Utilisation des continuations pour l'ingénierie d'agents conversationnels

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, Javascript, 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 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 allowing to explicitly capture and resume the current program state. They are present 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. The proposed approach especially facilitates modular interaction protocol implementation, 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 continuation.*

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

1. Introduction

Les systèmes multi-agents (SMA) sont un modèle de programmation particulièrement adapté à modéliser des systèmes complexes. Dans un SMA, les agents interagissent selon des modèles d'interactions plus ou moins élaborés, normalisés. Le modèle d'interaction utilisé conditionnera de nombreuses propriétés particulièrement recherchées des SMA, telles que l'autonomie des agents, l'interopérabilité, la robustesse, et l'auto-organisation (Jennings et al. 1998) (Luck et al. 2003).

Cependant, il est aussi la cause de difficultés importantes dans la mise en œuvre des SMA (Jouvin, 2000), et en particulier pour les SMA conversationnels. Pour palier ces difficultés, de nombreuses plates-formes multi-agents proposent des approches componentielles, afin de maximiser la réutilisation. Les agents sont alors dotés de composants comportementaux (Guillemet et al. 1999), définissant partiellement leur comportement par rapport à un type de conversation donné.

Cet article a pour objectif de montrer comment un concept de programmation, les continuations, peut faciliter considérablement l'écriture de tels composants, tout en apportant de la souplesse dans leur mise en œuvre, et de meilleures performances. Pour cela, nous introduirons en section 2 les continuations, leurs applications et variantes, puis nous identifierons en section 3 les difficultés de mise en œuvre des SMA conversationnels, et les réponses classiques des plates-formes de SMA. Nous présenterons en section 4 notre approche par continuations, et en section 5 les résultats d'un *framework* expérimental, avant de conclure.

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 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 « le reste du programme », ou encore « `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 la pile et les variables locales sont mémorisées.

Ce concept peut être décliné en différentes variantes ou restrictions, comme les co-routines ou les générateurs (voir section 2.2) ; mais les continuations restent la forme la plus générale, comme le montrent (Haynes et al. 1986) et (Allison, 1988).

REMARQUE – Les continuations ont un rapport certain avec les notions de *thread* et de clôture (en anglais *closure*) : de même qu'un *thread*, une continuation nécessite de mémoriser toute ou partie de la pile, et la position dans le programme ; de même qu'une clôture, elle mémorise aussi le contexte lexical (bien que cette dernière propriété dépende de l'implémentation des clôtures dans le langage hôte).

2.1. *Implémentations et exemples de continuations*

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. La figure 1 présente un exemple de générateur, avec à gauche sa définition – il s'agit d'une fonction contenant une ou plusieurs instructions `yield` – et à droite l'affichage des valeurs retournées itérativement par ce générateur, avec en italique la sortie console du programme.

```
def simple_generator(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"
```

```
for s in simple_generator(4):
    print s

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

Figure 1. *Exemple de générateur Python*

Comme le suggère cet exemple, 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, et provoque la mémorisation de l'état dans lequel se trouve le générateur, de sorte que la prochaine invocation reprendra à partir de cet état. Python supportant les clôtures, ce générateur pourrait aussi référencer une variable du contexte englobant.

L'intérêt d'un générateur est que l'état de l'itérateur n'a pas à être géré explicitement par le programmeur, ce qui simplifie l'écriture : il est alors possible d'utiliser les structures de contrôle naturelles du langage pour gérer les transitions entre états de l'automate implicite sous-jacent (Mertz, 2002). L'exemple de la figure 1 est simple et pourrait être écrit sans `yield` tout en restant lisible, cependant cela nécessiterait un attribut d'objet et plusieurs conditions supplémentaires.

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. Dans les langages fonctionnels, et en Ruby, en revanche, la continuation, « globale » au programme, est créée implicitement par une primitive qui appelle une procédure ou expression lambda et lui passe la continuation en paramètre.

