UNE APPROCHE A BASE DE PATRONS DE CONCEPTION POUR LA HIERARCHIE DE FORMALISMES A EVENEMENTS DISCRETS

Maamar El Amine Hamri, Claudia Frydman, Soraya Saci

9e Conférence Internationale de Modélisation, Optimisation et SIMulation - MOSIM'12
06 au 08 Juin 2012 - Bordeaux - France
« Performance, interopérabilité et sécurité pour le développement durable »

UNE APPROCHE A BASE DE PATRONS DE CONCEPTION POUR LA HIERARCHIE DE FORMALISMES A EVENEMENTS DISCRETS

M. A. HAMRI C. FRYDMAN S. SACI
LSIS UMR 6168
Aix-Marseille Université
{amine.hamri, claudia.frydman, soraya.saci}@lsis.org

RESUME : dans cet article nous proposons une solution informatique à base du patron de conception état-événement que nous avons défini pour l'implémentation des machines à états faisant partie de la hiérarchie des formalismes à événements discrets. En effet nous montrons comment ce patron peut être facilement étendu pour définir de nouveaux éléments de la hiérarchie grâce au paradigme objet sans remettre en cause la structure existante. Un avantage majeur pour la maintenabilité et l'évolution du code objet représentant la machine à états.

MOTS-CLES : *Formalismes à événements discrets : SMTS, TMM, TSM, les patrons de conception.*

1 INTRODUCTION

A nos jours, la modélisation et la simulation à événements discrets se répandent de plus en plus dans plusieurs domaines de recherche et industriels (biologie, électronique, chaines logistiques, etc.).

Des théoriciens et concepteurs ont travaillé ensemble pour proposer durant ces dernières décennies une panoplie de formalismes et d'approches pour définir un cadre théorique couvrant les deux phases de modélisation et de simulation. A ce titre, on note les travaux de Zeigler à l'origine du formalisme DEVS (Zeigler, 1976) très populaire pour sa modularité et son expressivité. La puissance de ce formalisme a permis de nombreuses extensions tout en conservant sa sémantique opératoire. On note ainsi les extensions suivantes : Parallel DEVS (Chow et al. 1994), Fuzzy-DEVS (Kwon el al. 1994), GDEVS (Giambiasi et al., 2000), Min Max DEVS (Hamri et al., 2006), etc.

La proposition d'une nouvelle extension, nécessite le développement d'un nouveau simulateur ou la maintenance d'un existant pour tenir compte des nouveaux concepts proposés par l'extension du formalisme tant au niveau modèle qu'au niveau simulateur. En général, les développeurs de simulation préfèrent le développement de nouveaux simulateurs au lieu de réutiliser ceux qui existent afin d'éviter une maintenance coûteuse d'un code qui peut s'avérer inexploitable pour l'ajout de nouveaux concepts.

Afin d'éviter de telles pertes de temps et d'obtenir des solutions informatiques maintenables, nous proposons d'adopter une conception par patron lors du développement du simulateur et du modèle à simuler. En effet la conception par patron est une approche issue du génie logiciel avec objectif de promouvoir les bonnes pratiques de la conception orientée objet : dépendance faible entre objets, extension facile, lisibilité du code, etc.

Dans cet article nous nous intéressons aux travaux de Giambiasi (Giambiasi, 2009abc) dont il propose une hiérarchie de formalismes à événements discrets pour proposer une hiérarchie de patrons de conception et montrer leur avantage. En effet, ses articles démontrent qu'un grand nombre de modélisateurs ne font pas appel à tous les concepts du formalisme DEVS et décrivent seulement une sous-classe de modèles. Par exemple, la variable e, définissant le temps écoulé dans l'état, est rarement présente dans la fonction de transition.

Ce papier est organisé comme suit : les sections 2 et 3 présentent un rappel des formalismes STMS, TMM et TSM, et un état de l'art sur les patrons de conception du génie logiciel. La section 4 présente le patron état-événement et la section 5 une approche objet à base de ce patron pour l'implémentation des formalismes de la hiérarchie. Enfin, nous concluons sur l'apport du patron proposé.

2 PRESENTATION DE LA HIERARCHIE DE FORMALISMES A EVENEMENTS DISCRETS

Les formalismes de la hiérarchie proposés dans (Giambiasi, 2009a), offrent un cadre de modélisation simple et efficace tout en limitant la portée des concepts définis par le formalisme DEVS. En effet, souvent les modélisateurs emploient DEVS tout en faisant abstraction au temps écoulé dans les états. De plus, les modèles construits présentent souvent un nombre limité d'états. Les formalismes de la hiérarchie préconisent des concepts à champ d'application limité pour modéliser des sous-classes de problèmes dont l'emploi du DEVS peut être évité et par conséquent s'abstenir des concepts inutiles.
Nous rappelons brièvement ci-dessous cette hiérarchie en partant du formalisme le plus raffiné vers le plus général.

