Continuations pour la programmation de comportement d'agent – intégration à la plate-forme Jade

Denis Jouvin

LIRIS (CNRS UMR 2508),
Bâtiment Nautibus, université Claude Bernard Lyon I,
43 boulevard du 11 novembre 1918,
69621 Villeurbanne

denis.jouvin@liris.cnrs.fr

RÉSUMÉ. *Les continuations sont un concept de programmation bien établi permettant de capturer explicitement l'état du programme en cours. Elles sont présentes dans des langages de programmation fonctionnelle (par exemple Scheme), dans le modèle d'acteurs de Hewitt, et depuis peu dans des langages dynamiques (tels que Ruby, Smalltalk, Python, et même Javascript ou Java). Elles ont été historiquement appliquées à la programmation d'automates, aux* threads *coopératifs, à des techniques de compilation, et ont dernièrement suscité un regain d'intérêt pour la programmation d'applications Web. Cet article montre comment ce concept s'avère particulièrement utile et élégant pour programmer le comportement d'agents (ou leurs composants comportementaux), au point d'en révolutionner l'écriture et la lisibilité. L'approche proposée, appliquée ici à la plate-forme multi-agents Jade, facilite notamment l'implémentation modulaire de protocoles d'interactions, une des difficultés majeures de l'ingénierie d'agents conversationnels.*

ABSTRACT. *Continuations are a well established programming concept that allows capturing and resuming the current program state. They can be found in several functional programming languages (such as Scheme), in Hewitt actor model, and more recently in dynamic programming languages (such as Ruby, Smalltalk, Python, and even Javascript or Java). They have been historically applied to automaton programming, cooperative threads, compilation techniques, and have lastly raised interest for web application programming. This paper shows how this concept happens to be especially useful and elegant to program agent behaviors (or behavioral components), by increasing code readability and ease of writing. It is shown that the proposed approach, applied here to the Jade multi-agents platform, facilitates the implementation of interaction protocols in a modular way, one of the main difficulties in conversational agent engineering.*

MOTS-CLÉS : *continuations, systèmes multi-agents conversationnels, génie logiciel orienté agent, composants comportementaux, automates à base de continuations.*

KEYWORDS: *continuations, conversational multi-agent systems, agent oriented software engineering, behavioral component, continuation-based automatons.*

1. Introduction

Les systèmes multi-agents (SMA) et la programmation orientée agents sont une approche de programmation particulièrement adaptée à modéliser des systèmes complexes. Dans un SMA, les agents interagissent selon des modèles d'interactions plus ou moins élaborés, normalisés, plus riches que dans les modèles de programmation plus classiques, tels que ceux des systèmes distribués orientés objets ou composants.

Un des principaux enrichissements est notamment la réification dans les messages, ou dans l'environnement support de l'interaction, d'éléments du contexte local de l'interaction (initiateur, destinataires, identification de la conversation en cours, identification d'une réponse, etc.), auxquels pourra faire référence une sémantique de la communication, apportée par un éventuel langage de communication entre agents (ACL pour *Agent Communication Language*). Les actes de langages, et la réification du contexte local de l'interaction, permettent notamment de définir des normes ou des protocoles d'interaction, voire de raisonner sur la base de la sémantique associée, selon des modèles cognitifs plus ou moins sophistiqués.

Cet enrichissement est un avantage dans le cas de systèmes fortement dynamiques ou adaptatifs, car il permet une plus grande dynamicité et une plus grande flexibilité dans l'organisation du système, tout en préservant la forte autonomie des agents et leur interopérabilité (Jennings et al. 1998) (Luck et al. 2003). Mais il constitue également une source de difficulté importante dans la mise en œuvre d'un SMA (Jouvin, 2000). Pour pallier ces difficultés, de nombreuses plates-formes multi-agents proposent des approches componentielles, afin de maximiser la réutilisation de composants comportementaux abstraits définissant partiellement le comportement de l'agent par rapport à un type de conversation donné.

Dans cet article, notre objectif sera de mettre en lumière comment les *continuations*, un concept de programmation assez ancien mais ayant suscité récemment une certaine attention, peut faciliter considérablement l'écriture de tels composants. Nous montrerons également comment intégrer cette approche à la plate-forme multi-agents Jade, à titre d'exemple, et proposerons une implémentation possible deux composants comportementaux abstraits permettant de gérer des conversations bilatérales et multi bilatérales à l'aide de continuations.

L'article est structuré comme suit : la section 2 introduit les continuations, leurs variantes et applications courantes. La section 3 identifie les difficultés de mise en œuvre de SMA évoquées plus haut, leur cause, et les réponses apportées dans la littérature. La section 4 présente notre approche par continuations, de l'implémentation d'un automate à la définition d'un composant comportemental d'agent basé sur des continuations. La section 5 présente l'intégration de cette technique à la plate-forme Jade, en détaillant les deux composants abstraits proposés. La section 6 illustre cela d'une expérimentation effectuée avec notamment un protocole d'enchère, accompagnée de mesures de performance. Enfin la section 7 conclut et présente les limites et les perspectives de cette approche.

2. Introduction aux continuations

Les continuations sont un concept de programmation assez ancien (Strachey et al. 1974), que l'on retrouve notamment dans des langages comme Scheme, ou ML, dans le modèle d'acteurs (Hewitt, 1977), et dans différentes algèbres de processus.

Le principe consiste à capturer dans une variable ou un objet manipulable programmatiquement, appelé continuation, l'état du programme en cours, puis à être capable de reprendre son exécution, à partir de cet état, en activant la continuation. Plus précisément, nous désignerons par *contexte d'exécution* l'état courant ainsi capturé, par analogie avec le contexte d'exécution d'un *thread* ou d'un processus, et par *continuation* l'objet ou artefact permettant de réactiver ce contexte.

Des définitions informelles des continuations, comme par exemple « le reste du programme », ou encore « un `goto` avec des paramètres », sont fréquentes dans la littérature. Toutefois, aucune n'est réellement satisfaisante : la première sous-entend une exécution linéaire du programme, qu'il faudrait préalablement « dérouler », et la seconde n'exprime pas le fait que le contexte d'exécution ne se limite pas à la position dans le programme, mais capture aussi la pile d'appel et les variables locales.

REMARQUE – Les continuations ont plusieurs points communs avec les *threads*, mais sont par nature non préemptives. Le contexte d'exécution d'un *thread*, tout comme celui d'une continuation, nécessite de mémoriser la pile, et la position dans le programme. Une continuation sera généralement plus économe, en ne stockant qu'une petite partie de la pile, et permettra des changements de contexte plus rapides. Bien entendu, les spécificités exactes dépendent des implémentations.

2.1. *Variantes*

Ce concept peut être décliné en différentes variantes ou restrictions, comme les *co-routines*, ou les *générateurs*. Un générateur se comporte comme un itérateur, mais s'écrit comme une fonction retournant successivement plusieurs valeurs (Mertz, 2001). Le mot clé `yield` renvoie une valeur, tout comme `return`, mais provoque aussi la mémorisation de l'état dans lequel se trouve le générateur, de sorte que la prochaine invocation reprendra à partir de cet état. L'intérêt d'un générateur est que l'état de l'itérateur n'a pas à être géré explicitement par le programmeur, ce qui en simplifie l'écriture : il est alors possible d'utiliser les structures de contrôle naturelles du langage pour gérer les transitions entre les états de l'automate implicite sous-jacent (Mertz, 2002).

Une co-routine est une fonction qui mémorise son état courant lorsqu'elle appelle une autre co-routine. Contrairement aux générateurs, une co-routine appelée ne « retourne » pas. Elle doit explicitement appeler une autre co-routine, qui sera alors réactivée, en lui passant éventuellement des paramètres.

Les continuations restent la forme la plus générale, comme le montrent (Haynes et al. 1986) et (Allison, 1988). Elles permettent en effet de définir très simplement co-routines ou générateurs, alors que les générateurs sont plus limités.

2.2. *Implémentations et exemples*

A titre illustratif, nous emprunterons des exemples des langages incluant nativement les continuations, comme Scheme ou Ruby, ainsi qu'une restriction simplifiée des continuations : les générateurs de Python.