```
callcc {|cont|
  for i in 0..4
    print "\n#{i}: "
    for j in i*5...(i+1)*5
      cont.call() if j == 17
      printf "%3d", j
    end
  end
}
print "\n"
```

```
0:   0  1  2  3  4
1:   5  6  7  8  9
2:  10 11 12 13 14
3:  15 16
```

Figure 2. *Exemple de continuation en Ruby*

La figure 2 présente un programme utilisant une continuation en Ruby, avec à droite la sortie console de ce programme. La primitive `callcc` (pour *call with current continuation*) appelle la clôture anonyme définie par le bloc entre accolades, et lui passe la continuation courante en paramètre. L'activation de la continuation, `cont.call()`, a pour effet de sortir prématurément des boucles `for`, en renvoyant le flot de contrôle juste après le bloc, là où elle avait été capturée.

NOTE. — Ce programme peut se traduire directement en Scheme, à l'aide de la primitive Scheme `call-with-current-continuation` au lieu de `callcc`.

D'autres langages répandus supportent nativement les continuations dans leurs versions les plus complètes. Citons par exemple Smalltalk, ML, ou Haskel. Elles sont également incluses dans Perl 6, et étaient annoncées dans une implémentation alternative du langage Python, appelée Stackless Python. Assez récemment, les continuations ont été ajoutées, par extensions ou 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 :

– La 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 indirectement 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 ;

– Javaflow[3], une librairie expérimentale permettant les continuations en Java, par modification dynamique de classe, que nous utiliserons par la suite et en section 5.

[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> {

   private transient T result;
   private Continuation current = Continuation.startWith(this);

   public T next() {
      if(!hasNext())
         throw new NoSuchElementException();
      T retval = result;
      current = Continuation.continueWith(current);
      return retval;
   }

   protected void yield(T param) {
      result = param;
      Continuation.suspend();
   }

   public boolean hasNext() { return current != null; }
   public Iterator<T> iterator() { return this; }
   public void remove() { throw new UnsupportedOperationException(); }
}
```

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

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`. A 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.

```
public class SimpleGenerator
      extends Generator<String> {

   private final int max;

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

   public static void main(String... a) {
      for(String s:new SimpleGenerator(5))
         System.out.println(s);
   }
```

```
   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 simple de générateur en Java, instrumenté par Javaflow*

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.

2.2. *Variantes et applications courantes*

Les possibilités offertes par les continuations et clôtures sont nombreuses : il est possible de réécrire les structures de contrôle classiques des langages, ou d'en concevoir d'autres, par exemple simuler un `goto` en Scheme. Cependant leur intérêt apparaît dans principalement trois types d'applications :

– L'implémentation et la composition d'automates (par exemple pour programmer des analyseurs syntaxiques), que nous développerons en section 4.1 ;

– Les co-routines et les *threads* coopératifs. Une co-routine est une fonction qui mémorise son état courant lorsqu'elle appelle une autre co-routine. Contrairement au générateur, 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 co-routines permettent d'implémenter des *threads* coopératifs, avec changement explicite de contexte d'exécution, à la différence de vrais *threads*. L'avantage se mesure surtout en termes de performance et d'encombrement : les *threads* coopératifs sont très économes, et leur changement de contexte très rapides, en comparaison de vrais *threads*. Cette technique n'est bien sur pas adaptée lorsque la préemption est nécessaire, mais elle est adaptée aux modèles concurrents non préemptifs, et à la programmation évènementielle ;

– 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 script 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 – Ce comportement peut être obtenu avec des *thread*, 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.

3. Ingénierie d'agents conversationnels : difficultés et approches classiques

Nous définissons ici par agent conversationnel, tout agent dont le comportement nécessite de mémoriser le contexte local de l'interaction en cours, que nous désignerons par conversation. Ce contexte peut prendre des formes diverses. Pour cette étude nous nous limiterons aux agents communiquant par messages asynchrones, pour lesquels une conversation sera constituée d'échange de messages inter relatés, entre plusieurs participants, dans le cadre d'une tâche collective donnée.

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

3.1. *Définition du problème*

L'expérience montre que, bien qu'apportant de nombreuses propriétés au système qu'ils composent, les agents conversationnels sont difficiles à mettre en oeuvre. Nous pouvons identifier les difficultés suivantes, liées au model d'interaction :

– Lors d'une conversation multi parties (i.e. avec plus de deux participants), il est nécessaire de combiner plusieurs comportements simultanés, synchronisés, correspondant aux interactions avec les différents agents participants à la conversation ;

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

– 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 difficile à modulariser et à réutiliser ;

– Les langages généralement utilisés pour la programmation de SMA ne possèdent pas les primitives et concepts adéquats ;

NOTE. — Des langages concurrents, comme Erlang, répondent à une partie de ces manques, mais souffrent d'autres carences, et ne sont pas assez répandus et utilisés pour avoir été choisis dans les plates-formes et *frameworks* multi-agents.

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, ce qui confirme qu'il s'agit bien là d'une des difficultés majeures dans la mise en œuvre de SMA conversationnels. Le problème devient alors : comment définir les comportements de façon modulaire, c'est-à-dire découplés 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 plateforme 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 plateforme (*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 Jade et FIPA-OS. Deux options sont possibles :

– 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 option n'est pas toujours praticable car le nombre de *thread* disponibles sur un système est limité, et trop de *thread* peuvent induire une perte de performance due à un ordonnancement excessif par rapport aux traitements effectués ;

– Soit l'état conversationnel de chaque composant comportemental est mémorisé explicitement : 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 représentant les états. Il s'agit de la technique la plus largement utilisée.

3.2. *Approches componentielles par automates*

Dans la plateforme multi-agents Jade (Bellifemine et al., 1999), chaque agent dispose d'un *thread* d'exécution. Les composants comportementaux sont dénommés *behaviors*, et se composent par héritage de sous-classes spécifiques de la classe abstraite `Behavior`. La figure 5 en donne un exemple, 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`, appelant les méthodes `op1()`, `op2()` et `op3()`. Ici la classe `SimpleBehavior` contribue très peu au comportement de l'automate, mais d'autres *behaviors*, tels que `ContractNetInitiator`, définissent de véritables implémentations abstraites de protocoles, liées au code spécifique de l'agent par héritage ou par *callback*. Certains composites ont en outre pour fonction de synchroniser d'autres composants comportementaux (par exemple `SequenceBehavior`, `ParallelBehavior`, etc.).

