7e Conférence Internationale de MOdélisation et SIMulation -MOSIM'08 - du 31 mars au 2 avril 2008 – Paris – France
« Modélisation, Optimisation et Simulation des Systèmes : Communication, Coopération et Coordination.».

UN FRAMEWORK FONCTIONNEL POUR LA MODÉLISATION ET LA SIMULATION INFORMATIQUE DE SYSTÈMES COMPLEXES

A. Laurent THIRY, B. Thomas COLLONVILLE, C. Bernard THIRION

[A,B,C] Ecole Nationale Supérieure d'Ingénieurs - Sud Alsace
12, rue des frères Lumière - 68093 Mulhouse cedex (France)
A. laurent.thiry@uha.fr, , B. thomas.collonville@uha.fr, C. bernard.thirion@uha.fr

RÉSUMÉ: *La modélisation informatique et la simulation de systèmes à comportements complexes requiert généralement l'intégration de plusieurs formalismes pour la représentation de la dynamique (continue/discrète) et d'aspects plus informatiques. Dans ce contexte, l'article propose un cadre de travail outillé (ou framework) profitant d'un langage fonctionnel moderne pour permettre une approche unifiée de ces systèmes. En particulier, le framework propose un ensemble de métamodèles pour les flots de données (pour les comportements continus), les automates états-transitions (pour les comportements discrets) et un métamodèle d'actions (pour les interactions avec une plate-forme matérielle ou des exécutions parallèles). Un modèle de robot hexapode est développé comme application du framework.*

MOTS CLEFS: *Ingénierie Dirigée par les Modèles, Langage fonctionnel Haskell*

1. INTRODUCTION

La modélisation informatique des systèmes complexes est en général une tâche difficile qui nécessite l'intégration de plusieurs types de modèles, ou langages de modélisation, (Thiry *et al*., 2006). Par exemple, les flots de données utilisés pour décrire des comportements continus, doivent pouvoir intégrer des éléments discrets pour décrire, par exemple, différents modes de fonctionnement, et des éléments plus informatiques comme la sémantique d'exécution.

Il existe différentes manières de décrire les modèles de systèmes: avec un langage impératif, orienté objets, fonctionnel, etc. L'intérêt d'un langage fonctionnel est qu'il dispose d'une fondation mathématique précise (formelle) et qu'il permet de décrire simplement les différents formalismes utilisés pour représenter les systèmes continus ou discrets. Malgré de nombreux intérêts, les langages et modèles fonctionnels restent trop peu utilisés dans le cadre de la modélisation et de la simulation. Pour mieux profiter des avantages d'un style fonctionnel, l'article propose un framework expliquant comment décrire puis intégrer des modèles de comportements et comment les combiner à des modèles d'implémentation (prenant en compte la plate forme d'exécution). L'intérêt du travail proposé est alors de montrer comment utiliser un langage fonctionnel avec un haut niveau de généricité, Haskell, pour modéliser/simuler autrement des systèmes dynamiques.

Les langages fonctionnels sont intéressants car ils permettent de capturer l'essence d'un langage de modélisation avec un ensemble de fonctions. Par exemple, les modèles continus discrétisés peuvent se décrire à l'aide du langage L = {constante, gain, retard, somme, ...} où chaque élément correspond à une fonction pour créer et combiner des signaux (ou des flots de données). Associé à l'opérateur de composition de fonctions (o), L permet d'exprimer la dynamique des systèmes discrétisés usuels. Les éléments de L sont eux-mêmes décrits par des fonctions plus génériques proposées par le langage fonctionnel considéré. Différentes réalisations sont alors possibles pour la simulation ou la génération de code cible. Dans le domaine de l'Ingénierie Dirigée par les Modèles (IDM), L est appelé Langage Spécifique de Domaine (DSL) où métamodèle, i.e. modèle d'un langage de modélisation, (Mellor *et al*., 2003). Ainsi, l'article explique comment spécialiser un langage fonctionnel moderne (Haskell) à l'aide de (méta)modèles continus, discrets et informatiques (concurrents). L'ensemble des éléments développés est décrit de manière compacte et intelligible, et permet l'étude d'une large gamme de modèles comme l'illustre le cas du robot hexapode de la partie 4. Contrairement aux outils dédiés, comme Matlab/ Simulink (Matworks), le framework proposé est ouvert et peut-être facilement adapté ou étendu ; par exemple, (Thiry *et al*., 2008), associe au modèle discret présenté dans la suite à un modèle de propriétés temporelles pour

permettre une vérification/validation automatique du bon fonctionnement d'un système à l'aide d'un model checker.

Cet article se compose de quatre parties. La première partie présente les principales classes de systèmes d'un point de vue mathématique, IDM puis fonctionnel. La seconde partie propose trois métamodèles fonctionnels regroupés dans un framework Haskell, et un ensemble d'outils, pour la modélisation et la simulation des systèmes continus/discrets, et concurrents/ communicants. La troisième partie présente un modèle de robot hexapode et une simulation du mouvement des différentes pattes lors d'un déplacement. La quatrième partie reprend les éléments importants de cet article et précise les perspectives envisagées.