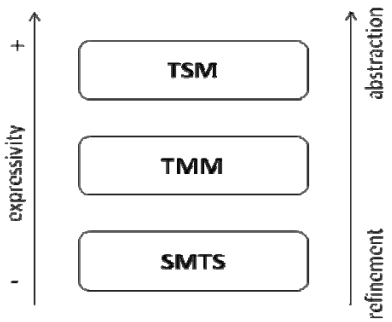


Figure 1 : Les trois formalismes hiérarchisés

2.1 La machine *SMTS* (*Sequential Machine with Transitory State)*

Le formalisme SMTS est une généralisation de la machine de Moore. Il propose de distinguer les états transitoires des états passifs avec une sémantique opératoire bien définie : les événements externes ne peuvent surgir que sur des états passifs et la durée de vie des états transitoires est égale à une valeur minimale ξ afin de respecter le principe de la non-instantanéité des systèmes.

Un modèle SMTS est défini par la structure suivante :

$SMTS = < X, Y, S, \delta_{int}, \delta_{ext}, \lambda_T >$ tels que:

X : est l'ensemble fini des événements d'entrée.
Y : est l'ensemble fini des événements de sortie.
S : est l'ensemble fini des états, avec:

$S = S_T \cup S_S$ et $S_T \cap S_S = \varnothing$

où S_T et S_S sont les sous-ensembles d'états transitoires et stables respectivement.
δ_{ext}: $S_S \times X \rightarrow S_T$ est la fonction de transition externe, qui spécifie que l'état change suite à l'occurrence d'un événement externe sur un état stable.
δ_{int}: $S_T \rightarrow S_S$ est la fonction de transition interne, qui définie le prochain état pour les états transitoires.
λ_T: $S_T \rightarrow Y$ est la fonction de sortie associée aux états transitoires.

2.2 La machine *TMM (Timed Moore Machine)*

La machine TMM présente l'avantage d'attribuer des durées de vie aux états transitoires des valeurs différentes à celle de la durée de vie conventionnelle ξ dans les machines SMTS. En effet une machine TMM définit clairement une fonction *lifetime* associée aux états et prend ses valeurs dans $R^+ \cup \infty$.
Une machine TMM est définie comme suit :

$TMM = < X, Y, S, \delta_{int}, \delta_{ext}, \lambda_T >$ tels que

X : est l'ensemble fini des événements d'entrée.
Y : est l'ensemble fini des événements de sortie.
S : est l'ensemble fini des états $S_S \cup S_T$, avec $s_i = (p_i, \sigma_i)$, $p_i \in V$ ensemble fini et σ_i la durée de vie
δ_{ext} : $S_S \times X \rightarrow S_T$ est la fonction de transition externe définie que pour les états stables.
δ_{int}: $S_T \rightarrow S_S$ est la fonction de transition interne définie que pour les états transitoires.
λ_T : $S_T \rightarrow Y$ est la fonction de sortie définie uniquement pour les états transitoires.
lifetime : $S \rightarrow R^+ \cup \infty$ est la fonction de durée de vie.
Ce formalisme comme DEVS fait appel aux concepts des états totaux $Q = \{(s_i, e) \mid s_i \in S, e \in R^+ \cup \infty\}$.
A noter que pour un état transitoire ne peut succéder qu'un état stable et vice versa. Ceci est garanti par les règles suivantes.

1- $\delta_{ext}(s_i,x) = s_j \Rightarrow s_i \in S_S \wedge s_j \in S_T$
2- $\delta_{int}(s_i) = s_j \Rightarrow s_i \in S_T \wedge s_j \in S_S$

2.3 *La machine TSM (Timed Sequential Machine)*

Elle est ainsi définie :

$TSM = < X, Y, S, \delta_{int}, \delta_{ext}, \lambda_T >$ tels que :

X : est l'ensemble fini des événements d'entrée.
Y : est l'ensemble fini des événements de sortie.
S : est l'ensemble fini des états avec : $s_i = (p_i, \sigma_i)$ et $p_i \in V$ et $\sigma_i \in R^+ \cup \infty$ est la durée de vie de l'état s_i
δ_{ext} : $S \times X \rightarrow S$ est la fonction de transition externe.
δ_{int} : $S_T \rightarrow S$ est la fonction de transition interne.
λ_T : $S_T \rightarrow Y$ est la fonction de sortie définie uniquement pour un état transitoire.

De même, le concept d'état total est introduit pour provoquer les changements d'état autonomes. Par contre aucune restriction n'est définie pour ce formalisme sur les deux fonctions δ_{ext} et δ_{int} en termes de succession d'état.

A noter que la fonction de transition externe δ_{ext} est définie pour les états stables et transitoires. Ceci permet d'avoir pour une même suite d'événements des comportements selon leur date d'occurrence.

2.4 Représentation graphique des machines à états

Etant donnés les ensembles X, Y, S sont des ensembles finis, une représentation graphique des machines définies ci-dessus peut être envisagée. Nous proposons les relations de correspondance suivantes (voir tableau 1).