```
class my3StepBehaviour extends SimpleBehaviour {
   private final int FIRST = 1, SECOND = 2, THIRD = 3;
   private int state = FIRST, numberOfExecutions = 3;

   public void action() {
      switch (state) {
      case FIRST: {op1(); state = SECOND; break;}
      case SECOND:{op2(); state = THIRD;  break;}
      case THIRD: {op3(); state = FIRST;
         numberOfExecutions--; break;}
      }
   }

   public boolean done(){
      return (!(numberOfExecutions>0));
   }

   private void op1(){ System.out.println("Step 1"); }
   private void op2(){ System.out.println("Step 2"); }
   private void op3(){ System.out.println("Step 3"); }
}
```

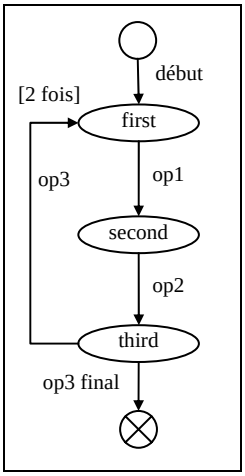

Figure 5. *Exemple de composant comportemental (*Behavior) de la plateforme *Jade*

Des techniques comparables sont présentes dans la plupart des plates-formes de SMA. Nous pouvons citer notamment les automates multi plans de Bond (Bölöni et al., 2000), les automates de Zeus (Nwana et al., 2000), les *Task* de FIPA-OS (Poslad et al., 2000), ou encore les automates conçus avec SEdit dans MadKIT.

4. Approche par continuations

Comme nous l'avons vu, 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*. D'autre part, la possibilité d'intégrer notre technique à des plates-formes existantes, nous incite également à oeuvrer dans cette voie. Toutefois, il serait également souhaitable de pouvoir retrouver l'aisance d'écriture du flot des transitions de l'automate avec les structures de contrôle naturelles du langage, comme c'est le cas avec la première option mentionnée en fin de section 3.1, tout en s'abstrayant des difficultés de synchronisation inhérentes à la programmation multi-*thread* (protection de ressources critiques, asynchronie, etc.).

Par rapport à ces deux points, 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 l'état d'exécution d'une méthode, 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 manipuler explicitement son état ;

– de ne pas être sujets aux problèmes de synchronisation entre *threads*, puisque ce modèle est non préemptif et les transitions exécutées séquentiellement.

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 Javascript ou Java, la continuation sera un objet particulier. Conceptuellement, cet objet est immutable et 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 encapsulant une continuation, similaire à l'implémentation du générateur vu en 2.1. La classe abstraite définie figure 6 comprend le minimum requis pour gérer un automate simple :

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

– le *point d'activation* de l'automate. Pour des raisons de clarté nous désignerons ce point d'activation par la méthode `activate()` ;

```
public abstract class Automaton implements Runnable {
   private Continuation current = Continuation.startSuspendedWith(this);

   protected void yield() { Continuation.suspend(); }

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

Figure 6. *Classe abstraite d'un automate simple (en Java) à base de continuations*

– une *méthode de capture* de l'état courant, `yield()`, qui doit interrompre le flot normal d'exécution du programme, et retourner le flot de contrôle à l'appelant de la méthode ou fonction d'activation de l'automate. En pratique cette primitive pourra être incluse dans les méthodes de lecture de message ;

A chaque exécution de l'automate depuis son point d'activation, 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 continuation correspondant au nouvel état capturé, puis retourne normalement.

