20 octobre 2006

# Utilisation des continuations pour l'ingénierie d'agents conversationnels

Denis Jouvin

LIRIS, university Claude Bernard Lyon 1

http://liris.cnrs.fr/denis.jouvin/
denis.jouvin@liris.cnrs.fr

# Introduction

- Approche orientée agents, systèmes multi-agents (SMA)
  - Adapté à la modélisation de systèmes complexes
    - propriétés intéressantes: autonomie, robustesse, distribution naturelle, interopérabilité, auto-organisation, etc.
  - …conditionnés par des modèles d'interaction particulièrement élaborés
    - Gestion des conversations = principale source de difficultés d'implémentation des SMA (qualifiés ici de « conversationnels »)
- Plates-formes multi-agents
  - Approches componentielles pour faciliter la gestion des conversations
    - Composants comportementaux réutilisables, prenant en charge totalement ou partiellement la gestion des conversations / des protocoles d'interaction
    - ces composants restent difficiles à écrire et à combiner !
- Approche à base de continuations
  - Implémentation concise et élégante de composants comportementaux
    - améliore lisibilité, cohérence, performances, facilite la synchronisation

# Introduction aux continuations

- Origines des continuations
  - Concept de programmation assez ancien
    - initialement présent dans les langages fonctionnels (exemple: Scheme)
    - … et dans des modèles concurrents comme le modèle d'acteurs
- Principe
  - Permet de stocker et d'activer un contexte d'exécution, c.à.d.

    "*Capturer l'état d'exécution d'un programme dans une variable, la continuation, manipulable programmatiquement, pour pouvoir réactiver ce programme par la suite, à partir de cet état*"
  - A rapprocher des *threads* et des clôtures (*closures*)
- Définitions informelles
  - « *Le reste du programme* » : suppose une exécution linéaire de ce dernier
  - « *Un goto avec paramètres* » : ne tient pas compte des variables locales

# Introduction aux continuations

## Variantes et *frameworks*

- Variantes et restrictions
  - Coroutines : fonctions « résumables » stockant leur propre état d'exécution
  - Générateurs : fonctions « résumables » renvoyant successivement plusieurs valeurs, tel un itérateur, et stockant leur état entre deux appel
  - Les continuations sont considérées comme la forme la plus générale
    - permettent de programmer simplement coroutines et générateurs
- Langages intégrant nativement les continuations / variantes
  - Scheme, Ruby, Python, Smalltalk, Haskel, Perl 6, certains Lisps, etc.
- *Frameworks* permettant les continuations
  - Cocoon: ajoute les continuations au moteur Javascript Rhino
  - RIFE: permet des continuations limitées en Java
  - Javaflow: ajoute les continuations à Java

# Introduction aux continuations

## Exemples (langages intégrant nativement les continuations)

- Les générateurs de Python
  - **yield** permet de renvoyer plusieurs valeurs successives
  - inutile de gérer l'état de l'itérateur dans des attributs
  - réutilise les structures de control natives du langage pour gérer les transitions

```
def my_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"
```

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

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

- Le *call/c.c.* de Ruby
  - **callcc** passe la continuation courante à la fermeture {..}
  - Similaire aux continuations Scheme (call-with-current-continuation)
  - la continuation est ici globale

```
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
```

# Introduction aux continuations

## Principales applications

- Applications historiques
  - Implémentation et composition d'automates
    - Exemples: analyseurs syntaxiques, compilateurs, validateurs
  - *Threads* coopératifs et coroutines
    - Changements de contextes explicites mais beaucoup plus rapides
  - Implémentation de modèles concurrents de type acteurs
- Application plus récente
  - programmation d'applications web
    - Interaction avec le navigateur = conversation à états
    - Permet de s'abstraire de la gestion explicite de l'état conversationnel
    - (avoir un *thread* dédié à chaque état conversationnel n'est pas faisable)
  - Regain d'intérêt certain pour les continuations (ou ce qui y ressemble)