2.2.1. *Générateurs Python*

La figure 1 présente un exemple de générateur dans la colonne de gauche : en haut la définition du générateur – il s'agit d'une fonction contenant une ou plusieurs instructions `yield` –, et en dessous une boucle affichant les valeurs retournées itérativement par ce générateur, avec en italique la sortie console du programme. Python supportant les clôtures, ce générateur pourrait aussi référencer une variable du contexte englobant.

Exemple simple de générateur Python

```
def simple_gen(max):
   i = 1
   yield "Let's count.."
   while i < max:
      yield "Odd %d" % i
      i = i+1
      yield "Even %d" % i
      i = i+1
   yield "The end"
```

Exécution et sortie console

```
>>> for s in simple_gen(4):
      print s

Let's count..
Odd 1
Even 2
Odd 3
Even 4
The end
```

Equivalent n'utilisant pas de générateur

```
class without_gen:
   def __init__(self, max):
      self.isEven = False
      self.i = 0
      self.limit = max + 1

   def __iter__(self):
      return self

   def next(self):
      if self.i > self.limit:
         raise StopIteration()
      if self.i == 0:
         r = "Let's count.."
      elif self.i == self.limit:
         r = "The end"
      elif self.isEven:
         r = "Even %d" % self.i
         self.isEven = False
      else:
         r = "Odd %d" % self.i
         self.isEven = True
      self.i = self.i + 1
      return r
```

Exécution et sortie console (identique)

```
>>> for s in without_gen(4):
      print s

Let's count..
Odd 1
Even 2
Odd 3
Even 4
The end
```

Figure 1. *Exemple de générateur en Python*

La colonne de droite de la figure 1 montre un programme équivalent en Python, définissant un itérateur mais sans utiliser de générateur. La classe ainsi définie doit alors comporter des attributs stockant l'état de l'itérateur (les attributs `isEven` et `i`), et cet état doit être géré explicitement (tests sur les valeurs avec un `if … elif …`, incrémentation de `i` en avant dernière ligne, et affectations de `isEven`), ce qui a pour effet de rendre le code plus difficile à lire et à écrire.

2.2.2. *Continuations en Ruby*

Dans les langages à objets, la capture du contexte d'exécution a pour effet de bord d'interrompre le flot de contrôle, pour le ramener au point d'appel de l'objet exécutable considéré comme « continuable », de la même manière que `yield` dans un générateur. La continuation est en quelque sorte locale à cet objet exécutable. Dans les langages fonctionnels (et en Ruby), en revanche, une continuation est globale au programme. Elle est créée implicitement par une primitive qui appelle une procédure ou expression lambda, et lui passe la continuation en paramètre.

Coroutine par continuations en Ruby

```ruby
def task(c1, name)
   puts "Step 1 in "+name
   c2 = callcc{|cc| c1.call(cc)}
   puts "Step 2 in "+name
   callcc{|cc| c2.call(cc)}
end
```

Exécution et sortie console

```ruby
b = lambda{|x| task(x,"bar")}
task(b, "foo")
```

```
Step 1 in foo
Step 1 in bar
Step 2 in foo
Step 2 in bar
```

Programme équivalent par threads

```ruby
def task(name)
   puts "Step 1 in "+name
   Thread.pass
   puts "Step 2 in "+name
   Thread.pass
end
```

Exécution et sortie console

```ruby
f = Thread.new do task("foo") end
b = Thread.new do task("bar") end
f.join
```

```
Step 1 in foo
Step 1 in bar
Step 2 in foo
Step 2 in bar
```

Figure 2. *Exemple de continuation en Ruby*

La colonne de gauche de la figure 2 présente un programme définissant une *coroutine* triviale en utilisant les continuations de Ruby, avec en dessous la sortie console en italique. La primitive `callcc` (pour *call with current continuation*) appelle la clôture anonyme définie par le bloc entre accolades, ici l'appel à la continuation `c1` passée en paramètre de la fonction `task`, et lui passe la continuation courante en paramètre. L'activation de la continuation `c1`, `c1.call()`, a pour effet d'activer l'autre coroutine, qui sera exécutée là où elle avait été capturée, et renverra sa propre continuation, transmise par `callcc` et affectée à la variable locale `c2`.

NOTE. — Ce programme peut se traduire facilement en Scheme, en utilisant la primitive Scheme `call-with-current-continuation` à la place de `callcc`.

La colonne de droite de la figure 2 présente quant à elle un programme équivalent mais utilisant des *threads*. Cet exemple est simpliste et ne comporte pas de problèmes de synchronisation : l'écriture du code est comparable à la version par continuation. Toutefois dans le cas général l'utilisation de *threads* apporte certaines difficultés dans la programmation (par exemple de synchronisation sur les zones critiques), et fait l'objet de limites intrinsèques au niveau système (nombre limité de *threads*, encombrement mémoire, lenteur des changements de contexte).

2.2.3. *Autres langages*

A part Scheme et Ruby, d'autres langages dynamiques répandus supportent nativement les continuations. Citons par exemple Smalltalk, ML, Haskel, Perl version 6, ou encore une implémentation alternative de Python, appelée Stackless Python.

Assez récemment, les continuations ont été ajoutées, par extensions ou via de simples librairies, à des langages hôtes comme C, Javascript ou Java, qui ne les supportent pas nativement. Ce type d'extension, lié à des aspects fondamentaux du langage comme la gestion de la pile et le contrôle du flot d'exécution, nécessite soit une manipulation directe de la pile, soit un interpréteur de code modifié, soit des techniques de modification dynamique de classe ou de transformation de code, comme le montrent (Pettyjohn et al., 2005). Nous pouvons citer notamment :

– Flowscript, une version modifiée de Rhino, un interpréteur Javascript en Java, comprise dans le *framework* d'applications web Cocoon[1] ;

– La librairie C `setjmp`, avec les fonctions `setjmp` et `longjmp`, qui permet d'implémenter des co-routines limitées ;

– Le *framework* RIFE[2], qui permet l'usage de continuations en Java, grâce à une technique de modification dynamique de classe ;

– en enfin Javaflow[3], une librairie expérimentale permettant les continuations en Java, par modification dynamique de classe, que nous utiliserons par la suite.

La figure 3 présente une implémentation de générateur Java, utilisant Javaflow, proche des générateurs Python. La fonction de capture de la continuation courante, `Continuation.startWith(..)`, prend en paramètre un objet `Runnable`. À la différence de Ruby ou Scheme, la continuation représente l'état du `Runnable`, et non l'état de l'appelant. On notera dans cet exemple qu'il est nécessaire de stocker temporairement les valeurs de retour du générateur en dehors de la méthode `run()`, ici par l'attribut `result`, car Javaflow ne véhicule pas de valeur de retour.

La figure 4 reprend l'exemple de la figure 1, traduit cette fois en Java, et utilisant la classe `Generator` introduite figure 3. Il produit les mêmes sorties console. Pour fonctionner, ces deux classes doivent être instrumentées par Javaflow au moment du chargement des classes ou à la compilation.

[1] http://cocoon.apache.org/2.1/
[2] http://rifers.org/
[3] http://jakarta.apache.org/commons/sandbox/javaflow/

```
public abstract class Generator<T>
      implements Runnable, Iterator<T>, Iterable<T> {

   /** utilisé pour transférer la valeur de retour */
   transient private T result;

   /** continuation capturant l'état courant */
   private Continuation current = Continuation.startWith(this);

   /** renvoie la prochaine valeur */
   public T next() {
      if (!hasNext())
         throw new NoSuchElementException();
      T retval = result;
      current = Continuation.continueWith(current);
      return retval;
   }

   /** joue le rôle du mot-clé yield en python */
   protected void yield(T yieldedValue) {
      result = yieldedValue;
      Continuation.suspend();
   }

   /** permet d'implémenter {@link Iterable} */
   public Iterator<T> iterator() { return this; }

   /** une continuation nulle signifie la fin du générateur */
   public boolean hasNext() { return current != null; }

   /** pas de sens car pas de collection sous-jacente */
   public void remove() { throw new UnsupportedOperationException();}
}
```

Figure 3. *Implémentation simplifiée des générateurs en Java, utilisant Javaflow*

```
public class SimpleGenerator extends Generator<String> {
   private final int max;

   public SimpleGenerator(int max) {
      this.max = max ;
   }

   public void run() {
      yield("let's count..");
      for(int i=1; i<max; i++) {
         yield("Odd " + i++);
         yield("Even "+ i);
      }
      yield("the end");
   }
}
```

Figure 4. *Exemple de générateur en Java basé sur la classe abstraite de la figure 3*

2.3. *Applications courantes*

Les possibilités offertes par les continuations sont nombreuses : il est possible de réécrire les structures de contrôle classiques des langages (*while*, *for*, …), d'en concevoir d'autres (par exemple `goto` en Scheme), d'effectuer du *backtracking*, etc. Cependant leur intérêt apparaît dans principalement trois types d'applications :

– La programmation et la composition d'automates. Il s'agit probablement de l'application la plus directe et la plus naturelle des continuations, que nous mettrons à profit en section 4. Cette technique permet notamment de programmer des analyseurs syntaxiques, des validateurs, etc.