2. (MÉTA)MODÉLISATION DES SYSTÈMES

2.1. Approche mathématique

D'un point de vue mathématique, un système est modélisé par une base de temps T, un ensemble d'états X, un ensemble d'entrées U, et une fonction de transition $F:X \times U \rightarrow X$, (Lee, 2003), (Maler, 1998). Une classification en fonction de T et X des différents types de modèles est donnée par le tableau 1, (Vangheluwe et de Lara, 2003). Le comportement est défini, à partir d'un état initial $x_0 \in X$, d'une entrée $u:T \rightarrow U$, par une fonction $x:T \rightarrow X$ qui vérifie $x(0)=x_0$ et $x(i+1) = F(x(i),u(i))$ pour T discret, et $dx(t)/dt = F(x(t),u(t))$ pour T continu. La sortie d'un système est donnée soit par une fonction $y:T \times X \times U \rightarrow Y$, avec Y l'ensemble des sorties, soit en complétant la fonction de transition $F:X \times U \rightarrow (X \times Y)$. U et Y peuvent être des valeurs numériques (cas continu) et/ou des événements (cas discret).

X \ T	Continu (|R)	Discret (|N)
Continu (|R^N, par ex.)	Equations Différentielles Algébriques (DAE)	Equations aux Différences
Discret	Physique naïve	Automates à Etats Finis (FSA) Réseaux de Petri (RdP)

Tableau 1. Classification des modèles de système.

Ces modèles permettent alors de décrire la plupart des systèmes dynamiques. Pour une mise en oeuvre informatique cependant, d'autres éléments doivent être considérés avec en particulier le modèle d'exécution (Jantsch et Sander, 2005). En particulier, un système doit pouvoir être décomposé en sous-systèmes décrits par un, ou plusieurs des modèles précédents, et évoluant en parallèle. Parmi les conséquences de cette décomposition, il y a la prise en compte d'un modèle d'interprétation synchrone ou asynchrone. Pour le continu, l'interprétation (et la sémantique) synchrone est plutôt utilisée ; dans le cas discret, c'est l'interprétation asynchrone qui est privilégiée. Dans les deux cas, un ordonnanceur fixe l'ordre d'exécution des différents composants. Dans le modèle synchrone, tous les composants évoluent dans la même base de temps T. Dans le cas asynchrone, chaque composant a sa propre base de temps T_i ; il est alors nécessaire d'introduire des canaux communication (ou buffer) pour permettre l'échange de données et la synchronisation des différents composants (Navet, 2006).

Cette description montre que la modélisation et la simulation des systèmes repose sur plusieurs types de modèles avec des parties décrivant un comportement continu/discret et d'autres parties décrivant la mise en oeuvre informatique ; ces différentes parties sont souvent complémentaires et doivent pouvoir être intégrées. La figure 1 présente un modèle de système combinant une partie continue (P) et une partie discrète (Q) qui doivent évoluer en parallèle (||) ; ce modèle servira d'illustration dans les différentes propositions de cet article. La partie 4 présente un exemple plus complet avec un modèle informatique pour un robot hexapode.

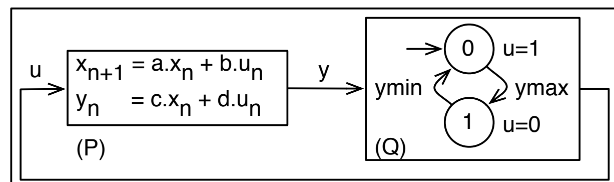

Figure 1. Modèle discret, continu et concurrent (P||Q).

2.2. Approche IDM

L'Ingénierie Dirigée par les Modèles (IDM) propose les concepts et les outils pour capturer et intégrer des langages de modélisation, (Mellor *et al.*, 2003). Chaque langage est décrit par un métamodèle, généralement un diagramme des classes UML (OMG, 2007) complété de contraintes logiques (Varro et Pataricza, 2003), et est souvent implémenté sous la forme d'un Langage Spécifique à un Domaine (DSL), (Deursen *et al.*, 2000). Des exemples de métamodèles sont donnés par (Breton et Bézivin, 2001), pour les réseaux de Petri et (Denckla et Mosterman, 2005), ou (Mathaikutty, 2005) pour les blocs diagrammes. Les métamodèles peuvent être implémentés dans un langage de programmation sous la forme de Langages Spécifiques à un Domaine (DSL). Plus précisément, dans un DSL, les métamodèles sont interprétés comme des types de données et des fonctions utilisés pour adapter un langage plus général à un domaine particulier. Par exemple, (Hudak *et al.*, 2003) propose un DSL pour la robotique et la vision.

Les métamodèles peuvent être utilisés pour configurer les méta-outils, proposés par la communauté IDM, c'est à dire des outils génériques pour créer et manipuler des types de modèles. Des exemples de méta-outils utilisés dans le cadre des systèmes sont l'Environnement de Modélisation Générique GME (Dubey, 2005), et Atom3 (Vangheluwe *et al*., 2003). Pour comprendre le concept de métamodèle, la figure 2 propose une spécification possible pour les blocs diagrammes. Les blocs diagrammes sont utilisés pour modéliser les systèmes dynamiques avec des signaux et des fonctions sur ces signaux. Une fonction f est représentée graphiquement par un bloc avec des entrées (correspondant alors aux arguments de f), des sorties (correspondant alors au résultat de f). La composition de fonctions est obtenue en reliant la sortie des blocs à l'entrée d'autres blocs. Les blocs peuvent être classés en deux catégories: les blocs de base, qui relient les entrées aux sorties par une équation algébrique ; et les composites dont la structure interne est un assemblage de blocs plus simples. Les composites permettent d'organiser les modèles de façon hiérarchique. Les concepts précédents peuvent être représentés par un diagramme des classes UML (OMG, 2007) comme le montre la figure 2: les concepts sont représentés par des classes et les relations entre ces concepts par des associations entre ces classes.