# Ingénierie de composants comportementaux

- Principales difficultés de l'ingénierie de SMA conversationnels
  - Gestion de plusieurs conversation en parallèle
  - Parallélisme de conversations multi parties
    - Gestion d'un comportement en parallèle pour chaque conversation bilatérale avec chaque participant
      - peuvent nécessiter des points de synchronisation
  - Parallélisme intrinsèque à un protocole donné
    - Même un protocole bilatéral peut comprendre des branches parallèles ou entrelacées (i.e. ordre partiel de messages)
  - Gestion d'erreurs dans les protocoles d'interaction
    - Totalement asynchrone (time-out, états mixtes, etc…)
- Langages concurrents (ex: Erlang)
  - Soufrent d'autres limitations
  - Pas assez répandus pour être utilisés dans les plates-formes de SMA

# Ingénierie de composants comportementaux

## Approches componentielles des plateformes de SMA

- Bibliothèque de composants comportementaux « abstraits »
  - Le problème devient: réutilisation de ces comportements abstraits
    - Composition et liaison avec le code de l'agent et d'autres composants
    - Synchronisation avec d'autres composants comportementaux correspondant à d'autres conversations / branches de conversation
- Synchronisation inter *thread* souvent nécessaire
  - Source d'erreur et difficulté de mise en œuvre / debugging
- Différentes techniques de composition
  - Callbacks / interfaces de callback
  - Héritage et polymorphisme / classes abstraites
  - Modèle d'évènement associé au modèle de composant (ex: JavaBean)

# Ingénierie de composants comportementaux

## Stratégies de gestion du parallélisme et de la synchronisation

- États conversationnels: 2 stratégies
  - Un *thread* par composant / branche + méthodes bloquantes
    - Facile à écrire (pas besoin de gérer l'état conversationnel explicitement)
    - Parfois difficile à synchroniser (sémaphores, files d'attentes, …)
    - Nécessite de nombreux *threads* → problèmes de performance
  - Composant sous forme d'automate (activation pas à pas)
    - Gestion explicite de l'état conversationnel (spécifique à chaque plateforme)
    - Difficile à programmer, mauvaise lisibilité (code fragmenté par transitions)
- La plupart des plateformes utilisent des stratégies mixtes
  - FIPA-OS: thread pool partagé par agents/tâches + « tâches automates »
  - JADE: un thread par agent + comportement automates (*behavior*)
  - BOND: un thread par agent + automates multi-plans;  ZEUS: similaire
  - MadKIT: automates SEdit, activation/ordonnancement paramétrable

# Approche par continuations

- Les continuations permettent de
  - capturer l'état conversationnel implicitement
  - définir les comportements comme automates à base de continuation
- Conséquences / stratégie 1: non préemptif
  - Activation pas à pas de l'automate
    - Pas de problèmes de synchronisation inter *thread* (non préemptif)
    - Ordonnancement paramétrable, selon le type de SMA
      - Meilleures performances qu'ordonnancement de thread
  - Plus économique en ressources systèmes (mémoire, etc.)
- Conséquences / stratégie 2: programmation aisée
  - Libère le programmeur de la gestion explicite de cet état
  - Utilisation possible des structures de control natives du langage hôte
  - Code non fragmenté par transition, meilleure lisibilité, concis

# Approche par continuations

## Automate à base de continuations en Java (javaflow)

- Continuation = immutable
  - objet encapsulant automate nécessaire
- Automate « abstrait » minimal
  - current : continuation courante
  - méthode activate()
  - méthode yield() (primitive)
    - interrompt flot de control
    - capture contexte d'exécution
  - Automate concret
    - sous-classe implémente run()
- insuffisant pour composants comportementaux d'agent

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

    /**
     * Suspends execution and captures current context
     * Should be called from inside run()
     */
    protected void yield() {
        Continuation.suspend();
    }

    /**
     * Advance the automaton one step
     */
    public void activate() {
        if(current != null)
            current =
                Continuation.continueWith(current);
    }
}
```

# Approche par continuations

## Exemple comparatif (Jade) : comportement simple à 3 pas

```
public class My3Step extends SimpleBehaviour {
    private int state = 1;
    private int numberOfExecutions = 5;
    public void action() {
        if (state == 1) {
            System.out.println("Step 1");
            state = 2;
        }
        else if (state == 2) {
            System.out.println("Step 1");
            state = 3;
        }
        else if (state == 3) {
            System.out.println("Step 2");
            state = 1;
            numberOfExecutions--;
        }
    }
    public boolean done() {
        return numberOfExecutions == 0;
    }
}
```

