      weak abort
        % le centimètre est compté deux fois !
        every immediate Centimetre do
          Distance := Distance + 1
        end every
      when Seconde do
        emit Vitesse(Distance)
      end abort
    end var
  end loop
end module
```

6.9 Test du statut d'un signal

En plus d'utiliser un signal pour préempter une instruction, il est possible de tester la présence d'un signal pour choisir entre plusieurs alternatives grâce à l'instruction `present`. Cette instruction existe sous deux formes :

```
present sigspec then
  p
else
  q
end present
```

```
present
  case sigspec1 do
    p1
  case sigspec2 do
    p2
  ...
  else
    q
end present
```

La première forme exécute *p* si *sigspec* est vraie, *q* sinon. La seconde exécute le corps du premier case dont le *sigspec* est vrai, ou exécute *q* si aucun n'est vrai.

sigspec est une combinaison de signaux par des opérateurs booléens. L'opérateur and permet de tester la présence simultanée de deux signaux, l'opérateur or permet de tester la présence d'au moins un signal, et l'opérateur not permet de tester l'absence d'un signal. Lorsque *sigspec* n'est pas réduite au nom d'un signal, elle doit être placée entre crochets `[` et `]`.

Par exemple, pour réagir différemment à un clic de souris selon que la touche `Control` est enfoncée ou non, on pourrait écrire :

```
module Clic:
  input MouseClick, ControlKey;
  output SimpleClick, ContextalMenu;

  every MouseClick do
    present ControlKey then
      emit ContextualMenu
    else
      emit SimpleClick
    end present
  end every
end module
```

Pour obtenir des comportements différents selon les touches enfoncés, on pourrait avoir :

```
module Clic:
  input MouseClick, ShiftKey, ControlKey;
  output SimpleClick, ContextalMenu, Select, StartDrag;

  every MouseClick do
    present
      case [ShiftKey and not ControlKey] do
        emit Select
      case [ControlKey and not ShiftKey] do
        emit ContextualMenu
      case [ShiftKey and ControlKey] do
        emit StartDrag
      else
        emit SimpleClick
    end present
  end every
end module
```

6.10 Causalité constructive

Esterel accorde un sens aux programmes qui contiennent des cycles de rétroaction selon la sémantique constructive vue au chapitre sur les automates. Un programme Esterel est dit réactif s'il produit une sortie parfaitement définie pour chaque entrée. Il est dit déterministe s'il ne peut produire qu'une seule sortie pour chaque entrée. L'exemple suivant n'est pas réactif :

```
module NotReactive:
  output O;

  present O else
    emit O
  end present
end module
```

Aucun statut pour `O` ne peut en effet satisfaire ce programme puisque si `O` est présent, il n'est pas émis, et s'il est absent, il est émis (ce qui le rend présent).

Le programme suivant est réactif mais non déterministe :

```
module NotDeterministic:
  output O;

  present O then
    emit O
  end present
end module
```

En effet, si `O` est présent, il est émis, et s'il est absent, il n'est pas émis. Les deux comportements sont possibles et rien ne permet de choisir entre les deux.

En général, les problèmes de déterminisme et de réactivité mettent en jeu un cycle de dépendances entre signaux, ce qui les rend plus difficiles à mettre en évidence. L'exemple suivant est un cycle avec deux signaux :

```
module TwoSignals:
  output O1, O2;

  present O1 then
    emit O2
  end present
||
  present O2 else
    emit O1
  end present
end module
```

Si O1 est présent, O2 est émis, mais O1 n'est émis que si O2 est absent. Ce programme n'est donc pas réactif.

Une méthode simple pour éviter ces problèmes de causalité est d'interdire les cycles, où de les couper à l'aide de l'opérateur `pre`. Les programmes qui contiennent des cycles sont toutefois utiles, et Esterel les accepte si l'on peut construire leurs sorties pas à pas selon les règles suivantes :

1. un signal inconnu peut être dit présent s'il est émis ;
2. un signal inconnu peut être dit absent si aucune instruction ne peut l'émettre ;
3. la branche `then` d'un test peut être exécutée si le test est exécuté et que le signal est présent ;
4. la branche `else` d'un test peut être exécutée si le test est exécuté et que le signal est absent ;
5. la branche `then` d'un test ne peut pas être exécutée si le signal est absent ;
6. la branche `else` d'un test ne peut pas être exécutée si le signal est présent ;

Ces règles ne permettent pas de calculer les sorties du programme suivant qui, bien que logiquement correct (il n'admet qu'un comportement où ses deux sorties sont absentes), n'est pas constructif :