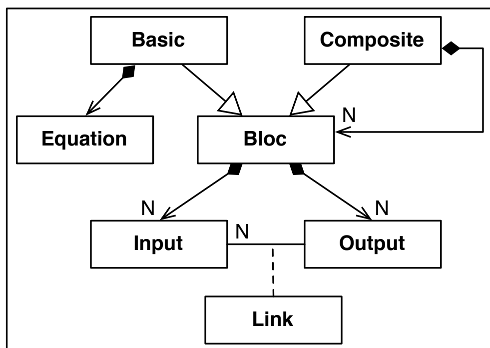

Figure 2. Métamodèle pour les blocs diagrammes.

La notion de métamodèle fonctionnel utilisée dans la suite de cet article consiste à interpréter les diagrammes précédents dans un langage fonctionnel (sous la forme de DSLs). Il est alors possible de réaliser des modèles conformes à ces métamodèles pour les simuler ou les transformer. La démarche suivie est expliquée dans les sections suivantes.

2.3. Approche fonctionnelle

L'approche fonctionnelle s'appuie sur un (méta)modèle, et un noyau mathématique, simple qui décrit formellement les concepts de fonction et de composition (de fonctions): le lambda calcul, (Wadler, 1996). Ce métamodèle permet d'expliquer différents modèles d'exécution, i.e. l'évaluation d'une fonction. Par exemple, l'évaluation paresseuse, dans laquelle les paramètres ne sont calculés que si nécessaire, est à l'origine du modèle flots de données présenté dans la suite, (Uustalu et Vene, 2006) ; (Harisson, 2006) présente un modèle d'exécution parallèle. Parmi les caractéristiques des modèles fonctionnels, il est possible de: 1) passer des fonctions en paramètre d'autres fonctions, et 2) regrouper des fonctions en types (de données). Le premier point permet de décrire des traitements complexes plus simplement que dans un style impératif. Le second point permet de spécifier soit les différents types/classes d'éléments qui composent un système, soit des concepts de modélisation ; cette idée est à l'origine du concept de métamodèles fonctionnels utilisé dans la partie 3.

Comme UML pour l'approche objet, Haskell est (ou tend à devenir) le langage fonctionnel de référence, (Hudak *et al*., 2007). Haskell présente plusieurs avantages dans un cadre IDM dont les plus importants sont donnés par (Mathaikutty, 2005). En particulier, il repose sur des concepts peu nombreux, bien formalisés, et permettant de décrire simplement une information ou un calcul complexe. Parmi ces concepts, les fonctions sont des entités de première espèce: elles sont regroupées pour définir des types de données éventuellement récursifs et paramétrés, et peuvent être passées en paramètre ou en retour d'autres fonctions. La définition d'un type est similaire à la définition d'une grammaire, i.e. un ensemble d'expressions et un langage représentant les valeurs possibles pour ce type. Enfin, Haskell privilégie un style déclaratif: les fonctions peuvent être définies de façon compacte et intelligible par filtrage de motifs (pattern matching).

Plus précisément, Haskell (Bird *et al*., 1998) est un langage fonctionnel typé avec évaluation retardée (ou paresseuse). Un type correspond simplement à un ensemble de fonctions (appelées constructeurs) pour construire des expressions et des valeurs pour ce type. Par exemple, le type *Liste* et la fonction permettant de connaître la *longueur* d'une liste sont définis ci-dessous. Il faut noter que la plupart des langages fonctionnels intègrent des facilités pour définir et manipuler des listes

(ou toutes autres collections d'éléments: vecteurs, matrices, streams/signaux, etc.).

```
data Liste a = Vide | Paire a (Liste a)

longueur Vide           = 0
longueur (x `Paire` xs) = 1+longueur xs

longueur ('a' `Paire` 'b' `Paire` Vide)
-- retourne 2
```

La définition générique, paramétrée par le type a, spécifie une valeur (*Vide*) et une fonction (*Paire*) prenant en paramètre un élément a et une liste de a, et retournant une liste de a. En fait, la fonction *Paire* peut être appliquée à 0 paramètre (elle se manipule alors comme une variable et peut être utilisée par d'autres fonctions), 1 paramètre (dans ce cas, *Paire x* est une fonction qui ajoute x au début d'une liste) ou 2 paramètres (et représente alors une liste). La définition d'une fonction sur un type donné est décrite par des règles (ou pattern matching) sur les différents constructeurs possibles comme le montre la fonction longueur ci-dessus.

Pour conclure, le modèle suivant décrit, par exemple, la syntaxe et la sémantique du lambda calcul, le métamodèle sur lequel repose Haskell: un *Terme* du langage se compose de *Symboles*, de définitions de *Fonctions*, et *d'Applications* d'une fonction à un argument. L'évaluation d'une application consiste à remplacer le paramètre de la fonction par l'argument dans le corps de la fonction puis à l'évaluer.

```
data Terme = Symbole  String
           | Fonction String Terme
           | Application Terme Terme

eval (Application (Fonction x e) a) =
  eval (replaceByIn x a e)
eval x = x
```

Ce métamodèle fonctionnel est équivalent au modèle UML/IDM de la figure 3. Plus généralement, la relation entre métamodèle UML et interprétation Haskell consiste à traduire un modèle par une expression du langage, un langage de modélisation par un type, un langage de modélisation par la définition d'un type, et une transformation de modèles par une fonction est un ensemble de règles (pattern matching).

La partie suivante propose alors un framework composé de trois métamodèles décrits dans le langage fonctionnel Haskell et permettant de modéliser puis simuler les systèmes informatiques.