– Les *threads* coopératifs. Les co-routines permettent d'implémenter des *threads* coopératifs, non préemptifs, avec changement explicite de contexte d'exécution. Ces *threads* sont dits coopératifs car chaque *thread* doit rendre la main régulièrement et explicitement, sans quoi il pourrait s'accaparer le temps processeur. L'avantage se mesure surtout en termes de performance et d'encombrement : les *threads* coopératifs sont très économes, et leur changement de contexte très rapide, en comparaison de vrais *threads*. Cette technique est adaptée aux modèles concurrents non préemptifs, et à la programmation événementielle.

– Et enfin la programmation d'applications Web par continuations, décrite notamment dans (Tate, 2005). Historiquement, elle a été introduite par le *framework* d'applications Web Seaside[4], en Smalltalk, puis par d'autres comme Cocoon ou RIFE. Cette application assez récente a suscité un regain d'intérêt certain pour les continuations. La raison est simple : l'interaction entre un navigateur Web et un serveur d'application Web prend typiquement la forme d'une conversation, dont il est nécessaire de mémoriser l'état. Les scripts ou routines contrôlant l'enchaînement des pages peuvent donc utiliser avantageusement les continuations pour stocker implicitement cet état, et le programmeur n'a plus à s'en soucier. Le principe étant simple et élégant, il a été très rapidement adopté.

NOTE – Un résultat équivalent pourrait être obtenu avec des *threads*, en faisant correspondre à chaque capture de l'état courant un appel bloquant. Toutefois il n'est pas envisageable de bloquer un *thread* pour chaque état de conversation à mémoriser, car ils peuvent s'avérer très nombreux, en particulier si l'application Web permet l'usage du bouton *back* du navigateur.

[4] http://www.seaside.st/

3. Ingénierie de comportements d'agents : difficultés et approches classiques

Nous définissons ici par agent conversationnel, tout agent dont le comportement nécessite de mémoriser et de tenir compte du contexte local de l'interaction, que nous désignerons par *conversation*. Bien que ce contexte puisse prendre des formes diverses, nous nous focaliserons dans cette section sur les agents communiquant par messages asynchrones, pour lesquels une conversation sera constituée d'une série d'échanges de messages corrélés, c'est-à-dire liés au même contexte de réalisation d'une tâche collective donnée, entre plusieurs participants.

3.1. *Définition du problème*

L'expérience montre que, bien qu'exhibant de nombreuses propriétés intéressantes, les systèmes multi-agents conversationnels sont difficiles à mettre en œuvre. Nous pouvons identifier les difficultés suivantes liées au mode d'interaction et à la nécessité d'implémenter des protocoles d'interaction (Jouvin 2000) :

– Un agent joue fréquemment le rôle de « pivot » entre plusieurs conversations simultanées, qu'il doit synchroniser. De façon similaire, les conversations multi parties, c'est-à-dire comportant plus de deux rôles conversationnels distincts, impliquent également une synchronisation entre plusieurs sous conversations correspondant à chacun des rôles.

– Lors de conversations multi bilatérales, c'est-à-dire mettant en jeu un initiateur et un groupe de participants, il est également nécessaire de synchroniser plusieurs comportements simultanés, correspondant aux interactions avec les différents participants, même si chaque comportement pris indépendamment correspond au même rôle conversationnel, et suit donc le même protocole.

– Une conversation bilatérale avec un participant donné peut elle-même comporter du pseudo parallélisme : l'ordre des messages de certaines séquences n'est pas total, il peut correspondre à un entrelacement de plusieurs branches.

– Une grande partie de la complexité induite provient de la gestion asynchrone des erreurs : non-respect du protocole, temporisations, etc. Cet aspect étant transverse et disséminé au sein du code, il est très difficile à modulariser et à réutiliser.

Nous voyons ici que ces difficultés sont dues au parallélisme ou pseudo parallélisme interne à chaque agent, inhérent au mode d'interaction entre des agents autonomes — nous ne parlons pas ici du parallélisme correspondant à l'exécution en parallèle des différents agents, mais bien du parallélisme intra agent. En outre, les langages généralement utilisés pour la programmation de SMA ne possèdent pas les primitives et concepts adéquats pour le gérer.

NOTE – L'utilisation des langages concurrents, comme Erlang, pourraient répondre à une partie de ces manques. Mais ces langages ont eux aussi leurs défauts, et ne sont malheureusement pas assez répandus et utilisés pour avoir été choisis dans les plates-formes et *frameworks* multi-agents.

3.2. *Approches componentielles par automates*

Il est intéressant d'observer que la plupart des plates-formes de SMA proposent des approches componentielles, afin de favoriser la réutilisation de composants comportementaux abstraits, et de faciliter leur synchronisation, ce qui confirme l'importance des difficultés évoquées en 3.1. Nous désignons par composant comportemental un composant définissant partiellement le comportement de l'agent, relativement à telle ou telle conversation ou branche de conversation donnée.

Le problème devient alors : comment définir les comportements comportementaux de façon modulaire, c'est-à-dire découplés de l'agent et encapsulés dans des composants réutilisables, et comment les combiner ou les synchroniser de manière à permettre leur exécution cohérente au sein des agents.

Pour reprendre l'exemple des protocoles d'interaction FIPA, une plate-forme FIPA proposera typiquement des composants abstraits implémentant partiellement ces protocoles, et s'interfaçant avec le code spécifique de l'agent selon une technique de composition particulière à la plate-forme (*callbacks*, héritage, événements).

Ces bibliothèques de composants comportementaux permettent au programmeur de s'abstraire des aspects délicats inhérents à la gestion des conversations et protocoles, comme la gestion des *timeouts*, la gestion des erreurs, le suivi en parallèle de plusieurs participants, *etc*. Elles sont largement utilisées par exemple dans les plates-formes Bond (Bölöni et al., 2000), Zeus (Nwana et al., 2000), Jade (Bellifemine et al., 2001), FIPA-OS (Poslad et al., 2000), ou MadKit (Gutknecht 2001).

3.3. *Stratégies d'implémentation des composants comportementaux*

Nous pouvons distinguer deux stratégies d'implémentation des composants comportementaux, correspondant à deux styles de programmation différents :

– Soit un *thread* est affecté à chaque branche parallélisable de chaque composant comportemental. Un état d'attente (par exemple l'attente d'un message) correspondra à une fonction bloquante sur le *thread*. Le contexte conversationnel est alors implicitement déterminé par le contexte d'exécution du *thread*. Cette stratégie n'est pas toujours praticable car le nombre de *thread* disponibles sur un système est limité, et un nombre excessif de *threads* peut induire une perte de performance due à un ordonnancement excessif par rapport aux traitements effectués. Par contre, elle offre l'avantage de moins fragmenter le code que la seconde stratégie.

– Soit l'état conversationnel de chaque composant comportemental est mémorisé explicitement dans un automate : cela nécessite que le code soit fragmenté selon une structure dépendant de l'automate et du modèle de composant, de façon à pouvoir interrompre et reprendre son exécution à un état donné. Typiquement, des méthodes ou objets distincts représenteront les différentes transitions, eux-mêmes associés à des objets distincts représentant les états. Il s'agit de la technique la plus largement utilisée dans les plates-formes SMA actuelles.