```
module TwoSignals:
  output O1, O2;

  present O1 then
    emit O1
  end present
||
  present [O1 and not O2] then
    emit O2
  end present
end module
```

Chapitre 7

Systèmes mixtes et modaux

7.1 Introduction

De nombreux outils mathématiques facilitent l'étude et la conception des systèmes linéaires (théorie des fonctions du plan complexe, diverses transformations). Malheureusement, de nombreux systèmes réels ne sont pas linéaires et sont donc beaucoup plus difficiles à modéliser. Toutefois, certains systèmes non-linéaires sont « linéaires par morceaux », c'est-à-dire qu'ils peuvent être modélisés par des systèmes linéaires dans différentes conditions de fonctionnement. On qualifie les systèmes modélisés par différents systèmes linéaires en temps continu de *systèmes hybrides*. Ces systèmes sont un cas particulier de systèmes modaux, c'est-à-dire de systèmes qui ont plusieurs modes de fonctionnement, chacun représenté par un modèle distinct. Le passage d'un mode à un autre se fait lorsque certaines conditions sur l'état et/ou les signaux traités par le système sont remplies.

7.2 Modèles mixtes

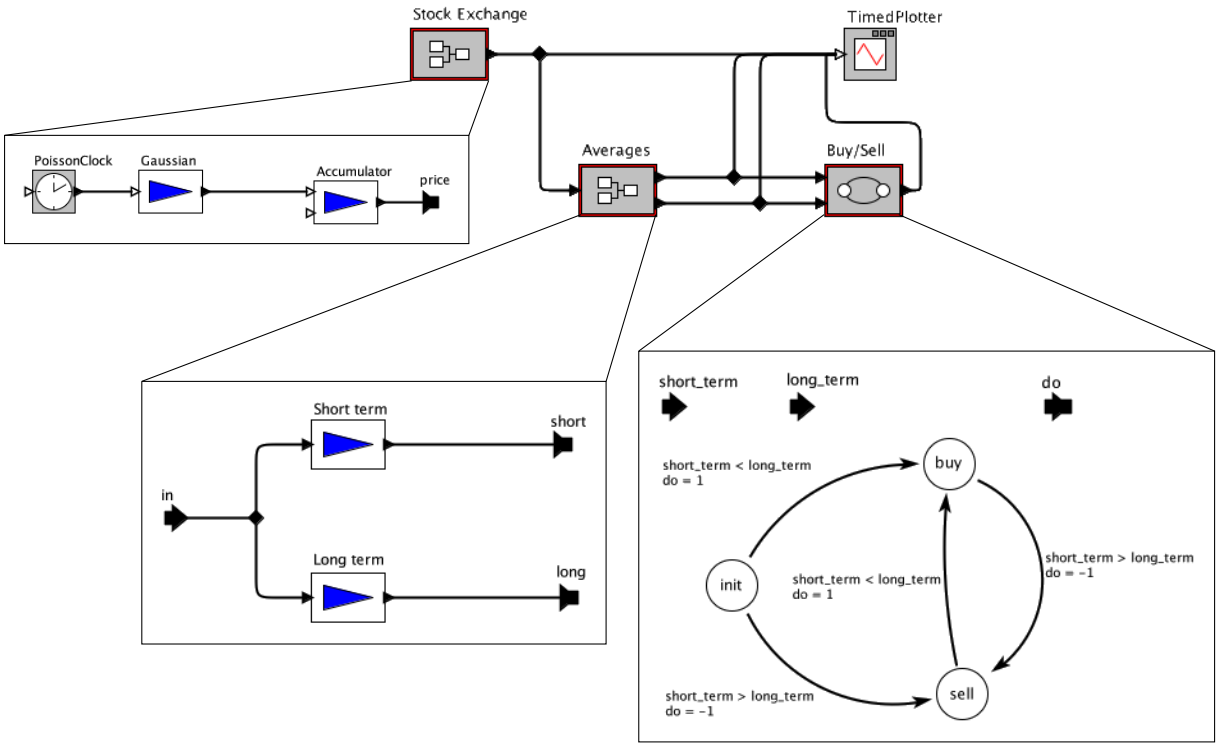


FIGURE 7.1 – Exemple de système mixte