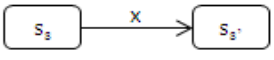
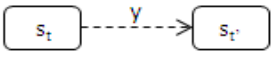
Elément de la hiérarchie	Forme graphique
$s_s \in S_S$	
$s_t \in S_T$ $lifetime(s_s) = \alpha$	
$\delta_{ext}(s_s, x) = s_{s'}$	
$\delta_{int}(s_t) = s_{t'}$ $\lambda(s_t) = y$	

Tableau 1 : Représentation graphique d'une machine de la hiérarchie

2.5 Etude comparative entre les formalismes de la hiérarchie

Ces formalismes définissent la même structure, un six-uplet composé des trois ensembles d'événements d'entrée X et de sortie Y, et d'états et trois fonctions définissant le comportement, la fonction de transition décomposé en deux sous-fonctions δ_{ext} et δ_{int}, et la fonction de sortie λ_T. Néanmoins chaque formalisme définit ses propres règles pour construire un modèle. En effet, le niveau d'expressivité en termes de temps et de comportement diffère d'un formalisme à l'autre. Nous présentons les principales différences dans le tableau suivant :

Formalisme	Temps	Comportement
STMS	-Indéfini, seulement l'ordre des événements est important.	$\delta_{ext}(s_i,x) = s_j$ $\Rightarrow s_i \in S_s \wedge s_j \in S_T$ $\delta_{int}(s_i) = s_j$ $\Rightarrow s_i \in S_T \wedge s_j \in S_S$
TMM	-Explicité par l'introduction du concept de l'état total. -Une fonction *lifetime* est associée à chaque état.	
TSM		Aucune restriction

Tableau 2 : Différences entre les formalismes de la hiérarchie

Par nature ces machines à états sont simulables donc une représentation informatique s'imposent. Dans ce qui suit nous proposons une approche originale pour la conception des machines à états en s'inspirant des travaux réalisés dans le génie logiciel.

3 Pourquoi une conception par patron des formalismes de la hiérarchie ?

Généralement, coder une machine à états finis consiste à utiliser une variable pour mémoriser l'état courant de la machine, ensuite selon l'événement reçu une série de tests est effectué pour identifier le nouvel état (ou le bloc d'instruction à exécuter) dont on fait souvent appel à l'instruction *switch case*. Cependant, l'inconvénient de cette approche malgré sa simplicité est la difficulté de maintenir le code associé à la machine à états. En effet, un tel code est difficile à lire et à modifier lorsque des changements sont apportés à la description de la machine.

Partant de ce constant, le *Gang Of Four* a proposé le patron état pour décrire des objets dont le comportement dépend de l'état de l'objet lui-même tout évitant le recours au *switch case*. Ce patron très utilisé dans le domaine du génie logiciel a donné naissance à plusieurs variantes.

Dans ce qui suit nous rappelons les avantages d'une conception par patron et une description détaillée du patron état.

3.1 Conception par patron : une approche génie logiciel

Un patron de conception est un modèle, un moyen d'atteindre un objectif, une technique en élaborant des méthodes communes et efficaces pour résoudre des problèmes qui se ressemblent avec un contexte différent. L'initiateur de cette idée est C. Alexandre qui dans son fameux ouvrage *A Pattern Language: Towns, Buildings, Construction, Design Patterns* propose environ 250 patrons pour la construction d'immeuble (Alexander et al., 1977). Depuis, ce concept a été projeté au génie logiciel et de nombreux travaux de recherche ont été réalisés donnant lieu à une bibliothèque de patrons.
Par la suite, le *GOF* (Gamma et al., 1995) a synthétisé les patrons de conception de la littérature informatique sous l'ouvrage *design patterns: elements of reusable object-oriented software*, devenu la bible des concepteurs d'applications informatiques. Les auteurs décrivent 23 patrons de conception formatés dans un langage textuel dédié avec des exemples d'application. Les avantages et les inconvénients de chaque patron sont aussi discutés.

Par ailleurs, les avantages d'une conception par patron orientée objet sont bien connus par la communauté génie logiciel que nous rappelons ci-dessous :

- Une solution abstraite réutilisable et maintenable,
- Un langage commun qui facilite la communication entre développeurs.
- Des modèles approuvés par la communauté génie logiciel grâce à leur réutilisation dans de nombreux développements d'applications.
- Un code lisible dont la solution adoptée est facilement identifié.

Dans ce qui suit nous faisons appel aux patrons de conception comportementaux, et plus précisément au patron état pour vérifier la faisabilité de notre approche.

3.2 Le patron de conception état

Après analyse de la bibliothèque des patrons de conception de *GOF*, le patron état s'avère le plus approprié à nos besoins de conception des machines décrites ci-dessus en termes de code lisible ; et d'autre part le code objet est facilement étendu à d'autres machines de même genre ou différent. Le patron état a été proposé pour décrire des objets dont le comportement varie selon le contexte. Ce patron se présente comme une alternative à l'approche classique reposant sur l'instruction *switch case.* De nombreux patrons ont été proposés ayant comme origine le patron état. Nous citons à titre d'exemple les travaux de (Dyson and Anderson, 1998) dont ils proposent une panoplie de variantes ayant pour base le patron état.