```
public class My3Step extends Automaton {
    public void run() {
        for(int i=0; i<5; i++) {
            System.out.println("Step 1");
            yield();
            System.out.println("Step 2");
            yield();
            System.out.println("Step 2");
            yield();
        }
    }
}
```

# Approche par continuations

## Primitives de communication et de synchronisation

- Primitives de consommation d'événements
  - Réception de messages
    - Par exemple receive(), ou receive(<condition ou motif sur le message>)
  - Réception d'événements de *time-out* ou de synchronisation
- Proposition : primitives de (pseudo-) parallélisme et de synchronisation
  - parallelize() : passe en mode parallèle dans une conv. multi-bilatérale, duplique l'automate en autant de sous-automates que de participants
  - join() : « attend » (non bloquant) que tous les sous-automates atteignent ce point, et repasse en mode synchrone
- In fine: exécution séquentielle (non préemptive)

# Expérimentation et tests

- Prototype – environnement de test
  - *Framework* pour tester l'implémentation de protocoles d'interaction ou de comportements par continuation
    - Téléchargeable sur http://liris.cnrs.fr/denis.jouvin/continuations.html
  - Implémentation en Java (utilisant Javaflow) des primitives de communication et de synchronisation (cf. diap. 11)
    - inclue la gestion des exceptions et *timeout*
- Expérimentation sur deux protocoles multi bilatéraux
  - Handshake : succession de salutations (requête-réponse)
  - English Auction : comparable à *FIPA-Auction-English*
  - Chaque initiateur/participant comprend les 2 comportements
    - Parallélisme induit par la gestion de plusieurs conversations en parallèle
  - Nombre de participants élevé (plusieurs milliers)
  - Deux versions : avec activation préemptive et non préemptive
    - Parallélisme induit par la gestion simultanée des conv. avec les participants

# Expérimentation et tests

## Détail: implémentation de l'initiateur de English Auction