Un modèle mixte est simplement un modèle qui juxtapose des modèles temporels et des automates. Par exemple, le modèle de la figure 7.1 analyse le cours d'une action en bourse en calculant sa moyenne sur deux fenêtres temporelles de largeurs différentes. La moyenne glissante qui a la fenêtre la plus étroite donne une valeur moyenne sur le court terme, tandis

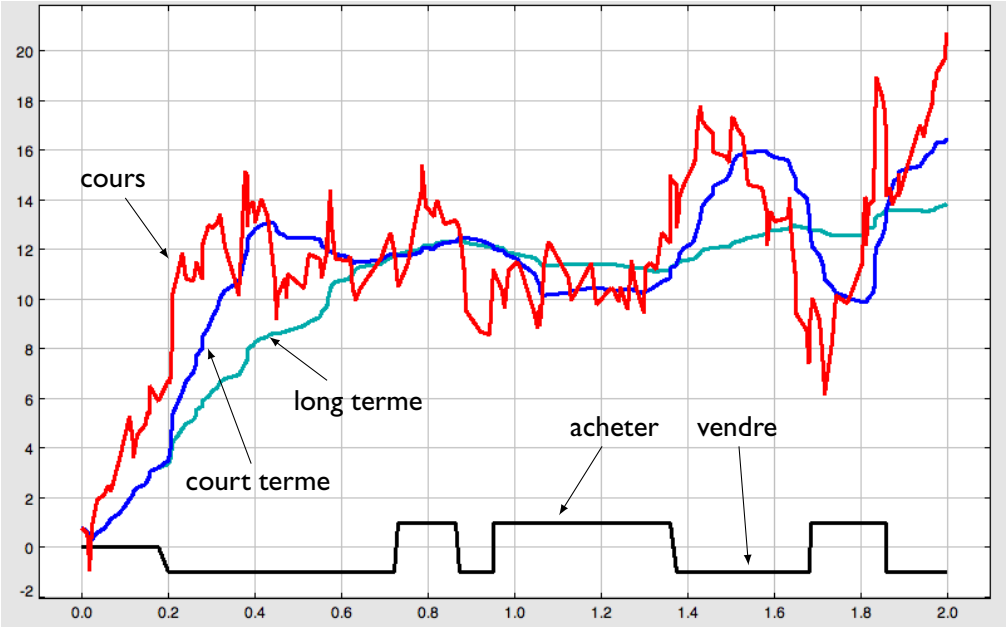


FIGURE 7.2 – Comportement d'un système mixte

que l'autre donne une valeur moyenne sur le long terme. Lorsque la valeur moyenne à long terme est supérieure à la valeur moyenne à court terme, il faut acheter (l'action coûte peu par rapport à ce qu'elle peut rapporter). Lorsque la valeur moyenne à long terme est inférieure à la valeur moyenne à court terme, il faut vendre (l'action rapporte plus maintenant). Un automate qui change d'état à chaque fois qu'une des moyennes passe au-dessus de l'autre permet de modéliser un comportement acheteur ou vendeur selon les valeurs respectives des moyennes. Ce comportement est illustré par la figure 7.2 qui montre le cours d'une action, les deux moyennes glissantes — l'une ayant une largeur de 10 unités de temps, l'autre de 25 — et les changements d'état de l'automate.

Ce modèle est bien linéaire par morceaux : tant que les deux moyennes restent dans la même configuration, ses sorties sont constantes.