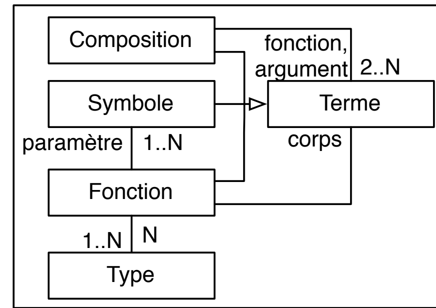

Figure 3. Métamodèle de langage fonctionnel.

3. FRAMEWORK FONCTIONNEL

Les modèles de systèmes, présentés dans la partie 2.1, peuvent être formalisés par un métamodèle (§2.2), puis mis en oeuvre et intégrés sous la forme d'un Haskell (§2.3). Cette partie propose alors un framework, et un ensemble d'outils, intégrant les concepts de flots de données (pour les comportements continus), d'automates à états-transitions (pour les comportements discrets), et d'actions (pour les interactions et les exécutions concurrentes).

3.1. Flots de données

Les flots de données (ou dataflows) sont un modèle standard dans le cadre des systèmes continus, (Thielemann, 2004). Ce type de modèle précise un ensemble de relations entre des séquences d'éléments. Comme l'explique la partie précédente, il est possible de modéliser ces séquences et ces relations par un métamodèle (qui précise la syntaxe et la sémantique de ce langage de modélisation) ; ce métamodèle peut être implémenté alors dans le langage fonctionnel Haskell et servir pour modéliser ou manipuler des modèles flots de données. Plus précisément, un flot est soit une séquence d'éléments (valeurs numériques ou matrices de valeurs, par exemple) soit une fonction qui calcule un élément en fonction du temps (Lee, 2003), (Mathaikutty, 2005). Par exemple, un flot représentant un échelon unité peut se représenter soit par une liste (*échelon=Paire 1 échelon*), soit par une fonction (*échelon' n=1*) ; pour obtenir la valeur numérique de ce signal à l'instant k, il suffit alors d'utiliser en Haskell, soit (*échelon !! k*) pour la première représentation, soit (*échelon' k*) pour la deuxième représentation. Il faut noter que la première représentation profite de l'évaluation paresseuse pour créer un flot infini (i.e. la réduction de échelon conduit à *échelon=Paire 1 (Paire 1 ...)*).

Les flots sont combinés à l'aide de blocs ; un bloc est soit une fonction mathématique (somme, intégration, multiplication par une constante, décalage d'index, etc.),

soit un assemblage de blocs (cf. Figure 2). La figure 4 présente un exemple de modèle flot de données, et bloc diagramme, pour un système du premier ordre (le modèle Haskell correspondant est précisé par la suite).

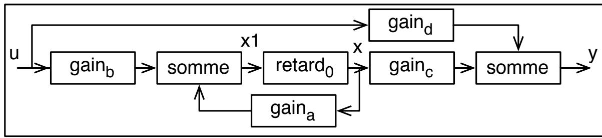

Figure 4. Exemple de flot de données.

Partant de cette description, la figure 5 présente le métamodèle UML correspondant. Ce dernier peut être utilisé alors par un outil IDM pour créer des modèles flots de données, ou implémenté sous la forme d'un DSL.

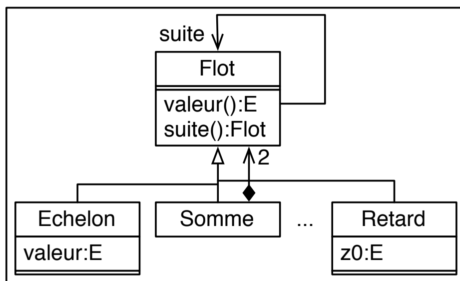

Figure 5. Métamodèle de flots de données.

L'interprétation fonctionnelle du métamodèle pour les flots de données, et proposée dans cet article, s'appuie sur le type Liste présenté dans la partie précédente. Pour rappel, une liste, interprétée comme un Flot et un signal ci-dessous, est définie récursivement à l'aide de la liste vide (noté en Haskell []) et de l'opérateur de construction (noté en Haskell :) qui ajoute un élément au début d'une liste. Le Pattern Matching permet, à partir de cette définition, de décrire les blocs élémentaires (somme, gain, retard, etc.) ; par exemple, 1:(2:[]) décrit la liste [1, 2] et une fonction f sur cette structure sera définie par deux règles pour [] et (x:xs). Les autres blocs et des modèles de systèmes sont définis alors en composant ces blocs fonctionnels comme le montre l'exemple du premier ordre ci-dessous. Le mot clef *data* correspond à une définition de type ; ainsi un Flot d'éléments de type a peut être vide [] ou (|) composé d'un élément a suivi (:) d'un Flot de a.

```
-- spécification d'un flot/signal
data Flot a = [] | a:(Flot a)

-- exemples de blocs élémentaires
somme (x:xs) (y:ys) = (x+y):(somme xs ys)
gain k (x:xs)       = (k*x):(gain k xs)
retard z0 x         = z0:x
```

```
-- exemple de bloc composite ; cf. figure 4
premierOrdre a b c d u  = y
  where x  = retard 0 x1
        x1 = somme (gain a x) (gain b u)
        y  = somme (gain c x) (gain d u)
```