Pour être utilisable, un composant comportemental abstrait doit également comprendre un moyen de consommer des messages, ou des évènements comme la survenue d'un *timeout*, lors de l'avancement de l'automate. Dans le cas d'agents conversationnels, chaque agent est en principe doté d'une file d'attente de messages. Une file d'attente au niveau du composant comportemental peut aussi être envisagée, associée à un mécanisme de présélection des messages. Cette présélection sera généralement basée sur un identifiant de la conversation ou de l'échange de message, par exemple l'attribut de message FIPA `conversation-id`.

4.2. *Pseudo parallélisme et primitives de synchronisation*

Pour palier 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, etc.). Ce cas implique un parallélisme inhérent à la gestion des interactions avec *n* participants, au niveau du rôle de l'initiateur, même si le comportement associé à chaque participant est le même. Afin de faciliter la programmation de tels comportements, nous proposons d'introduire les deux primitives suivantes :

– `parallelize()` : une primitive permettant de passer en mode parallèle, en dupliquant l'automate courant en *n* sous-automates, un pour chaque participant ;

– `join()` : la primitive inverse, valide uniquement en mode parallèle, permettant d'attendre (notons qu'il ne s'agit pas d'une attente sur un *thread*) que tous les sous-automates atteignent ce point, pour ensuite repasser en mode séquentiel standard.

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

Figure 7. *Comportement à 3 états de la section 3.2 : approche par continuations*

4.3. *Exemple et comparaison*

La figure 7 reprend l'exemple trivial de la section 3.2, figure 5, décrivant un comportement en boucle sur 3 états. En comparaison du composant Jade équivalent, la première observation est qu'aucun attribut de classe 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.

REMARQUE – 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. Framework expérimental

Dans le cadre de cette étude nous avons réalisé un *framework* expérimental basé sur Javaflow. Les classes abstraites `BilateralRole` et `MultiBilateralRole` implémentent les primitives décrites section 4.2. Par manque de place, cet article n'inclut pas le code Java correspondant ; le tableau 1 ci-dessous synthétise les méthodes importantes de ces classes.

Signature de la méthode ou primitive	*Description et effet de bord*
`void activate()`	Point d'activation de l'automate
`void yield()`	Interruption du flot, capture de la continuation
`Message receive()` `Message receive(Message.Type... types)`	Lecture du prochain message (éventuellement typé). Appelle `yield()` si aucun disponible.
`void parallelize()`	Passe en mode parallèle, duplique l'automate
`void join()`	Passe en mode synchrone (voir 4.2)

Tableau 1. *Primitives fournies par* `BilateralRole` *et* `MultiBilateralRole`

5.1. *Expérimentation sur un protocole d'enchère*

Afin de représenter deux 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* (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 gérer ces deux comportements en parallèle, et, pour chaque comportement, les différents participants en parallèle.

La figure 8 détaille la méthode `run()` du rôle d'initiateur de l'enchère. La presque totalité de la gestion de ce protocole tient ici en quelques lignes particulièrement

intuitives, ce qui montre l'élégance de l'approche. Un comportement équivalent programmé sous forme d'automate explicite nécessite de nombreux états et conditions, et résulte un code fragmenté, difficile à lire, et à valider.