7.3 Modèles modaux

Au lieu de juxtaposer les modèles de calcul comme dans les modèles mixtes, on peut les imbriquer hiérachiquement. Lorsqu'un état d'un automate « contient » un modèle qui décrit le fonctionnement du système quand l'automate est dans cet état, on parle de système *modal*. On peut en effet considérer chaque état de l'automate comme un mode de fonctionnement du système. Le modèle qui se trouve « sous » un état dans la hiérarchie, et qui décrit donc le comportement du système dans le mode correspondant, est appelé *raffinement* de l'état.

La figure 7.3 donne la structure générale d'un système modal. Les transitions entre états (ou modes) ont une garde et des sorties, mais aussi une action qui permet de choisir l'état initial du raffinement de l'état cible de la transition. Ceci permet notamment d'assurer la continuité des signaux temporels dans un système linéaire par morceaux.

Les gardes des transitions sont sensibles aux événements d'entrée, à des conditions sur la valeur des signaux d'entrée, mais aussi sur la valeur des paramètres du raffinement de l'état courant.

Exemple : Une balle qui rebondit sur le sol peut être considérée comme un système modal à 3 états (voir figure 7.4) : l'état initial est quitté immédiatement par une transition qui initialise les paramètres (position et vitesse) de l'état « free » qui correspond au mode « chute libre » de