Le métamodèle fonctionnel ainsi défini, malgré son apparente simplicité, permet de modéliser puis (contrairement à une approche IDM classique) de simuler directement des comportements continus. Comme illustration, le modèle suivant décrit comment obtenir les 5 premières valeurs de la réponse à un échelon unité u d'un premier ordre. Ces valeurs peuvent être sauvegardées dans un fichier puis éditées avec un tableur pour visualiser la courbe de réponse ; les figures 8 et 11 ont été obtenues de cette manière.

```
systeme = premierOrdre 0.9 0.1 1 0
u = 1.0:u   -- ou Paire 1 u
y = systeme u
take 5 y
-- retourne [0, 0.1, 0.19, 0.27, 0.34]
```

L'intérêt du métamodèle proposé est que des composants discrets, i.e. des automates à états-transitions, peuvent être intégrés dans des blocs. La partie suivante reprend alors la démarche suivie ; à savoir, définir précisément ce qu'est un comportement discret (à l'aide d'un diagramme des classes UML) puis fournir une implémentation fonctionnelle Haskell pour permettre la modélisation et la simulation de comportements discrets (ou discrets+continus).

3.2. Automates à Etats-Transitions

Les comportements discrets sont généralement représentés par un automate états-transitions. Un automate A est une structure composée d'un ensemble d'états Q, d'un ensemble d'événements E appelé aussi alphabet, d'un ensemble de transitions $T \subseteq Q \times E \times Q$, et d'un état initial $q \in Q$. La figure 6 présente un exemple d'automate à deux états Q={0, 1} et deux événements E={min, max} ; cet automate est associé, dans le cas de l'illustration considérée, à un état continu y et les événements sont émis lorsque y<=0.1 (pour min) ou y>=0.9 (pour max). Chaque état précise la valeur de sortie (cf. figure 1) en fonction du temps et la courbe de réponse en y est donnée par la figure 8; cette courbe a été obtenue à l'aide du métamodèle fonctionnel de la figure 7 et du code Haskell correspondant.

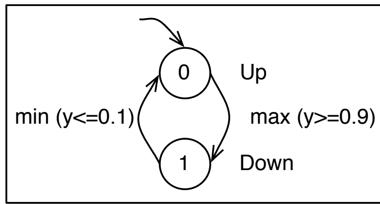

Figure 6. Exemple d'automate.

Les automates permettent de représenter les systèmes ayant un nombre discret d'états, et à temps discret (cf. Tableau 1). Pour reprendre les concepts de la partie précédente, un automate est un bloc piloté par un flot d'événements et pouvant générer en sortie un flot de valeurs quelconques. Il existe plusieurs modèles d'exécution possibles pour décrire l'évolution d'un automate (i.e. de son état interne). Un modèle simple consiste à tirer la première transition franchissable ; une transition (s,e,t) est franchissable lorsque l'état courant q de l'automate est s et que l'entrée est e. Le franchissement remplace q par t, et modifie la valeur de sortie. Comme pour les flots de données, il est possible de formaliser les automates à états-transitions par un métamodèle (figure 7) puis de l'implémenter dans le langage Haskell ; ceci pour étendre le framework et le métamodèle fonctionnel de la section 3.1.

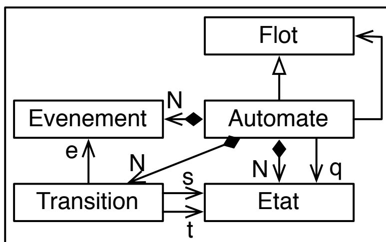

Figure 7. Métamodèle d'automates à états-transitions.

Le métamodèle fonctionnel correspondant est donné ci-dessous ; par soucis de lisibilité, l'ensemble des états Q et l'alphabet E ont été enlevés. Le mot clef *type* définit un type synonyme et permet d'accroître l'intelligibilité des modèles présentés ; ici, *Transition* est une fonction (->) qui prend une paire composée d'un état et d'un événement (d'entrée) et retourne une paire composée d'un état et d'une valeur de sortie. La fonction *compile* est utilisée ici pour transformer une structure d'automate en un flot de données compatible avec les éléments de la partie précédente.

```
type Transition=(Etat,Evenement)->(Etat,Sortie)
data Automate = Automate Etat Transition (Flot
Evenement)

compile :: FSM -> Flot Sortie
compile (FSM e t (i:is)) =
  let (e',o) = t (e,i)
      os = compile (FSM e' t is) in o:os
```

Le métamodèle proposé permet de décrire puis de simuler des comportements discrets, et s'intègrent aux modèles continus de la partie précédente. Comme illustration, le modèle Haskell suivant décrit le commutateur des figures 1 et 6: lorsque la sortie y du système atteint un seuil haut (ici 0.9), l'entrée u passe à 0 ; lorsqu'elle atteint un seuil bas (ici, 0.1), elle passe à 1. L'ajout de ces éléments à l'exemple du premier ordre (*u=1:u* étant remplacé par *u=compile commutateur*) permet alors d'obtenir la courbe de réponse, figure 8.

```
t::Transition
t (0,y) = if y>=0.1 then (1,0) else (0,1)
t (1,y) = if y<=0.9 then (0,1) else (1,0)

commutateur :: Automate 0 t y
u = compile commutateur
```

Ainsi, les métamodèles proposés permettent de modéliser puis de simuler des comportements continus, discrets et continus+discrets. Par contre, ils ne permettent pas de décrire les interactions avec l'environnement ou la plate forme sur laquelle devront évoluer les différents blocs. Pour résoudre cette limitation, il est nécessaire de compléter le framework avec les concepts, et un métamodèle, d'actions et d'interactions.