Le patron état repose sur une idée originale, *mapper* les différents états de l'objet en classes instanciables. Par conséquent chaque état détermine l'état futur selon l'événement provoqué. Ceci évite de concentrer le

comportement de l'objet sur une classe mais sur plusieurs classes. De plus les changements d'état sont délégués aux états et non pas déclenchés par l'instruction *switch case*. En effet, ces états vont déclencher la méthode correspondante à l'événement reçu et retourne l'état futur de la transition à franchir. Ci-dessous nous rappelons brièvement la description du patron état (Gamma et al. 1995).

Nom : patron état
Problème : objet se comportant différemment suite à une action reçue de l'extérieure.
Solution :

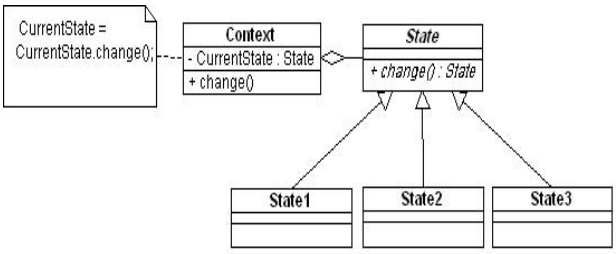


Figure 2 : Diagramme de classes du patron de conception état

Avantages :

- Un code lisible. Tous les états de l'objet sont implémentés sous forme de classes.
- Les objets sont faiblement couplés entre eux.
- Une maintenance facile à réaliser. L'ajout et la suppression d'états se font sans une remise en cause des classes déjà implémentées.

Cependant, nous estimons que cette solution souffre des lacunes suivantes :

- L'équivalence entre la machine à états et le code n'est pas directe. Certes les états peuvent être facilement identifiés sur le code, alors que les événements sont représentés par des méthodes et non par des objets comme tels.

- La surcharge des transitions par des expressions booléennes conditionnant les changements d'états, oblige le développeur à surcharger les méthodes d'événements. Par conséquent le code devient moins structuré et illisible.

- La composante date ne peut pas être représentée clairement dû au fait que les événements ne sont pas des objets ; donc impossible de tenir compte du concept événement daté. On pourra ajouter un paramètre aux méthodes d'événements pour représenter le temps mais ceci nuit au concept de réification du concept événement.

Après cette analyse, nous déduisons que la non réification du concept événement pose de sérieux problèmes pour une conception orientée objet d'une machine à états. Ces problèmes sont la non réutilisation du code pour d'autres types de machines d'états et par conséquent notre hiérarchie de formalismes. D'où la nécessité de définir un nouveau patron pour répondre à nos besoins.

4 LE PATRON ETAT-EVENEMENT

A l'image du patron état, nous proposons le patron état-événement (*state-event*). Cependant, au lieu de représenter seulement les états par des classes, nous réifions aussi les événements. Une telle conception permet d'obtenir un code de la machine à états structuré autour de classes état et événement. Ainsi, le principe du patron état-événement repose sur :

1- représenter la structure de la machine à états en étendant les deux classes abstraites `State` et `Event` pour représenter les ensembles d'états et d'événements, et
2- instancier pour chaque sous-classe événement l'ensemble des couples (état présent, état futur) à partir des sous-classes état instanciées. Cet ensemble peut prendre la forme d'un tableau ayant pour indice de parcours un objet de type état.

Une fois ces deux étapes accomplies par le développeur, la structure statique de la machine i.e. les ensembles d'états et d'événements, et les changements d'état possibles sont codés. Par conséquent nous obtenons un code objet opérationnel de la machine à états en question.

Par ailleurs, la dynamique de la machine à états est reproduite de la façon suivante : lorsqu'un événement est reçu i.e. un changement d'état à effectuer, l'objet correspondant à la machine à états délègue ce changement à l'événement reçu pour identifier le nouvel état sur la base de l'état courant de la machine. Cette solution peut paraître difficile à appréhender à première vue mais elle reprend complètement le comportement d'une transition. En effet, nous ne procédons pas par le couple (état, événement) pour identifier l'état futur mais à l'événement reçu d'identifier dans son ensemble de couples (état présent, état futur) l'état futur de la machine. Ainsi le code de la méthode `setChange(Event e)` se présente comme suit :

```
void setChange(Event e) {
 CurrentState = e.targetState(CurrentState);

}
```

Cette méthode encapsule tous les changements possibles de la machine à état. Il suffit de lui passer en paramètre l'événement reçu ; ensuite déléguer à ce dernier le changement d'état. Elle prend aussi en charge la mise à jour de l'état courant et le remplace par l'état futur.