3.4. *Exemples*

Dans le modèle de concurrence de la plate-forme multi-agents Jade, chaque agent dispose d'un *thread* d'exécution. Les composants comportementaux sont dénommés *behaviours*, et se composent par héritage de sous-classes spécifiques de la classe abstraite `Behaviour`, et par composition grâce à des `CompositeBehaviour`. Ces derniers ont en outre pour fonction de synchroniser les composants comportementaux enfants (par exemple `SequenceBehaviour`, `ParallelBehaviour`, etc.).

La figure 5 présente un exemple trivial de composant comportemental Jade, avec à droite l'automate correspondant. Cet exemple montre la représentation explicite de l'état de la conversation, et la fragmentation du code en trois blocs dans le `switch`, correspondant aux opérations 1, 2 et 3. Ici la classe `Behaviour` contribue très peu au comportement de l'automate, mais d'autres *behaviors*, tels que `ContractNetInitiator`, définissent de véritables implémentations abstraites de protocoles, reliées au code spécifique de l'agent par héritage ou par *callback*.

```
public class ThreeStepBehaviour extends Behaviour {
   private int step = 1;
   private int iteration = 3;

   public void action() {
      switch (step) {
      case 1:
         System.out.println("Operation 1");
         step = 2;
         break;
      case 2:
         System.out.println("Operation 2");
         step = 3
         break;
      case 3:
         System.out.println("Operation 3");
         step = 1;
         iteration--;
         break;
      }
   }

   public boolean done() {
      return iteration == 0;
   }
}
```

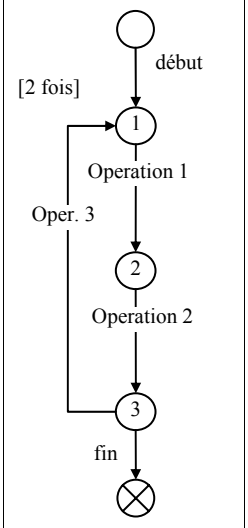

Figure 5. *Exemple de composant comportemental (*Behavior) de la plate-forme *Jade*

Des techniques comparables sont présentes dans la plupart des plates-formes de SMA. Nous pouvons citer notamment les automates multi plans (*multi-plan FSM*) de Bond, les graphes (*recursive transition network graphs*) de Zeus, les `Task` de FIPA-OS, ou encore les automates conçus avec SEdit dans MadKIT.

4. Approche par continuations

Comme nous l'avons évoqué en section 3, le besoin de pouvoir gérer le parallélisme sans monopoliser inutilement des *threads* implique qu'un composant comportemental d'agent soit manipulable comme un automate, sans attente ou appel bloquant sur un *thread*. Toutefois, il serait également souhaitable de pouvoir retrouver l'aisance d'écriture du flot des transitions de l'automate comme une simple fonction ou méthode, en utilisant les structures de contrôle naturelles du langage hôte, comme c'est le cas pour les générateurs.

Par rapport aux deux stratégies antagonistes de la section 3.3, les continuations apportent une solution particulièrement élégante, qui offre le meilleur des deux mondes. En effet, l'utilisation de continuations va nous permettre :

– de capturer implicitement le contexte d'exécution, et *a fortiori* l'état conversationnel, d'un composant comportemental, puis de reprendre son exécution dans cet état (pile d'appel et variables locales comprises). Le composant peut ainsi être utilisé comme un automate, sans pour autant nécessiter une manipulation explicite de son état par des attributs ou variables.

– de ne pas être sujets aux problèmes de synchronisation inter *threads* (accès concurrent à des structures ou objets, zones critiques, etc.), puisque ce modèle est non préemptif, et les transitions sont donc *in fine* exécutées séquentiellement.

A titre d'illustration, et afin de faciliter la compréhension de la section 5, les exemples et implémentations de référence que nous donnerons dans la suite de cette section utiliseront les continuations en Java du *framework* Javaflow.

4.1. *Automates comportementaux à base de continuations*

Comme nous l'avons vu, une continuation représente l'état du programme au moment de sa capture. Dans des langages à objet, tels que Java, une continuation est un objet particulier, immutable, qui n'est pas en lui-même un automate. Afin de définir un automate à partir de continuations, il est nécessaire d'introduire un objet automate englobant, encapsulant une ou plusieurs continuation(s), similaire à l'implémentation du générateur vue en 2.2. Le minimum requis pour gérer un automate simple avec Javaflow est donc une classe abstraite comprenant :

– un attribut `current` de type `Continuation`, référençant la continuation en cours de l'automate ;

– le *point d'activation* de l'automate, ici une méthode `activate()`, qui doit réactiver la continuation courante pour faire avancer l'automate d'un pas, reprenant ainsi l'exécution de la méthode `run()` jusqu'au prochain `yield()` ;

– une *méthode de capture* de l'état courant, `yield()`, qui interrompt le flot normal d'exécution dans la méthode `run()`, et renvoie le flot de contrôle à l'appelant de la méthode d'activation de l'automate. En pratique cette primitive pourra être appelée dans les méthodes de lecture de message.

A chaque exécution de l'automate depuis son point d'activation `activate()`, ce dernier effectue un pas. Chaque pas se termine soit par l'invocation de `yield()`, soit par la fin de la méthode `run()`, ce qui a dans les deux cas pour effet de repasser le contrôle à la méthode `activate()`. Cette dernière met à jour `current` avec une nouvelle continuation correspondant au nouvel état capturé, puis retourne normalement et rend le contrôle à la méthode appelante.

Afin de définir un composant comportemental, le programmeur doit étendre cette classe abstraite et implémenter la méthode `run()`, qui correspond au comportement souhaité de l'agent.

Dans cette méthode `run()` sont accessibles des méthodes (d'accès `protected`), qui affectent le flot d'exécution, et manipulent la ou les continuation(s) courante(s). À la seule fin de les distinguer, nous les qualifierons de *primitives*. Ces primitives doivent être appelées uniquement dans le cadre de la méthode `run()` (voir section 5.2.2). Pour l'instant nous avons défini uniquement la primitive `yield()`.

4.2. *Primitives de communication*

Pour être utilisable, un composant comportemental abstrait doit également comprendre un moyen de consommer des messages ou des événements lors de l'avancement de l'automate, comme le déclenchement d'un *timeout*, ou un signal de synchronisation provenant d'un autre composant comportemental. Ces consommations d'événements correspondent à des transitions dans notre automate, et appelleront la méthode `yield()` si l'événement n'est pas encore arrivé, ou si la précondition associée n'est pas remplie.

Nous définirons une primitive simple, `receive()`, permettant de lire un message ou un événement, avec mise en attente du composant comportemental si aucun message n'est disponible. Toutefois, étant donné qu'un agent peut comporter plusieurs comportements différents associés à des conversations ou branches de conversation différentes, une telle méthode sera d'une utilité réduite.

En pratique il est donc nécessaire d'ajouter à cela une famille de primitives `receive(…)`, prenant en paramètre un ensemble de préconditions à la transition, sous une forme ou une autre, ou tout autre mécanisme permettant d'associer une transition à des préconditions. Il peut s'agir par exemple de règles de présélection de messages. Une alternative consiste à fournir une méthode permettant de consulter des événements sans les consommer, ou encore d'effectuer du *backtracking* si le comportement s'aperçoit après coup que les événements ne le concernent pas, ce qui revient à déléguer à chaque consommateur potentiel la sélection. Toutefois cette méthode est peu performante si le nombre de consommateurs potentiels d'un message est élevé. La section suivante développe plus en détail ce problème.

L'envoi de messages ne pose quant à lui pas de problème de synchronisation, et n'implique pas de transition de l'automate. L'envoi se fera typiquement par une méthode ou un service fourni par la plate-forme SMA utilisée.