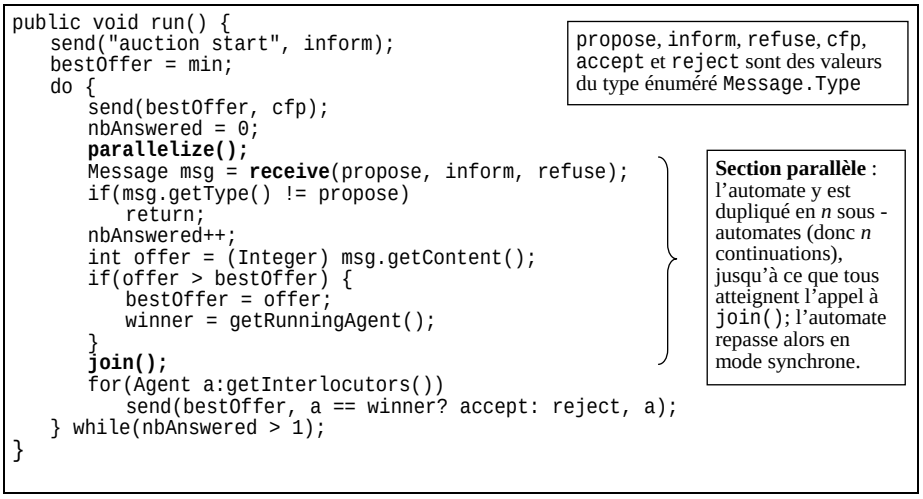

Figure 8. *Rôle d'initiateur du protocole d'enchère (`EnglishAuctionInitiator`)*

NOTE – Les variables `bestOffer`, `nbAnswered` 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 2.1 un attribut pour stocker temporairement la valeur de retour du générateur). Alternativement, un langage supportant les clôtures permettrait d'utiliser pour cela des variables du contexte englobant.

Les rôles de participants, 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()`.

5.2. *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. Pour évaluer ce gain, nous avons également défini une version préemptive de la plateforme, avec une file d'attente bloquante de messages et un *thread* pour chaque composant comportemental. La figure 9 donne le temps d'exécution en fonction du nombre d'agents. Dans la configuration du test, un agent initiateur est à la fois initiateur de l'enchère et du *Handshake*, et chaque participant cumule les rôles de `HandshakeParticipant` et d'acheteur, ce qui permet de représenter les deux formes de parallélisme évoquées en section 3.1. Les résultats mettent en évidence un gain moyen de 50%.

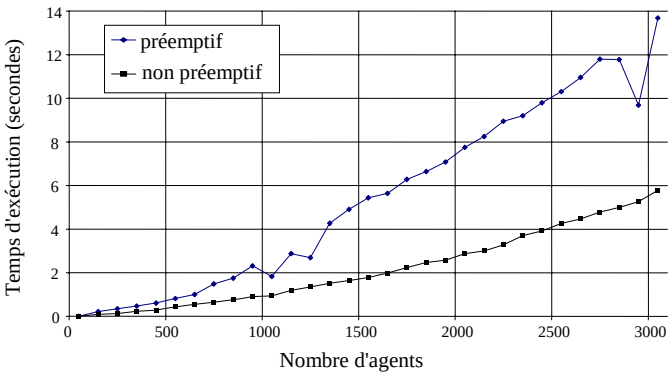

Figure 9. *Temps d'exécution en modes préemptif et non préemptif*

6. 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 de ces comportements, sous forme d'automates, tout en améliorant significativement la lisibilité du code. Elle permet en effet d'écrire l'automate sous-jacent comme une simple fonction, sans qu'il ne soit nécessaire de stocker et gérer explicitement son état. Ainsi, elle incite non seulement à profiter des structure de contrôle naturelles du langage hôte (boucles, récursivité, exceptions, etc.), mais aussi à respecter les contraintes structurantes et validations 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 sont laissées à la discrétion du programmeur.

La possibilité d'intégrer cette technique dans les plates-formes de SMA existantes dépend du support des continuations dans le langage hôte de ces plateformes, toutefois, si elles sont supportées, cette intégration est aisée. Elle laisse au concepteur le choix du modèle d'activation des agents, adaptable selon le type de SMA, ce qui permet un gain de performance, et une économie de *threads*.

Dans le cas général, cette approche permet de se dédouaner des problèmes de synchronisation de *thread*. La section 4.2 parle en effet de pseudo-parallélisme (entrelacement) dans la mesure où les transitions de l'automate sont en réalité exécutées séquentiellement, et non préemptives. Toutefois, rien n'empêche de combiner cette technique avec l'utilisation de *threads* préemptifs si cela se justifie.

Une des principales perspectives de ce travail, en cours de réalisation, est de l'intégrer à la plateforme Jade, sous la forme d'un `ContinuationBehavior`, et à d'autres plates-formes telles que FIPA-OS ou MadKIT. Si l'engouement pour les continuations dans la communauté des SMA suit celui observé pour le développement d'applications Web, les continuations y auront un avenir certain.

Références

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

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.

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

Guillemet A., Haïk G., Meurisse T., Briot J.P., Lhuillier M., « Mise en œuvre d'une approche componentielle pour la conception d'agents », JFIADSMA, Novembre 1999, Hermes.

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 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/