3.3. Actions, concurrence et communication

Comme pour les sections précédentes, la modélisation des interactions doit être précisée à l'aide d'un métamodèle. La notion d'action, présente dans les langages/modèles impératifs, peut être modélisée dans un langage fonctionnel par une fonction transformant un état en un autre état plus un résultat. L'état représente alors la mémoire et les entrées/sorties à un instant donné. Un opérateur de combinaison, appelé *bind* ci-dessous, permet de récupérer le résultat d'une action et de composer séquentiellement deux actions. Des actions élémentaires sont, par exemple, l'accès à une variable placée en mémoire (*get*) et l'écriture d'une valeur dans une variable (*set*) ; celles-ci et l'opérateur de mise en séquence sont représentés par le métamodèle fonctionnel suivant.

```
type State    = [(Var,Val)]
type Action a = State -> (a,State)

get   v vs = (findIn v vs,vs)
put v e vs = ((),addIn v vs)
bind p f c = let (r,c')=p c in f r c'
```

Les blocs des sections 3.1 et 3.2 peuvent alors être interprétés comme des processus (i.e. des listes d'actions) communicants par variables partagées: une

variable, appelée généralement canal de communication, est modifiée par un processus et est lue par un autre. Par exemple, les processus P et Q de la figure 1 échangent des valeurs à travers les canaux y et u. Le modèle producteur *prod* ci-dessous décrit un processus (et un bloc) qui émet une suite de valeurs sur le canal *chan*, et le processus *cons* lit ces valeurs puis les affiche sur la sortie standard *output*. Le modèle du premier ordre (§3.1) associé au commutateur (§3.2) se représente alors de manière analogue.

```
type Process = [Action ()]
prod, cons::Process
prod = p 0
  where p x = (put "chan" x):(p (x+1))

cons = display:cons
  where display = bind (get "chan")
                       (put "output")
```

La formalisation des processus à l'aide de listes d'actions permet de proposer des modèles d'exécution parallèle comme l'interleaving qui consiste à imbriquer deux processus ; une sémantique possible est donnée par la fonction *parallel* ci-dessous. L'exemple suivant décrit alors comment exécuter 10 pas, ou actions, élémentaires pour obtenir l'état à cet instant ; la fonction *sequence* exécute une séquence d'actions et repose sur l'opérateur *bind*.

```
parallel (x:xs) (y:ys) = x:y:(parallel xs ys)
sequence :: Process->Action ()
run p = (sequence (take 10 p)) []

main = prod `parallel` cons
run main == [("output","012345"),("chan","5")]
```

Plus simplement et pour reprendre les éléments des sections 3.1 et 3.2, un modèle informatique de plate forme correspond à un ensemble de processus modélisés au niveau logique par des *Blocs* et des *Flots* (de valeurs ou d'événements) et au niveau physique par des flots *d'Actions* (les actions élémentaires étant la lecture des entrées, la mise à jour des variables internes et l'écriture des sorties). A partir de là, l'interprétation en terme de *Processus* du système utilisé comme illustration permet d'obtenir la courbe de réponse suivante (figure 8) ; celle-ci est obtenue simplement en observant la valeur du canal *y* (cf. figure 1).

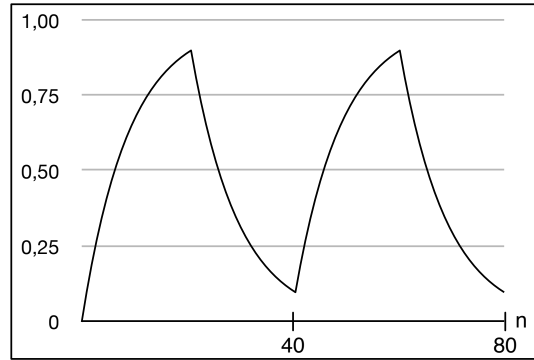

Figure 8. Courbe de réponse y(n) pour l'exemple.

La partie suivante propose une utilisation du framework avec un modèle informatique de plate forme robotique.

4. APPLICATION

4.1 Présentation du système

Le cadre et les outils présentés ont permis de modéliser puis simuler la commande informatique du robot hexapode de la figure 9, (Thiry *et al*., 2006). Les principaux blocs utilisés pour piloter le système sont donnés sur la figure 10. Le déplacement des différentes pattes dx_i est calculé à partir du modèle cinématique inverse f_i et de la consigne en vitesse (v,ω) pour la plate-forme. Ce déplacement est utilisé par le contrôleur de patte a_i qui fournit la position x_i en fonction du mode de fonctionnement (i.e. une patte peut pousser, attendre ou se déplacer) et d'un signal de contrôle s_i (pour la stabilité du robot, une patte permet ou interdit le déplacement de la patte voisine, i.e. (i+1) mod 6).

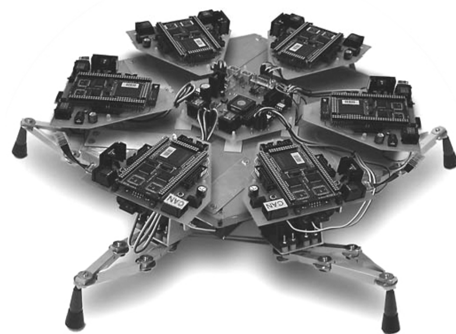

Figure 9. Robot hexapode modélisé.