4.3. *Distribution des évènements et messages, ordonnancement*

La façon dont les messages, et par extension les événements, sont présélectionnés avant d'être distribués aux composants comportementaux de l'agent affecte directement la façon dont ces différents composants seront activés, c'est-à-dire leur ordonnancement. En effet, dans la mesure où ces derniers sont des automates non préemptifs, cela permet d'avoir un contrôle précis sur leur ordonnancement.

La plupart des plates-formes adoptent une stratégie d'ordonnancement neutre de type *round-robin*, peu performant si l'agent gère un grand nombre de composants. Certaines plates-formes permettent de spécialiser cet ordonnancement.

Plus les composants comportementaux pourront fournir de façon déclarative leur préconditions de transitions, règles de présélection ou filtres de messages, et plus l'ordonnanceur pourra optimiser cette présélection, à l'aide d'index par exemple. Cette présélection sera généralement basée sur des attributs discriminant les messages de cette conversation, ou branche de conversation, par rapport aux autres (par exemple, dans un environnement FIPA, les attributs `performative`, `protocol`, `conversation-id`, ou `sender`). Ainsi, il est probable qu'un seul composant comportemental, ou branche conversationnelle, réponde à ces critères et soit activé.

Dans le cas d'agents conversationnels, chaque agent est en principe doté d'une file d'attente de messages. Si un mécanisme de présélection des messages est utilisé, une file d'attente au niveau du composant comportemental, voire au niveau de chaque branche associée à chaque participant, peut aussi être envisagées.

4.4. *Extension : primitives de pseudo parallélisme et de synchronisation*

Pour pallier les difficultés liées au parallélisme évoquées en 3.1, il est nécessaire d'introduire des mécanismes de pseudo parallélisme et de synchronisation. Le cas de conversations multi bilatérales, entre un initiateur et un groupe de participants associés au même rôle, est en particulier très courant (*contract-net*, négociations, enchères, etc.), et nécessite ce type de synchronisation. Ce cas implique un parallélisme inhérent à la gestion des interactions avec *n* participants par l'initiateur, même si le comportement associé à chaque participant est le même. Afin d'en faciliter la programmation, nous proposons d'introduire les deux primitives suivantes :

– `parallelize()` : une primitive permettant de passer en mode parallèle, qui duplique l'automate courant (y compris la continuation sous-jacente), en *n* sous automates (donc *n* continuations), un pour chaque participant ;

– `join()` : la primitive inverse, valide uniquement en mode parallèle, permettant d'attendre (à l'aide de `yield()`) que tous les sous automates atteignent ce point, pour ensuite repasser en mode séquentiel standard. La dernière continuation ayant appelé `join()` est alors utilisée comme continuation globale.

Ces deux primitives permettent une gestion particulièrement élégante des conversations multi bilatérales car le déroulement de chaque sous automate associé

à chaque participant, en section parallèle, est implicite, pour le moins en ce qui concerne les variables locales. Les attributs restent quand à eux partagés par toutes les branches conversationnelles ainsi créées, du fait de Javaflow.

4.5. *Exemples et comparaison*

La figure 6 reprend l'exemple trivial de la figure 5, section 3.4, décrivant un comportement en boucle sur 3 états. En comparaison du composant Jade équivalent, la première observation est qu'aucun attribut n'est ici nécessaire. Les transitions sont gérées par la succession normale des instructions et une boucle *for*, ce qui améliore la lisibilité du code, et réduit le risque d'erreur.

```
public class My3Step extends ContinuableBehavior {
   public void run() {
      for(int i=0; i<3; i++) {
         System.out.println("Operation 1");
         yield();
         System.out.println("Operation 2");
         yield();
         System.out.println("Operation 3");
         yield();
      }
   }
}
```

Figure 6. *Comportement à 3 états de la section 3.4 : approche par continuations*

Un autre exemple, représenté figure 7, illustre le fait que cette approche permet de mieux profiter des vérifications statiques effectuées par le compilateur. Ici, la variable `info` représente une valeur mémorisée entre deux états successifs de l'automate. Le compilateur Java nous avertit ici que cette variable pourrait ne pas avoir été initialisée. De telles vérifications ne sont pas envisageables lorsque le code est fragmenté entre plusieurs objets ou méthodes.

```
public class VariableInitialization extends ContinuableBehaviour {
   public void run() {
      String info;
      ACLMessage msg = receive(); // première transition
      if(msg.getPerformative() == INFORM)
         info = msg.getContent();
      msg = receive(); // deuxième transition
      ACLMessage reply = msg.createReply();
      reply.setContent(info);
      send(reply);
   }
}
```

the local variable info may not have been initialized

Figure 7. *Variable non initialisée détectée statiquement par le compilateur*

NOTE – Au vu des exemples précédents, nous serions tentés de croire que cette approche implique que la gestion complète de l'automate tienne dans une méthode,

ce qui représenterait un inconvénient. En réalité ce n'est pas le cas : la pile d'appel étant restituée par la continuation, la gestion de l'automate peut être répartie dans autant de méthodes, éventuellement récursives, et d'objets que l'on souhaite. L'avantage est que ce découpage sera dicté par des considérations de modularité et d'encapsulation, et non imposé par les transitions et la structure de l'automate.

5. Intégration à la plate-forme multi-agents Jade

L'intégration de cette technique à la plate-forme multi-agents Jade, sous forme d'extension, présuppose principalement deux choses :

– avoir la possibilité d'utiliser des continuations dans l'environnement de Jade, sans interférer avec le fonctionnement de ce dernier (dans notre cas nous utiliserons pour cela le *framework* Javaflow) ;

– et avoir la possibilité de définir des classes de composants comportementaux héritant de la classe `Behavior`, respectant le contrat de cette dernière, qui intègrent l'utilisation de continuations.

Le premier point est, comme nous le verrons en section 5.2, un problème d'instrumentation de classes. Le deuxième point, quant à lui, peut être appréhendé en proposant une ou plusieurs classe abstraites intermédiaires (au même titre que les différentes sous classes abstraites de `Behaviour` proposées par Jade), intégrant les primitives décrites en section 4, et jouant ainsi le rôle de « colle logicielle » entre ces primitives, l'usage de continuation sous-jacent, et l'infrastructure des *behaviours* de Jade. Il s'agit en effet de ne pas substituer les difficultés de programmation d'agents par des difficultés de manipulation de continuations !

Nous proposerons deux composants comportementaux abstraits basés sur les continuations, correspondant aux deux cas d'utilisation les plus courants : un `ContinuableBehavior`, adapté aux conversations bilatérales simples, décrit en 5.3, ou au rôle de participant des conversations multi bilatérales ; et un `MultiContinuableBehavior`, correspondant au rôle d'initiateur de conversations multi bilatérales, telles que par exemple les enchères, décrit en 5.4.

5.1. *Modèle d'agent et de comportement de Jade*

La plate-forme Jade possède un modèle de concurrence particulièrement simple, en comparaison d'autres plates-formes multi-agents, puisqu'elle affecte par défaut un *thread* d'exécution à chaque agent actif, et qu'elle considère que tout comportement est un automate répondant à l'interface suivante :

– une méthode `action()`, l'activation pas à pas de l'automate ;

– et une méthode booléenne `done()`, renvoyant vrai si l'automate a terminé.

La classe `Agent` comprend quant à elle les méthodes suivantes :

– `setup()`, surchargée par les classes d'agent concrètes pour initialiser l'agent, en particulier créer et activer les comportements initiaux ;

– `addBehaviour(Behaviour)` et `removeBehaviour(Behaviour)`, permettant d'ajouter à l'agent, ou de lui supprimer, des comportements ;

– `receive()`, `receive(MessageTemplate)`, qui permettent de lire et consommer le message suivant dans la file d'attente de l'agent, renvoyant `null` si aucun n'est disponible. La deuxième forme associe un ensemble de règles de présélection, le

`MessageTemplate`, à cette lecture de message. À noter que ces deux méthodes ne sont pas bloquantes ;

– `blockingReceive()` et `blockingReceive(MessageTemplate)`, qui sont les versions bloquantes de deux méthodes précédentes, et qui ne devraient *en aucun cas* être appelées à partir de composants comportementaux automates, car elles bloquent totalement l'agent ;

– et enfin `send(ACLMessage)`, qui permet d'envoyer un message FIPA à un ou plusieurs destinataires.