Le patron état-événement que nous proposons est décrit de la manière suivante :

Nom : patron état-événement

Contexte : ce patron est utile pour concevoir des machines à états orientées objet dont de nouveaux comportements peuvent être ajoutés.
Solution :

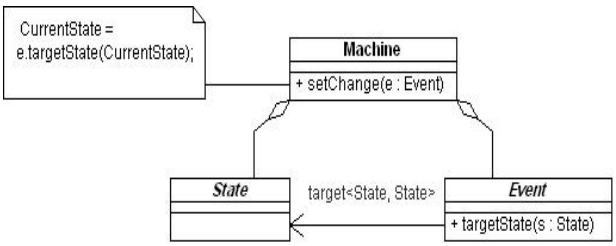


Figure 3 : Patron de conception état-événement

Avantages : le patron état-événement offre une meilleure structuration du code de la machine à états. En effet, le *mapping* de la machine à états vers le code objet est naturel. Grâce à la clarté du code structuré autour de deux sous-classes *State* et *Event*, le concepteur peut avec une simple lecture vérifier la conformité du code à la machine à états spécifié.

D'autre part, la maintenance de ce code est facile et ne remet pas en cause les classes étendues déjà existantes à partir de *State* et *Event*. De nouveaux comportements peuvent être spécifiés pour la machine à états à moindre effort de codage. Il suffit de concevoir les nouvelles sous-classes étendant les classes *State* et *Event*, et les incorporer au code de la machine. Ceci peut se faire en phase de pré- ou post-compilation.

A noter l'unicité des instances des sous-classes d'état correspondant aux différents états de la machine, nous a amenés à déléguer les changements d'état aux classes d'événements. Par ailleurs, le patron singleton peut être combiné avec le patron état-événement pour la création de ces instances.

Exemple I :
Soit la porte décrite par l'automate ci-dessous. Initialement, la porte est fermée. A chaque fois l'utilisateur appuie, la porte change d'état.

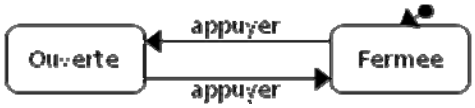


Figure 4 : Machine à états de la porte

Le diagramme de classes à base du patron état-événement pour cette machine est construit comme suit : nous étendons la classe abstraite *State* pour définir deux sous-classes Ouverte et Fermee. Ensuite nous faisons de même pour définir la sous-classe Appuyer en étendant la classe abstraite `Event`.

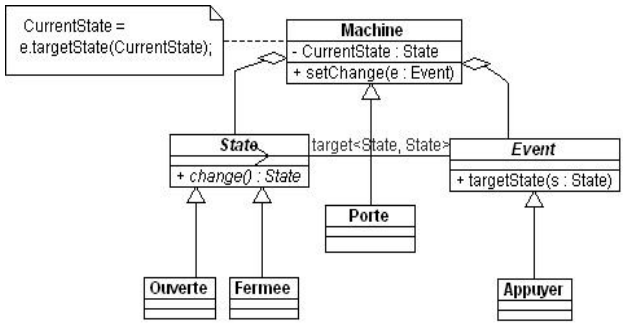


Figure 5 : Diagramme de classes de la porte

Pour la clarté de l'exemple nous avons introduit la classe Porte qui étend la classe Machine. Cette classe définit le comportement de la machine décrite par la figure 4. Une fois les instances des classes Ouverte et Fermee définies, la classe Appuyer remplit le tableau `target<State, State>` avec les changements d'état spécifiés par la machine. L'unicité des états garantit le fait que chaque indice représenté par l'instance état, retourne un seul objet correspondant à l'état futur. Ci-dessous une partie du code en Java de l'exemple de la porte.

```
public class Fermee extends State {
        public Fermee() {
                super();
        }
}
…
public class Appuyer extends Event {
        public Appuyer() {
        // Transition Fermee → Ouverte
          this.addTransition(Porte.getFERMEE(),
                        Porte.getOUVERTE());
        // Transition Ouverte → Fermee
         this.addTransition(Porte.getOUVERTE(),
                        Porte.getFERMEE());
        }
}

public class Porte extends Machine {
   private final static Fermee FERMEE = new
                                        Fermee();
   private final static Ouverte OUVERTE = new
                                       Ouverte();
        public Porte() {
           super();
           this.CurrentState = Porte.FERMEE;
        }

        …
}
```

La dynamique de ce code est générée par la création d'instance de la classe Appuyer. Ces instances sont passées en paramètre pour la méthode `setChange()` qui déclenche le changement d'état adéquat et met à jour la variable `CurrentState`. Le diagramme de séquence ci-dessous illustre l'interaction entre les objets Porte, Appuyer et Utilisateur déclencheur de changement d'état de la porte.

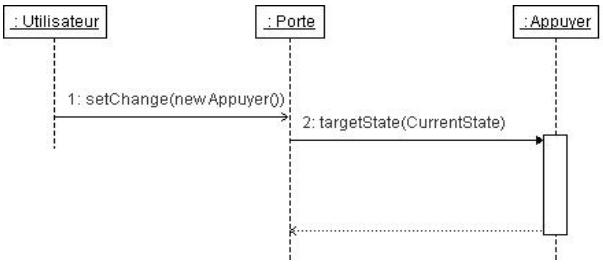


Figure 6 : Diagramme de séquence d'un changement d'état de la porte

Exemple II : Evolution de la machine à états de la porte et maintenance du code.