4.2 Modèle fonctionnel

Le modèle Haskell suivant donne une réalisation synchrone (cf. §2.1) des blocs précédents: l'état d'une patte est définie par une position P et un mode E ; les synchronisations entre pattes sont définies par S et des flots [S] ; T et F correspondent au type automate de la

section 3.2 ; la fonction *comp* transforme les automates des différentes pattes (*legs*) en flots/signaux (*s1…s6*). La variable *states* contient alors l'état des 6 pattes pour chaque instant n ; par exemple, l'état de la patte i à l'instant n est obtenu par *(states !! i) !! n*, la fonction (!!) permet d'obtenir l'élément à la position p dans une liste, et donc aussi un flot. De la même manière, il est possible d'obtenir la position des pattes à appliquer à la plate-forme matérielle.

```
data E = Push | Await | Move -- State
data S = CanMove | Cant      -- Signal
type P = Int                 -- Position
type T = (E,P,S) -> (E,P,S)  -- Transition
data F = F E P T [S]         -- Automate

x0,xmax,xmin :: P
(x0,xmin,xmax) = (0,0,5)

tr :: T
tr (Push ,x,_) = if x>=xmax then
(Await,x,CanMove)
  else (Push,x+1,CanMove)
tr (Await,x,CanMove) = (Move ,x,Cant)
tr (Await,x,Cant)    = (Await,x,CanMove)
tr (Move ,x,_) = if x<=xmin then
(Push ,x,CanMove)
  else (Move,x-1,Cant)

comp :: F -> [(E,P,S)]
comp (F e p t (i:is)) = let (e',p',s) = t
(e,p,i)
  in (e',p',s):(comp (F e' p' t is))

legs = map (F Push x0 tr) [s6,s1,s2,s3,s4,s5]
[sa,s2,s3,s4,s5,s6] = map (third . comp) legs
s1 = CanMove:sa -- bloque si equitable !
states = map (first . comp) legs
```

Le modèle Haskell ci-dessus représente l'interprétation, avec le framework proposé, du modèle de la figure 10. Ce modèle Haskell, qui intègre des blocs continus et discrets, permet alors de simuler l'état des différentes pattes en fonction du temps.

4.3 Résultats

La figure 11 présente le mode de fonctionnement pour chacune des pattes: une valeur basse correspond à une poussée, une valeur moyenne à une attente, et une valeur haute à un déplacement. A l'instant 0, toutes les pattes poussent jusqu'à leur position postérieure extrême (x_{max} ci-dessus). A cet instant, les pattes 1-3-5 attendent et les pattes 2-4-6 se déplacent jusqu'à leur position antérieure extrême (x_{min}). A cet instant, les pattes 2-4-6 se posent puis poussent, les pattes 1-3-5 se déplacent jusqu'à x_{max} où les deux groupes de pattes commutent de fonctionnement poussée↔déplacement ; le système est alors en régime établi (appelé aussi tripode alterné).

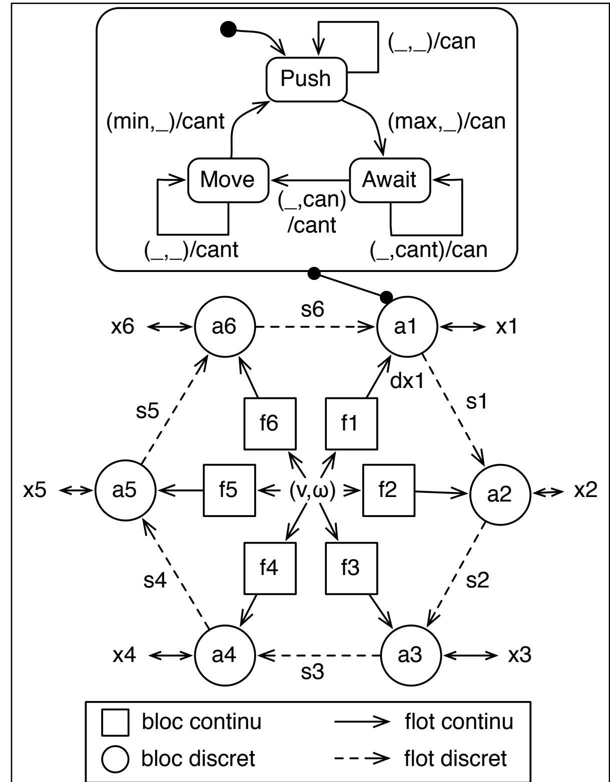

Figure 10. Architecture de commande.

Ainsi, le framework fonctionnel proposé permet de modéliser puis de simuler des systèmes à comportements complexes. L'intérêt de ce framework par rapport à des outils comme Matlab/Simulink est qu'il peut être facilement adapté ou étendu ; en particulier, le métamodèle *d'Actions* (§3.3) permet de préciser le modèle simulé pour tenir compte de la plate forme matérielle (et faciliter ainsi le ciblage).

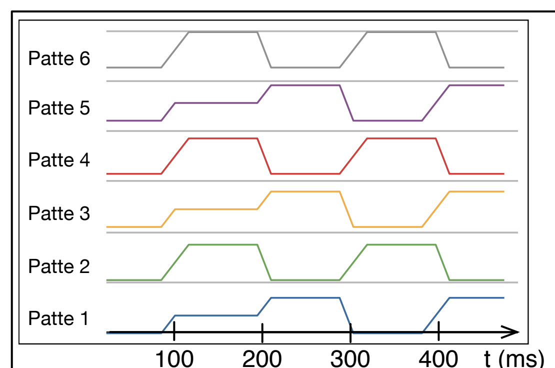

Figure 11. Réponse du système.