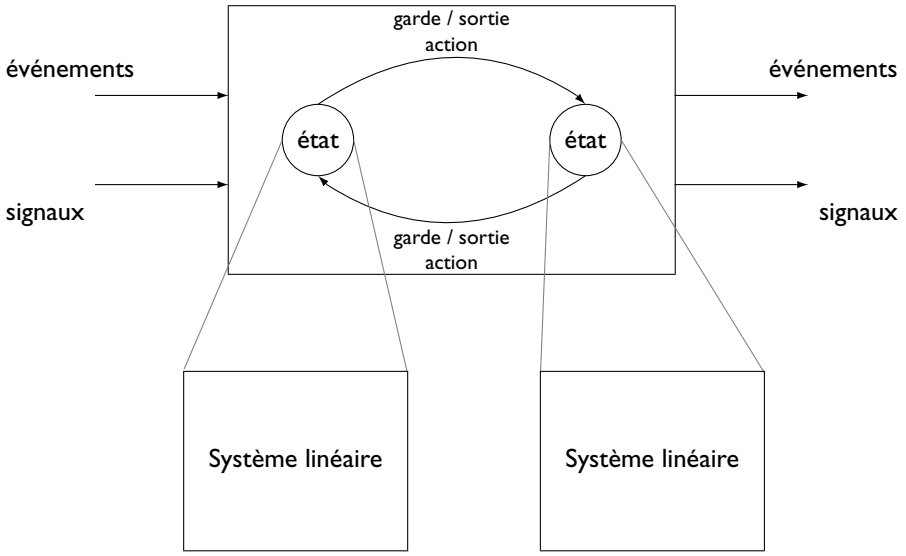


FIGURE 7.3 – Structure d'un système modal

la balle. Si on fait abstraction de la phase de rebond dans notre modèle, à chaque fois que la balle heurte le sol, elle effectue une transition du mode « free » vers lui-même qui inverse sa vitesse, aux dissipations dues à l'élasticité imparfaite du choc près. Lorsque l'amplitude des rebonds devient trop faible pour décoller la balle du sol, cette dernière s'arrête et passe dans l'état « stop ».

Le raffinement de l'état « free » modélise la balle en chute libre come indiqué sur la figure 7.5. La vitesse de la balle est obtenue en intégrant son accélération à partir de sa vitesse initiale, et sa position est obtenue en intégrant cette vitesse à partir de sa position initiale. Le rebond est détecté en comparant la position de la balle à celle du sol. Les paramètres de ce mode sont donc la position et la vitesse initiales de la balle, et sont fixés par les transitions qui mènent à l'état « free ».

L'évolution de la position de la balle selon ce modèle est donnée par la figure 7.6.

7.4 Automates temporisés