Nous souhaitons améliorer le comportement de la porte en ayant la possibilité de la maintenir ouverte par moments. Ainsi, nous proposons le nouvel automate pour tenir compte de ces modifications.

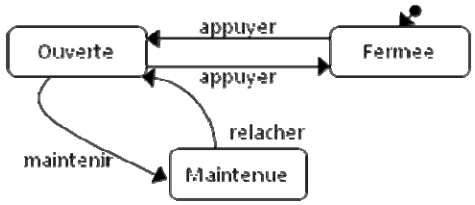


Figure 7 : Automate de la porte modifiée

Le code décrivant cet automate est dérivé du code précédent. En effet, nous définissons les classes suivantes : `Maintenue`, `Maintenir` et `Relacher` correspondant aux nouveaux éléments de l'automate : Maintenue, maintenir et relacher respectivement. Ensuite, nous initialisons, le tableau `target` de chaque événement ajouté avec les nouveaux changements d'état.
Par conséquent, le nouveau diagramme de classes est illustré par la figure ci-dessous.

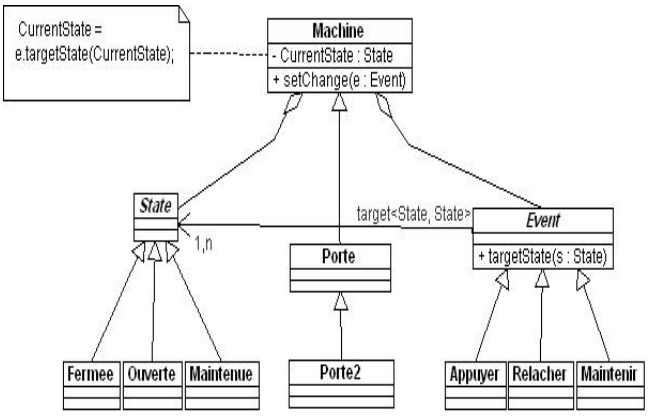


Figure 8 : Diagramme de classes de la porte modifiée

Grâce à la structure réfléchie du patron état-événement, les classes existantes Ouverte, Fermee et Appuyer ne sont pas remises en causes. L'ajout de nouveaux états et événements Maintenue, maintenir et relacher s'effectuent indépendamment des états et événements déjà conçus. Par contre, les changements d'état décrits par ces derniers peuvent être modifiés si cela est nécessaire. Par exemple, nous pourrons facilement préciser le fait que lorsque la porte est maintenue, l'événement appuyer ne provoque pas de changement d'état et donc le maintien de l'état précédent.

5 UNE APPROCHE OBJET POUR LES FORMALISMES DE LA HIERARCHIE

Rare sont les travaux de recherche qui se sont intéressés à la structuration du code en vue d'une simulation. Malgré les nombreuses extensions du formalisme DEVS, l'effort est souvent porté sur la conception du simulateur et non pas sur le code à simuler.

5.1 Une hiérarchie de patrons de conception

Dans ce qui suit, nous proposons une approche objet pour coder la hiérarchie de formalismes à événements discrets rappelés dans la section 2. Nous souhaitons montrer l'utilité du patron état-événement pour la conception de toute machine de la hiérarchie par simple extension des classes *State* et *Event*. Il suffit donc de typer ces classes selon les propriétés du formalisme choisi : distinguer les états transitoires, des états passifs, dater les événements, etc.

Nous procédons ainsi : chaque machine est encapsulée dans un package ou se trouve toutes les classes abstraites nécessaire au codage de la machine à états. Les packages sont organisés selon l'ordre de la hiérarchie du plus général au spécialisé (TSM→TMM→SMTS). Ceci permet une meilleure organisation du code. D'autre part, les spécificités de chaque formalisme sont pris en compte dans le package correspondant.

Reste à la charge du concepteur, l'extension de ces classes pour définir les éléments de sa machine i.e. les ensembles d'événements d'entrée et de sortie, et d'états en faisant appel au package adéquat ; ensuite initialiser le tableau `target` de chaque événement d'entrée seulement avec les changements d'état décrits par la machine. La particularité des formalismes de la hiérarchie nous a amenés à définir une nouvelle classe d'événement `IntEvent` (internal event) pour coder les changements d'état autonomes provoquant l'exécution de la fonction δ_{int}. Cette classe a pour attribut le tableau `target` qui stocke les couples $(s, \delta_{int}(s))$.