5. CONCLUSION

Cet article propose un framework fonctionnel dédié à la modélisation informatique de systèmes à comportements complexes (combinant à la fois des aspects continus et discrets, et prenant en compte les interactions et les exécutions concurrentes). Plus précisément, ce framework s'appuie sur trois langages de modélisation, appelés métamodèles dans le cadre IDM, avec les flots de données (pour les comportements continus), les automates états-transitions (pour les comportements discrets), et les actions (pour les entrées-sorties et les exécutions parallèles). Ces métamodèles sont traduits, sous la forme de Langages Spécifiques de Domaines (DSL), dans le langage fonctionnel moderne Haskell ; il est alors possible de modéliser puis de simuler des comportements continus/discrets, des interactions ou encore des exécutions concurrentes. Contrairement aux outils standards, ce framework s'appuie sur un langage de programmation de haut niveau: il est alors possible de profiter du langage pour étendre ou adapter le framework avec d'autres fonctionnalités. Le framework peut être utilisé pour étudier des systèmes non triviaux comme l'illustre le cas du robot hexapode de la partie 4.

A ce jour, d'autres métamodèles fonctionnels ont été étudiés et devraient être présentés avec, notamment, les aspects logiques/mathématiques ou graphiques. Par exemple, un métamodèle pour la logique temporelle est utilisé pour exprimer des propriétés et vérifier qu'un automate à états-transitions satisfait celles-ci, (Thiry *et al*., 2008). Les représentations graphiques pour les flots de données et les automates peuvent être obtenues en appliquant une transformation vers le langage DOT, un métamodèle de graphe qui permet une mise en forme automatique de schémas composés de noeuds et de liens.

Ainsi, l'utilisation du concept de métamodèle fonctionnel mis en oeuvre dans le framework permet de: 1) formaliser différents langages utilisés dans le domaine de la modélisation/simulation ; 2) simplifier la description des systèmes et leur étude en profitant des avantages d'un langage fonctionnel moderne.

RÉFÉRENCES

Bird R., Scruggs T.E., Mastropieri M.A., 1998, *Introduction to Functional Programming using Haskell*, Prentice Hall Eds

Breton E., Bézivin J., 2001, Towards an Understanding of Model Executability, *International Conference on Formal Ontology in Information Systems*, pp. 70-80

Denckla B., Mosterman P.J., 2005, Formalizing Causal Block Diagrams for Modeling a Class of Hybrid Dynamic Systems, *IEEE Conference on Decision and Control*

Deursen A., Klint P., Visser J., 2000, Domain-Specific Language: an annoted bibliography, *SIGPLAN Notices*, Vol. 35, N°6, pp. 26-36

Dubey A., 2005, *Metamodel Based Language ans Computation Platform for Algorithmic Analysis of Hybrid Systems*, PhD Thesis of the Faculty of the Graduate School of Vanderbilt University

Harisson W.L., 2006, The Essence of Multitasking, *Lecture Notes on Computer Science*, Vol. 4019, Springer Eds, pp. 158-172

Hudak P., Hughes J., Peyton-Jones S., Wadler P., *A*, 2007, History of Haskell: Being Lazy with Class, *3rd ACM Sigplan History of Programming Language*, pp. 1-55

Hudak P., Courtney A., Nilsson H., Peterson J., 2003, Arrows, Robots and Functional Reactive Programming, *Lecture Notes on Computer Science*, Vol. 2638, Springer Eds, pp. 158-172

Jantsch A., Sander I., 2005, Model of Computation and Languages for Embedded Systems Design, *Computer and Digital Techniques*, Vol. 152, Issue 2, pp. 114-129

Lee E.A., 2003, *Structure and Interpretation of Signals and Systems*, Addison-Wesley Eds.

Maler O., 1998, A Unified Approach for Studying Discrete and Continuous Dynamical Systems, *Proceedings of the 37th IEEE Conference on Decision and Control*, Vol. 2, pp. 2083-2088

Mathaikutty D.A., 2005, *Functional Programming and Metamodeling frameworks for System Design*, PhD Thesis of the Faculty of Virginia Polytechnic Institute and State University

Matworks, Matlab, www.matworks.fr

Mellor S.J., Clark A.N., Futagami T., 2003, Guest Editors' Introduction: Model-driven development, *IEEE Software*, Vol. 20, N°5, pp. 14-18

Navet N., 2006, *Systèmes Temps-Réel: Techniques de description et de vérification*, Lavoisier Eds.

OMG - Object Management Group, 2007, UML 2.1.1 Superstructure specification, *disponible à partir de www.uml.org*

Thielemann H., 2004, Audio Processing using Haskell, *7th International Conference on Digital Audio Effects (DAFx'04)*

Thiry L., Perronne J.M., Thirion B., 2006, IDM pour une conception intégrée des logiciels de commande, *Conférence Internationale Francophone d'Automatique (CIFA'06)*, Bordeaux, France

Thiry L., Thirion B., 2008, Functional (Meta)Models for the Development of Control Software, *International Federation of Automatic Control (IFAC'08 World Congress)*, Seoul, 6-11 juillet, à paraître

Uustalu T., Vene V., 2006, The essence of dataflow programming, *Lecture Notes on Computer Science*, Vol. 4164, Springer Eds, pp. 135-167

Vangheluwe H., de Lara J., 2003, Foundations of multi-paradigm modeling and simulation: computer automated multi-paradigm modelling, *Proceedings of the 35th conference onWinter simulation: driving innovation*, pp. 593-603

Varro D., Pataricza A., 2003, VPM: Mathematics of Metamodeling is Metamodeling Mathematic, *Journal of Software and Systems Modelling*, pp. 1–24

Wadler P., 1996, The essence of functional programming, *19th Symposium on Principles of Programming Languages*