À cela s'ajoutent des méthodes vides permettant d'intercepter des événements par polymorphisme (comme par exemple la terminaison d'un comportement), à surcharger dans les sous classes de `Agent` et de `Behaviour`.

5.1.1. *Limites*

L'utilisation de ce modèle est simple, mais il comprend plusieurs limites quelque peu regrettables. Tout d'abord le modèle de concurrence et d'activation des agents semble totalement fixé. Un SMA comprenant un grand nombre d'agents réactifs en souffrira. Il semble clair du reste que Jade n'a pas été conçue pour ce type de SMA. Il est donc difficile *a priori* de profiter d'un gain éventuel de performance par l'utilisation de continuations dans ce cadre là.

D'autre part, la file d'attente de messages des agents est peu accessible et offre des possibilités limitées. Il est difficile par exemple de la parcourir sans consommer de message. Ce dernier détail nous encourage à opter pour une présélection effective des messages avant distribution aux différents composants comportementaux.

Enfin Jade utilise largement l'héritage et le polymorphisme, ce qui en soit est assez limitatif : nous regretterons l'absence de sources d'évènements JavaBean, ou autres techniques de composition, entraînant un couplage moins fort.

5.1.2. *Composition de comportements*

Les sous classes de `CompositeBehaviour` permettent de composer plusieurs composants comportementaux enfants dans un composite. On notera en particulier la classe `FSMBehavior` qui permet de définir une machine à états finis de façon explicite (états et transitions). C'est en jouant sur ces structures de composition qu'il devient possible de parvenir à un certain degré de réutilisation de composants comportementaux, mais l'aspect synchronisation reste délicat à gérer.

5.2. *Mise en œuvre de Javaflow avec Jade*

Javaflow (Curdt et al. 2006) est un *framework* expérimental qui permet d'ajouter des continuations au langage Java. Ces continuations sont relatives à un objet implémentant l'interface `Runnable`. La création et la manipulation des continuations se fait au travers de la classe `Continuation`.

5.2.1. *Instrumentation des classes*

Ce *framework* procède par modification (instrumentation) de classes déjà compilées, soit comme post-traitement de la compilation, en produisant de nouveaux fichiers *.class*, soit au moment du chargement des classes dans la machine virtuelle.

Cette dernière technique est toutefois délicate à mettre en œuvre car elle implique d'utiliser un *class loader* spécifique, ce qui pose des problèmes avec Jade étant donnée la façon dont sont instanciés les agents. Sans entrer dans le détail, cela obligerait à définir ce *class loader* comme *class loader* par défaut d'un certain nombre de classes de Jade qui, du reste, ne devraient pas être instrumentées.

NOTE – En théorie toute classe devrait pouvoir être instrumentée sans que son comportement initial ne soit modifié, toutefois ce *framework* étant expérimental, une instrumentation inconditionnelle de toutes les classes est déconseillée. Il est recommandé de n'instrumenter que les classes susceptibles d'être dans la pile d'appel locale d'un `Runnable` sujet à continuations (voir ci-dessous).

5.2.2. *Contrainte technique de Javaflow sur l'instrumentation des classes*

Lorsqu'une classe implémentant l'interface `Runnable` est instrumentée, elle devient « continuable », c'est-à-dire apte à être l'objet de captures de continuations. Lorsqu'une classe quelconque est instrumentée, l'appel à l'une de ses méthodes peut être inclus dans la pile d'appel en aval de l'appel à la méthode `run()` d'une classe « continuable », sans affecter cette propriété d'être « continuable », comme le montre la figure 8, où est délimité un segment contigu dans la pile d'appels de méthodes appartenant à des classes instrumentées par Javaflow.

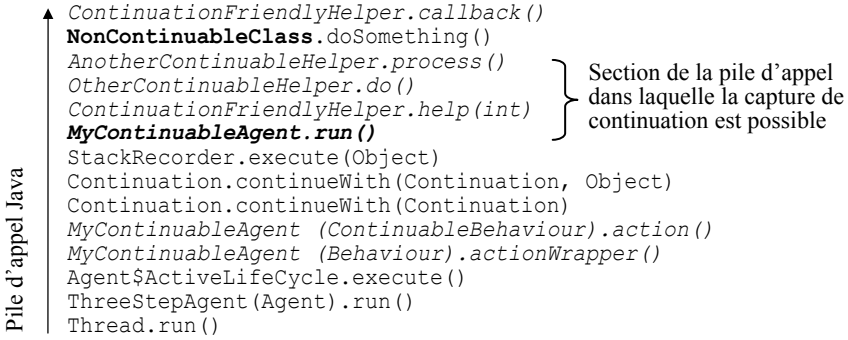

Figure 8. *Caractère continuables des classes dans la pile d'appel*

Si toutefois une classe non instrumentée s'insère dans cette pile d'appel, comme `NonContinuableCass` sur la figure, toute capture de continuation dans les appels imbriqués, à savoir le haut de la pile à partir de ce point, lèvera une exception Javaflow. Tant que la méthode n'appelle pas directement ou indirectement une méthode de type `Continuation.suspend()` provoquant la capture du contexte d'exécution,

aucune exception n'est levée, et la pile redevient « continuable » lorsque la méthode retourne normalement (si tel n'était pas le cas, aucune classe de la librairie de base de Java ne pourrait être utilisée dans les méthodes continuables !).

En pratique, cela signifie qu'il faut être vigilant lorsqu'une méthode « continuable » est utilisée comme callback par une autre classe non instrumentée, comme la méthode `ContinuationFriendlyHelper.callback()` sur la figure 8 : une telle méthode ne doit pas invoquer `Continuation.suspend()`, ou bien les classes appelantes doivent être instrumentées par Javaflow.

5.3. *Cas d'une conversation bilatérale :* `ContinuableBehaviour`

Il s'agit de la classe de base que nous proposons pour l'intégration de composants comportementaux à base de continuations dans la plate-forme Jade. Cette classe abstraite met en œuvre les primitives définies en section 4.1, 4.2 et 4.3, récapitulées dans le tableau 1 ci-dessous. La méthode `reset()` permet de réinitialiser et donc de réutiliser un automate, sans que l'on soit obligé d'en instancier un nouveau.

Signature de la primitive	*Description et effet de bord*
`void yield()`	Interrompt le flot, capture la continuation

`ACLMessage receive()`
`ACLMessage receive(int...)`
`ACLMEssage receive(MessageTemplate)`
	Lit le prochain message (éventuellement contraint par un `MessageTemplate` ou une liste de performatifs). Appelle `yield()` si aucun n'est disponible.

`void reset()` `void resetFromRun()`	réinitialise l'automate. `resetFromRun()` peut être appelée à partir de la méthode `run()`, contrairement à `reset()`

Tableau 1. *Primitives fournies par* `ContinuableBehaviour`

D'autre part, cette classe implémente les méthodes `action()` et `done()` de la classe `Behaviour` de Jade. La figure 9 donne l'implémentation de ces deux méthodes, ainsi que de la primitive `yield()`.

```
public void action() {
   if(current != null)
      current = Continuation.continueWith(current);
   if(toReset)
      reset();
}

public boolean done() { return current == null; }

protected void yield() {
   block();                        // mark behavior as blocked
   Continuation.suspend();
}
```

Figure 9. *Méthodes* `action()`, `done()` *et* `yield()` *de* `ContinuableBehaviour`

On notera que la principale différence avec l'implémentation du générateur Java donnée en section 2.2 (outre la gestion retardée du `reset()`) est l'appel à la méthode `Behaviour.block()`, qui a pour effet de mettre le comportement en état d'attente du point de vue de l'agent. Ce comportement ne sera réactivé par l'agent qu'à l'arrivée d'un nouvel événement potentiellement stimulant. Si tous les comportements sont en état d'attente, alors l'agent lui-même se met en attente.

NOTE – Comme le lecteur l'aura constaté, les exemples de composant comportementaux à base de continuation donnés en section 4.5 sont hérités de cette classe.