```
public void run() {
    send("auction start", inform);
    bestOffer = min;
    do {
        send(bestOffer, cfp);
        nbAnswered = 0;
        parallelize();
        Message msg = receive(propose, inform, refuse);
        if(msg.getType() != propose)
            return;
        nbAnswered++;
        int offer = (Integer) msg.getContent();
        if(offer > bestOffer) {
            bestOffer = offer;
            winner = getRunningAgent();
        }
        join();
        for(Agent a:getInterlocutors()) {
            Message resp = (a == winner)? accept: reject;
            send(bestOffer, resp, a);
        }
    } while (nbAnswered > 1);
}
```

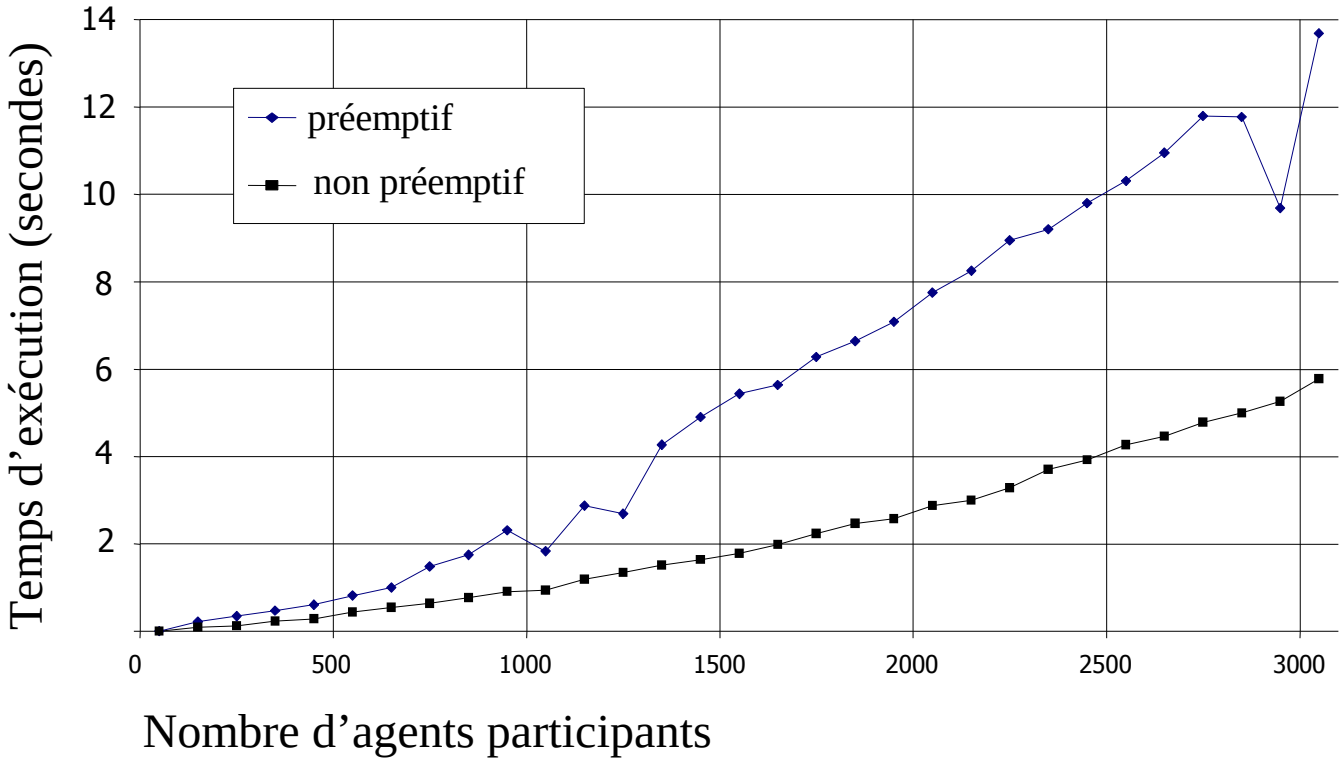
- Protocole d'interaction relativement complexe en quelques lignes !

Section parallèle : automates (continuation associée) dupliqués pour chaque participant

utilisation d'un simple while pour gérer les itérations de l'enchère

# Expérimentation et tests

## Résultats: comparaison des perf. préemptif / non préemptif




- Gain moyen de 50% en temps d'exécution de la version non préemptive à base de continuations par rapport à la version préemptive (*threads*)

# Conclusion et perspectives

- Approche par continuation pour l'ingénierie de composants comportementaux d'agents
  - Permet de définir un composant comportemental comme un automate

ET

  - Libère le programmeur de la gestion explicite de l'état conversationnel
  - Style de programmation de comportement concis et élégant
    - Améliore significativement la lisibilité du code, facilite le *debugging*
  - Utilisation des structures de control natives du langage pour le flot de control du protocole, et des vérifications du compilateur associées
    - Le code n'est pas fragmenté selon les états et transition de l'automate
      - Organisation du code selon motivations de génie logiciel (modularité)
  - Pas de problème de synchronisation inter thread
  - Activation et ordonnancement des agents/comportements configurable
    - meilleures performances que *threads* préemptifs dans certaines situations

# Conclusion et perspectives

## Applicabilité et limitations

- Gain en performances non nécessairement significatifs
  - MAIS apporte flexibilité certaine dans le mode d'activation des agents
    - Ex.: si plusieurs milliers d'agents, mode d'activation des agents = facteur clé de performances et de bonne exécution
- Utilisation de thread peut s'avérer toujours nécessaire
  - cas de tâches longues, en arrière plan, indépendantes
  - peut être combiné avec une programmation par continuation sans aucun problème
- Agents mobiles
  - Dépend de la possibilité de sérialiser les continuations du langage hôte
    - en java/javaflow, pas de problème ; en Ruby ou Scheme, problème…

# Conclusion et perspectives

## Intégration dans les plateformes de SMA existantes

- Applicable à toute plateforme SMA pouvant utiliser un langage ou *framework* supportant les continuations
- Exemple: intégration dans JADE avec Javaflow
  - Déf. d'un classe abstraite de comportement ContinuableBehavior
  - Déf. d'une famille de sous-classes de ContinuableBehavior
    - BilateralBehavior : primitives simples (receive(…), yield()…)
    - MultiBilateralRole : primitives de pseudo parallélisme (parallelize(), join()…)
    - etc.
  - Toute les classes ainsi définies doivent être instrumentées par Javaflow (soit à la compilation, soit au chargement des classes)