Par ailleurs, la définition de la fonction λ_T pour les différents formalismes de la hiérarchie demande la réécriture de la méthode `setChange(Event e)` responsable des changements d'état de l'automate. Afin de tenir compte des changements d'état autonomes qui provoquent la fonction de sortie λ_T, nous définissons ainsi le nouveau code de cette méthode comme suit :

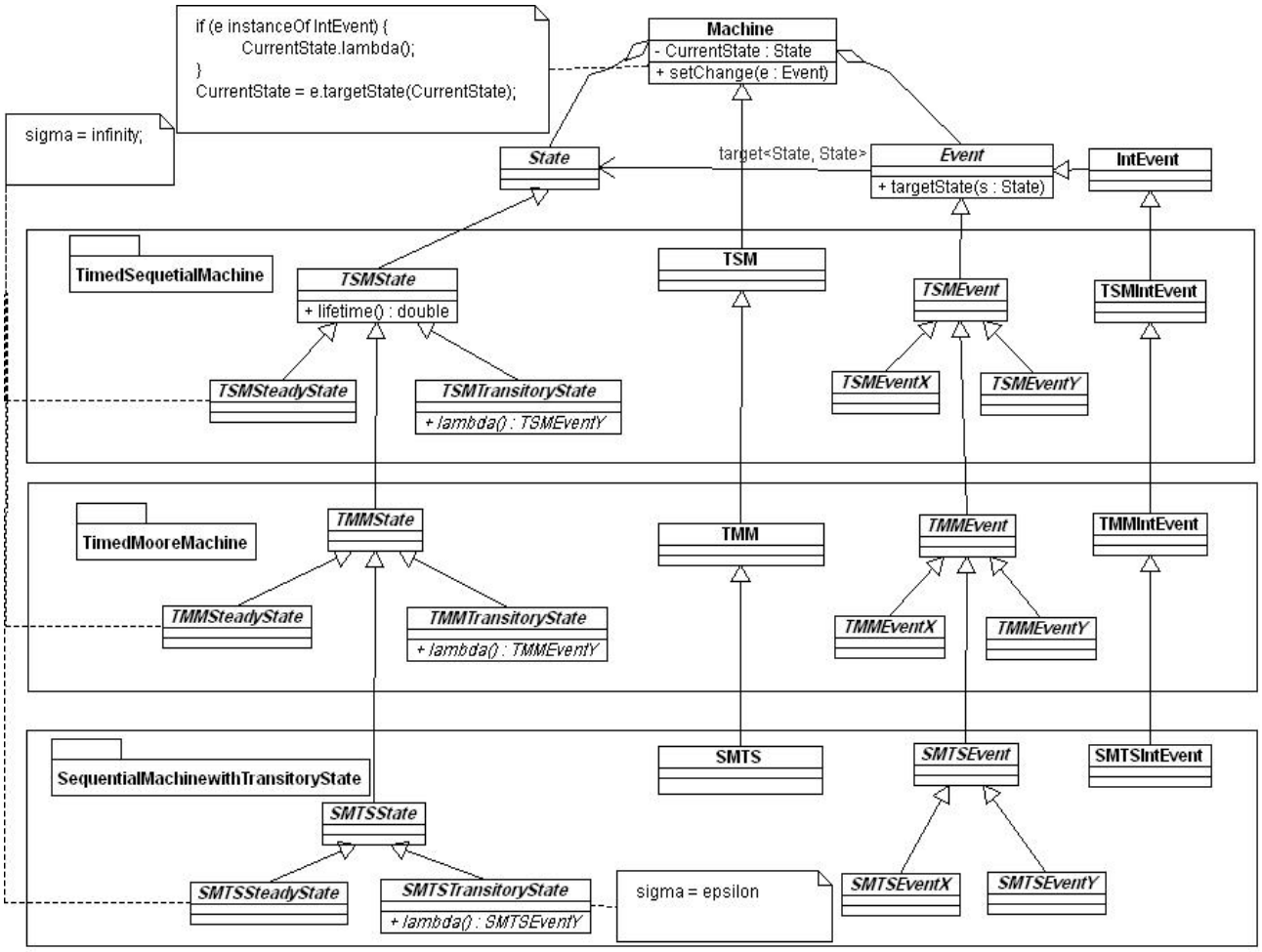


Figure 9 : Diagramme de classes de la hiérarchie de formalismes SED

```
    void setChange(Event e) {
     if (e instanceOf IntEvent) {
        CurrentState.lambda();
     }
     CurrentState = e.targetState(CurrentState);
}
```

En effet comme il est précisé pour tout formalisme temporel, un changement d'état autonome provoque l'exécution en parallèle de la fonction de sortie λ_T. Cette fonction est implémentée sous-forme de méthode que nous appelons *lambda()*. Cette méthode fait partie des classes d'états transitoires, elle n'est pas définie pour les classes d'états stables. Elle retourne l'événement de sortie spécifié. Si aucun événement de sortie n'a été précisé pour un tel changement d'état, la méthode *lambda()* retourne le pointeur *null*.

5.2 Les apports de l'approche

Le patron état-événement se présente pour notre solution comme un méta-patron pour le codage des machines de la hiérarchie. Ses avantages indiqués en section 5 se répercutent sur les sous-classes des machines à états TSM, TMM et SMTS. En effet l'ajout de nouveaux états et événement se manifestent par l'ajout de nouvelles classes ne remettant pas en causses celles existantes déjà.

La dynamique est assurée par la méthode `setChange()` commune à toutes les machines à états. Cette méthode déclenche la fonction de sortie λ_T et délègue le changement d'état à l'événement reçu. Ce dernier identifie l'état futur.