5.4. *Cas d'une conversation multi bilatérale :* `MultiContinuableBehaviour`

Cette extension de `ContinuableBehaviour` ajoute entre autres les deux primitives de la section 4.4, récapitulées dans le tableau 2 ci-dessous. Ces primitives impliquent un nouveau mode possible de fonctionnement, le mode parallèle, en plus du mode standard de fonctionnement que nous qualifierons de mode synchrone.

Signature de la méthode ou primitive	*Description et effet de bord*
`void parallelize()`	Passe en mode parallèle, duplique l'automate, un pour chaque participant
`void join()`	Passe en mode synchrone (voir 4.4)
`void receiveAll()`	En mode synchrone, permet d'attendre que tout le monde ait répondu

Tableau 2. *Primitives ajoutées par* `MultiContinuableBehaviour`

`MultiContinuableBehaviour` est conçu pour faciliter la gestion et la programmation de conversations multi bilatérales, mettant en jeu un initiateur, unique, et un ensemble de *n* participants. Afin d'optimiser l'ordonnancement, et du fait des limitations de la classe d'agent évoquées en section 5.1.1, une présélection des messages est effectuée sur la base de l'envoyeur du message en mode parallèle. La sémantique des primitives `receive(..)` en est modifiée, et selon le mode ces dernières vont lire la file d'attente globale de message de l'agent (en mode synchrone), ou la file d'attente associée à un participant donné (en mode parallèle).

Le principe de cette exécution pseudo parallèle des branches de conversation associées à chaque participant (en mode parallèle), est de mémoriser une continuation différente pour chaque participant, de sorte que chaque branche possède son propre sous automate autonome. Lors d'une synchronisation (appel à `join()`), la dernière continuations ayant atteint le point de synchronisation est réaffectée à la continuation globale, et le composant comportemental repasse en mode synchrone. Il est important de comprendre que la valeur des variables locales modifiées en mode parallèle est celle du dernier participant, et n'est pas nécessairement significatives globalement. C'est pourquoi une bonne pratique consiste à délimiter une section de code parallèle, entre un `parallelize()` et un `join()`, à l'aide d'un bloc Java.

6. Tests et expérimentation

6.1. *Configurations testées*

Dans le cadre de cette étude nous avons réalisé un *framework* expérimental basé sur Javaflow, implémentant les primitives décrites en section 4, sans utiliser Jade ; puis nous avons adapté les classes de test ainsi définies au *plug-in* Jade décrit en section 5. L'objectif du premier test était de mesurer précisément, sans le biais de l'infrastructure de communication de Jade, le gain en performance d'automates à base de continuations non préemptifs en comparaison d'automates équivalents préemptifs à base de *threads* classiques. À titre d'information nous avons également effectué les mêmes mesures avec la version Jade.

Dans la version non préemptive, le système entier est exécuté dans un seul *thread*. Dans la version préemptive, chaque composant comportemental de chaque agent a son propre *thread* d'exécution. Dans la version Jade, le modèle est imposé et chaque agent a son propre *thread*.

Dans le but de représenter plusieurs formes de parallélisme liées à la gestion de conversation, nous avons défini quatre composants comportementaux correspondant respectivement aux rôles d'initiateur et de participant d'un protocole nommé *Handshake*, un échange linéaire de messages *inform* avec une boucle simple, et aux rôles d'initiateur et de participant d'un protocole d'enchère comparable à *FIPA-English-Auction*. Un agent initiateur devra ici gérer ces deux comportements en parallèle, et, pour chaque comportement, les différents participants en parallèle.

6.2. *Protocoles d'interaction utilisés*

La figure 10 détaille la méthode `run()` du rôle d'initiateur du protocole d'enchère. Nous pouvons observer que la presque totalité de la gestion de ce protocole complexe tient ici en quelques lignes particulièrement intuitives, ce qui montre l'élégance de l'approche. Un comportement équivalent programmé sous forme d'automate classique nécessiterait de nombreux états et conditions, et résulterait en un code particulièrement fragmenté dans diverses classes, objets et méthodes, difficile à lire, de ce fait peu évolutif, et difficile à tester.

NOTE – Les variables `bestOffer`, `haveProposed` et `winner` sont ici des attributs : nous sommes contraints de procéder ainsi pour communiquer entre les différents sous automates, du fait que Javaflow n'autorise pas le passage de paramètres (de la même manière que nous utilisions en 2.2 un attribut pour stocker temporairement la valeur de retour du générateur).

Les rôles de participants à l'enchère, symétriques des rôles d'initiateurs, sont bien entendu plus simples, car ils ne comportent pas de parallélisme interne, et n'utilisent donc pas les primitives `parallelize()` et `join()`.

```
public void run() {
   send("auction start", INFORM); // envoi au groupe
   bestOffer = min;

   do {
      send(""+bestOffer, CFP); // envoi au groupe
      haveProposed.clear();
      parallelize(); // bloc défini pour section parallèle
      {
         ACLMessage msg = receive(PROPOSE, INFORM, REFUSE);
         if(msg.getPerformative() == INFORM)
            return; // elimine ce participant
         else if(msg.getPerformative() == PROPOSE) {
            haveProposed.add(getRunningAID());
            int offer = Integer.parseInt(msg.getContent());
            if(offer > bestOffer) {
               bestOffer = offer;
               winner = getRunningAID();
            }
         }
      }
      join();
      for(AID a:haveProposed) {
         if(a.equals(winner))        // gagnant potentiel
            send(a, ""+bestOffer, ACCEPT_PROPOSAL);
         else                        // perdant..
            send(a, "", REJECT_PROPOSAL);
      }
   }                           // tant qu'une competition subsiste
   while(haveProposed.size() > 1);

   for(AID a:getParticipants())
      if(a.equals(winner))
         send(a, "bravo", REQUEST); // demande l'argent au gagnant
      else
         send(a, "done", INFORM); // notifie la fin de l'enchère
}
```

Figure 10. *Rôle d'initiateur du protocole d'enchère (`EnglishAuctionInitiator`)*

6.3. *Performances*

A l'instar des *threads* coopératifs par rapport aux *threads* préemptifs, une activation non préemptive par continuations permet de gagner en performance, en comparaison d'une activation par *threads* préemptifs.

La figure 11 donne le temps d'exécution en fonction du nombre d'agents. Les résultats mettent en évidence un gain moyen de 50% de la version non préemptive à base de continuations par rapport à la version préemptive. Ce gain s'explique principalement par l'ordonnancement plus spécialisé de la version à base de continuation, et des changements de contextes plus rapides que ceux des *threads*.

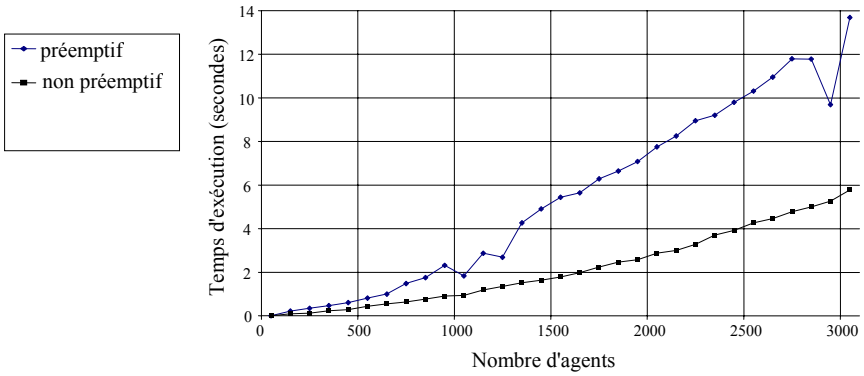

Figure 11. *Temps d'exécution en modes préemptif et non préemptif*

La figure 12 présente le temps d'exécution de la version Jade, pour information. L'augmentation significative du temps d'exécution s'explique par les temps de routage, d'encodage et de décodage de messages, de l'infrastructure de communication de Jade, comparés aux appels de méthodes directs des tests précédents. Au-delà de 1000 agents, on observe également une consommation de mémoire importante qui se traduit parfois par une saturation de la machine virtuelle si une quantité importante n'a pas été allouée dès le début. Il serait intéressant de déterminer le coût supplémentaire, en terme de mémoire, de l'utilisation des continuations, par rapport à une implémentation des mêmes comportements sous forme de machines à états finis explicite.