Les automates temporisés sont la forme la plus simple de système hybride à temps continu : le raffinement d'un mode est un système qui ne fait que mesurer le passage du temps.

La figure 7.7 montre un automate temporisé à deux états. L'état initial est « mode1 ». Dans cet état, la dérivée de s par rapport au temps est 1 et le signal tick vaut 0. Lorsque s atteint la valeur 1, la garde de la transition vers l'état « mode2 » est satisfaite. Cette transition est donc effectuée : le signal tick passe à -1 et l'automate passe dans l'état « mode2 ».

Dans l'état « mode2 », la dérivée par rapport au temps de s vaut aussi 1, mais la garde de la transition qui ramène vers l'état « mode1 » n'est satisfaite que lorsque s atteint la valeur 2. L'automate reste donc deux unités de temps dans l'état « mode2 » alors qu'il n'était resté qu'une unité de temps dans l'état « mode1 ».

Lorsqu'on revient dans l'état « mode1 », la transition doit se faire en mode « réinitialisation », sinon, le raffinement de cet état serait réactivé dans l'état où il était la dernière fois qu'il était actif, c'est-à-dire avec un intégrateur qui intègre à partir de 1. La transition vers « mode2 » serait alors immédiatement prise. Il en est de même pour la transition de « mode1 » vers « mode2 ».