Cette approche est une alternative aux approches classiques de codage d'une machine à états (bloc d'instruction *switch case* et matricielle). Elle profite de la puissance du paradigme objet pour offrir un code évolutif et facile à maintenir.

D'autre part, l'approche permet le passage d'une machine particulière à une générale grâce à la relation d'héritage. En effet une classe de type SMTS ou TMM est de type TSM. En faisant appel à la technique de *cast*, les objets d'états et d'événements de niveau *i* peuvent être ramenés à un type supérieur. Ceci s'avère utile, dans le cas où le concepteur a fait appel à une machine de niveau i dans un premier temps, qui doit s'en passer pour une machine de niveau i+1 ou plus. Le

concepteur peut reprendre le code de sa machine initiale et l'adopter ou le compléter par de nouvelles classes pour coder les comportements ajoutés non supportés par le formalisme initial. Autrement dit, le code d'une machine de niveau i est incorporable par *cast* dans une machine de niveau supérieur.

6 CONCLUSION

La hiérarchie de formalismes à événements discrets définis par (Giambiasi, 2009a) propose un ensemble de formalismes structurant le travail du modélisateur. En effet, elle offre l'avantage de faire appel qu'aux concepts utiles au problème lors de la construction des modèles. Néanmoins, elle ne propose pas un cadre pour l'implémentation des modèles à événements discrets.

Afin de remédier à ce problème, nous avons généralisé le patron de conception état en patron état-événement. L'avantage majeur de ce patron est la facilité de maintenance du code de la machine à états, contrairement au patron état qui remet en cause l'interface de la classe abstraite `State` et par conséquent toutes les classes existantes lors de l'ajout de nouveaux événements. De plus, ce patron peut être adopté par la communauté du génie logiciel comme un standard pour l'implémentation d'objet dont le comportement dépend de l'état courant.

Ensuite, nous nous sommes appuyés sur ce nouveau patron pour définir une approche objet pour implémenter la hiérarchie des formalismes SMTS, TMM et TSM. Cette approche montre toute la puissance du patron état-événement pour définir de nouveaux patrons de conception afin de faciliter l'implémentation des formalismes notés ci-dessus.

Dans le futur proche, nous souhaitons réunir les efforts de la communauté de Modélisation et simulation à événements discrets autour de la standardisation de la phase de codage des modèles. Afin d'accroître la performance et la réutilisation des simulations existantes en adoptant des solutions maintenables.

REFERENCES

Alexander C., S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King and S. Angel, 1977. A Pattern Language: Towns, Buildings, Construction, Design Patterns. ISBN 0-19-501919-9.

Chow ACH. And B. Zeigler, 1994. Parallel DEVS: a parallel, hierarchical, modular, modeling formalism. 26th conference on Winter simulation, WSC'94.

Dyson P. and B. Peterson, 1998. State Patterns. Pattern Languages of Programming Design 3, edited by Martin, Riehle and Buschmann. Addison Wesley.

Gamma G., R. Helm, R. Johson and J. Vlissides, 1995, Design patterns: Elements of reusable Object-oriented Software. Addison-Wesley.

Giambiasi, N., B. Escude and S. Ghosh, 2000. G-DEVS A Generalized Discrete Event Specification for Accurate Modeling of Dynamic Systems. Transactions of the Society for Computer Simulation International, 17 (3) pp. 120-134.

Giambiasi N., 2009a. From Sequential Machines to DEVS Formalism, International Simulation Multi-conference, SCS.

Giambiasi N., 2009b. TMM: Temporal Moore Machines. 13th IFAC Symposium on Information Control Problems in Manufacturing.

Giambiasi N. 2009c. TSM: Temporal Sequential Machines. 21st European Modeling and Simulation Symposium, SCS, Canary Island, Tenerife.

Hamri M. A. and L. Baati, 2010. On Using Design Patterns for DEVS Modeling and Simulation Tools, Spring Simulation Multi-conference-Symposium Theory of Modeling & Simulation-DEVS Integrative M&S symposium, DEVS'10, Orlando, FL, USA.

Hamri M. A., N. Giambiasi and C. Frdyman, 2006. Min Max DEVS Modeling and Simulation. SIMPAT Elseiver.

Henney K., 2000. Collection for States. Patterns in Java. Java report.

Henney, K., 2003 Methods for States - A Pattern for Realizing Object Lifecycles. In vikingPlop'03. Bergen Norway, March 2003.

Kwon, Y. W., H. C. Park, S. H. Jung and T. G. Kim 1996. Fuzzy-DEVS Formalism: Concepts, Realization and Applications. AI Simulation and Planning, AIS'96.

Palfinger G., 1997. State Action Mapper. In Patterns Programming Languages, plop'97.

Van Gurp J. and J. Bosch, 1999. On the Implementation of Finite State Machines. IASTED'99.

Zeigler B., 1976. Theory of Modelling and Simulation. 1st edition, Robert E. Kreiger publishing company, Florida, USA.

Zeigler B., H. Praehofer and T. G. Kim 2000. Theory of Modeling and Simulation. 2nd Edition, Academic Press, New York, USA.