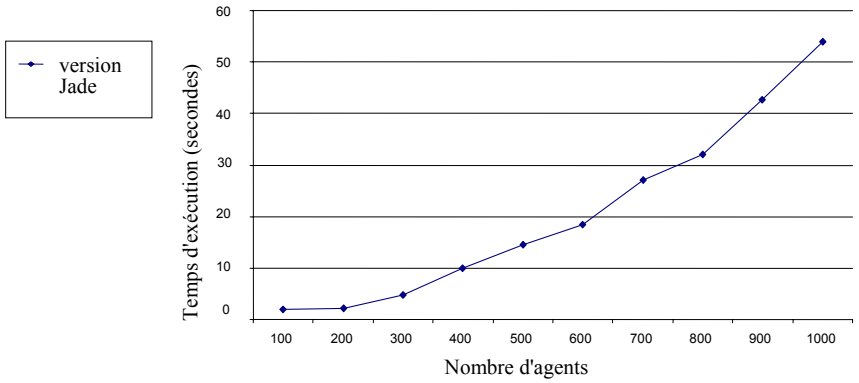

Figure 12. *Temps d'exécution de la version Jade*

7. Conclusion et perspectives

Dans cet article nous avons défini comment utiliser les continuations pour faciliter l'ingénierie d'agents conversationnels et de leurs de composants comportementaux. Cette approche permet une programmation élégante, concise et intuitive de la dynamique des comportements, sous forme d'automates, tout en améliorant significativement la lisibilité du code. Elle permet en effet d'écrire l'automate conversationnel sous-jacent comme une simple fonction, ou ensemble de fonctions ou d'objets, sans qu'il soit nécessaire de stocker et gérer explicitement son état.

Nous avons également décrit comment intégrer cette technique de programmation de comportement d'agents dans la plate-forme multi-agents compatible FIPA Jade, par le biais de deux classes abstraites, et de l'instrumentation de ces classes par le *framework* de continuations Javaflow. Nous avons présenté les contraintes de ce *framework*, ainsi que les limites inhérentes au modèle de concurrence de Jade. Le résultat de cette intégration montre que les comportements définis à l'aide de continuations sont beaucoup plus simples à programmer, et lisibles, que leurs équivalents en utilisant le style classique de programmation de comportement Jade.

Cette approche incite non seulement à profiter des structures de contrôle naturelles du langage hôte (boucles, récursivité, exceptions, etc.) pour gérer les transitions de l'automate conversationnel, mais aussi à respecter les contraintes structurantes de ce dernier à l'échelle de la durée de vie de l'automate : la localité des variables à un bloc, l'initialisation des variables avant leur utilisation, etc. Avec une approche classique, ces vérifications sur la validité des attributs dans un état donné sont laissées à la discrétion du programmeur.

7.1. *Applicabilité et limites*

Dans le cas général, cette approche permet de se dédouaner des problèmes de synchronisation de *thread*. La section 4 emploie le terme de pseudo parallélisme (entrelacement) dans la mesure où les transitions de l'automate sont en réalité exécutées séquentiellement, et non préemptives. Elle est applicable à tout type de comportements synchronisés pour lesquels la préemption n'est pas indispensable.

Toutefois, rien n'empêche de combiner cette technique avec l'utilisation de *threads* préemptifs si cela se justifie. Si par exemple un agent effectue une tâche de fond suffisamment longue, et doit rester réactif pendant ce temps à d'autres stimuli, alors l'utilisation d'un *thread* pour cette tâche de fond est un bon choix.

7.2. *Composition de comportements « continuables »*

Une des principales perspectives de ce travail est d'étudier comment des composants comportementaux continuables peuvent être composés entre eux, et dans quelle mesure et quelles situations le fait d'utiliser des continuations facilite cette composition, en comparaison d'approches classiques.

7.3. *Intégration à d'autres plates-formes de SMA*

La possibilité d'intégrer cette technique à d'autres plates-formes de SMA que Jade dépend du support des continuations dans le langage hôte de ces plates-formes. Toutefois, si elles sont supportées, cette intégration est particulièrement aisée, comme le démontre ce travail.

Références

Allison L., « Continuations Implement Generators and Streams », *The Computer Journal*, vol. 33, n°5, 1990, pp. 460-465.

Curdt T., Kawaguchi K., Cooper M., ASF Jakarta, projet Commons Javaflow, 2006 http://jakarta.apache.org/commons/sandbox/javaflow/

Bellifemine F., Poggi A., Rimassa G., « JADE – A FIPA-compliant agent framework », *International Conference and Exhibition on the Practical Application of Intelligent Agents and Multi-Agent Technology* (PAAM), Londres, avril 1999.

Bellifemine F., Poggi A., Rimassa G., « JADE, a FIPA2000 Compliant Agent Development Environment », *Intl. Conf. on Autonomous Agents*, Montréal, Canada, juin 2001, ACM.

Bellifemine F., Caire G., Trucco T., Rimassa G., *Jade Programmer's Guide*, documentation, TILab S.p.A. et CSELT S.p.A., http://jade.tilab.com/doc/index.html

Bölöni L., Marinescu D., « A Multi-Plane State Machine Agent Model », *International Conference on Autonomous Agents* (AA), Barcelone, juin 2000, ACM.

Gutknecht O, Ferber J., Michel F., « Integrating Tools and Infrastructures for Generic Multi-Agent Systems », *Intl. Conf. on Autonomous Agents*, Montréal, Canada, juin 2001, ACM.

Hewitt C., « Viewing control structures as patterns of passing messages », *Artificial Intelligence*, vol. 8, n°3, 1977, pp. 323-364.

Haynes C.T., Friedman D. P., Wand M., « Obtaining coroutines with continuations », *Computer Languages*, vol. 11, n°3, 1986, pp. 143-153.

Jennings N., Sycara K., Wooldridge M., « A Roadmap of Agent Research and Development », *Intl. Conference on Autonomous Agents*, Minneapolis, USA, mai 1998, ACM.

Jouvin D., « Utilisation des continuations pour l'ingénierie d'agents conversationnels », JFSMA 2006, Annecy, France, octobre 2006, Hermes.

Jouvin D., « Utilisation de la délégation pour l'adaptation de protocoles de conversations entre agents », JFIAD-SMA 2000, Saint-Jean-La-Vêtre, France, octobre 2000, Hermes.

Luck M., McBurney P., Preist C., *Agent Technology: Enabling Next Generation Computing, A Roadmap for Agent Based Computing*, rapport d'Agentlink II, janvier 2003.

Mertz D., « Charming Python: Iterators and simple generators », rapport interne IBM, http://www-128.ibm.com/developerworks/library/l-pycon.html, 2001

Mertz D., « Charming Python: Generator-based state machines », rapport interne IBM, http://www-128.ibm.com/developerworks/library/l-pygen.html, 2002

Nwana H., Ndumu D., Lee L., Collis J., « ZEUS: A Toolkit and Approach for Building Distributed Multi-Agent Systems », *Intl. Conference on Autonomous Agents* (AA), Seattle, WA, USA, mai 1999, ACM

Pettyjohn G, Clements J., Marshall J., Krishnamurthi S., Felleisen M., « Continuations from Generalized Stack Inspection », ACM SIGPLAN *International Conference on Functional Programming* (ICFP), septembre 2005, Tallinn, Estonie, ACM.

Poslad S., Buckle P., Hadingham R., « The FIPA-OS Agent Platform Open Source for Open Standards », *International Conference and Exhibition on the Practical Application of Intelligent Agents and Multi-Agent Technology* (PAAM), Manchester, avril 2000.

Strachey C., Wadsworth C., « Continuations: A mathematical semantics for handling full jumps », *Programming Research Group Technical Monograph PRG-11*, Oxford, 1974. (réédité dans *Higher-Order and Symbolic Computation*, vol. 13, 2000, pp. 135-152).

Tate B., « Crossing borders: Continuations, Web development, and Java programming, A stateful model for programmers, a stateless experience for users », rapport interne IBM, http://www-128.ibm.com/developerworks/java/library/j-cb03216/