Le fonctionnement de l'automate est illustré par la figure 7.8 où le signal carré tracé en bleu dans le haut du diagramme donne la valeur du signal mode qui vaut 1 dans l'état « mode1 » et 2 dans l'état « mode2 » ; le signal d'allure triangulaire tracé en vert sous le précédent donne la valeur du signal s ; et le signal tracé en rouge dans le bas du diagramme donne l'évolution du signal tick qui est mis à -1 lors de chaque transition.

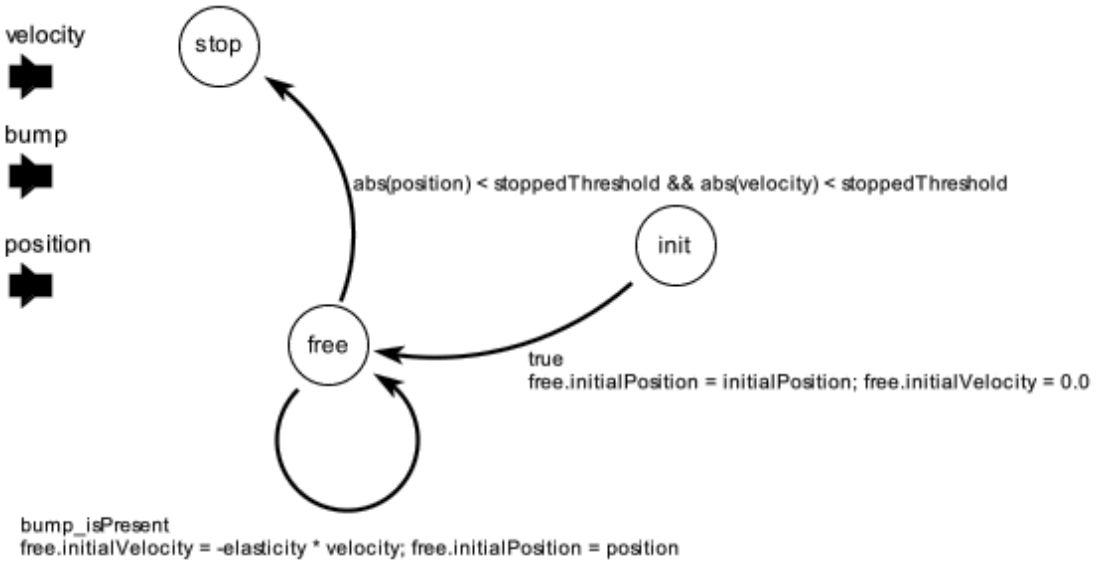


FIGURE 7.4 – Modèle modal d'une balle qui rebondit.

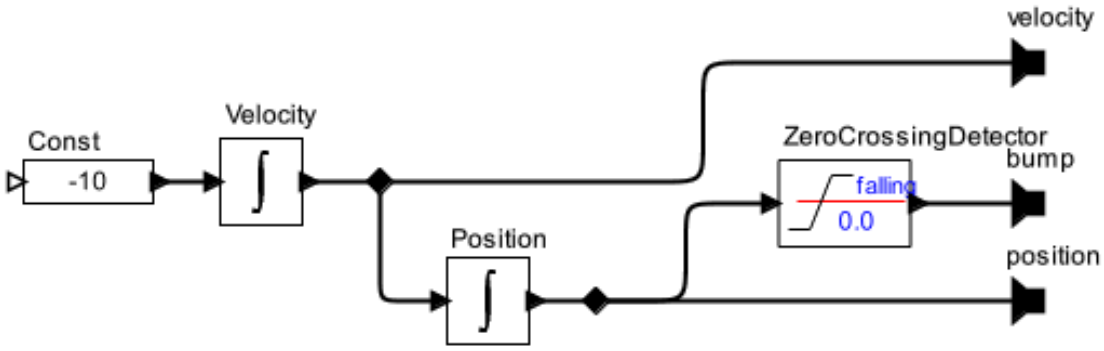


FIGURE 7.5 – Modèle d'une balle en chute libre.

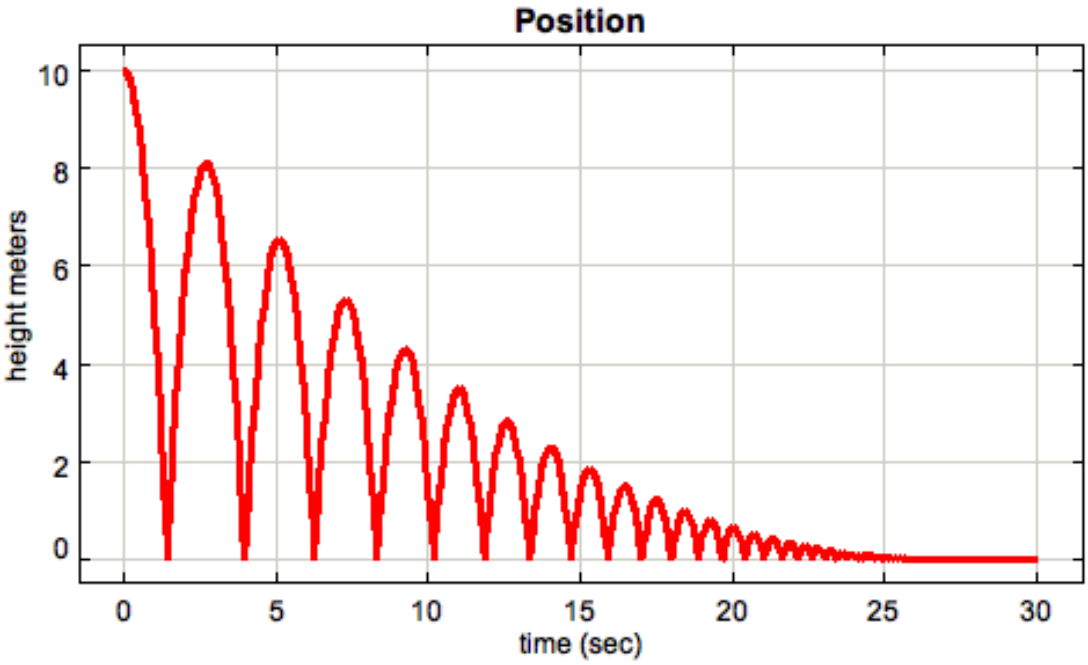


FIGURE 7.6 – Évolution de la position de la balle au cours du temps.

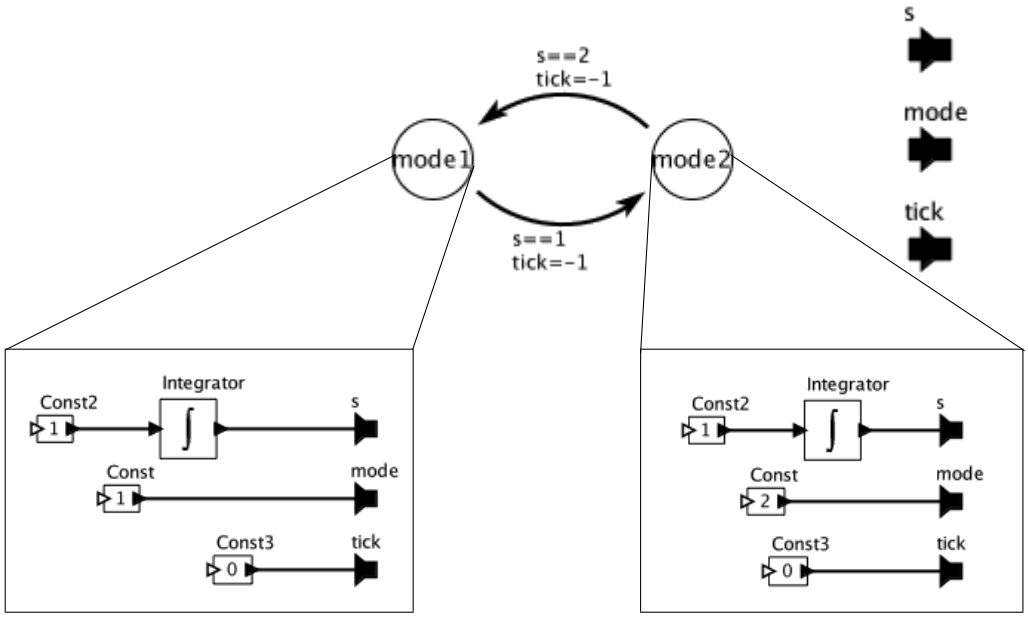


FIGURE 7.7 – Exemple d'automate temporisé

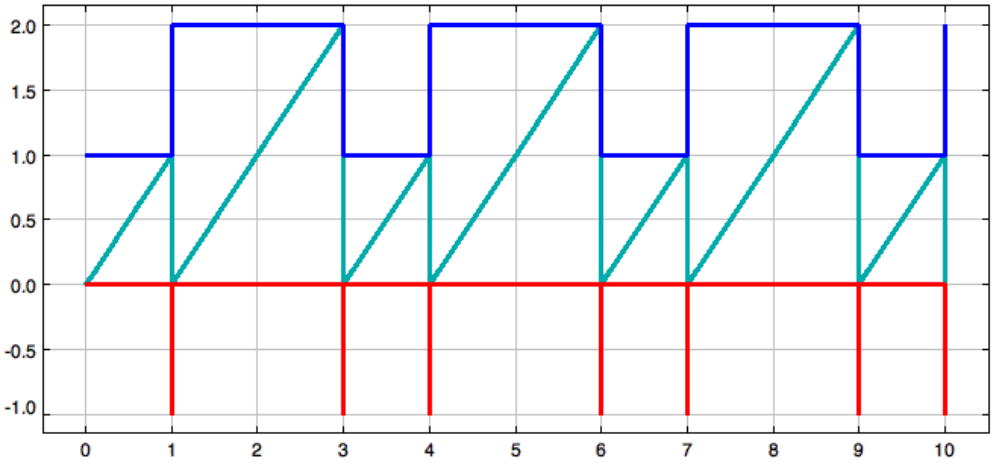


FIGURE 7.8 – Évolution du mode et des signaux de l'automate

Au lieu d'utiliser des transitions en mode réinitialisation, on aurait pu leur associer une action de remise à zéro de la valeur initiale de l'intégrateur du mode cible. Il aurait alors fallu faire apparaître cette valeur initiale en tant que paramètre du raffinement des états de l'automate.

Composé le 20 novembre 2009 par pdfTEX 1.40-10
avec Adobe Utopia, *Computer Modern* Sans Serif *et* `Typewriter`, *en 11 pt.*