Continuations and behavior components engineering in multi-agent systems

Denis Jouvin

LIRIS, université Claude Bernard Lyon 1
denis.jouvin@liris.cnrs.fr

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 concurrent models such as Hewitt actor model or process calculi, and more recently in dynamic programming languages (such as Ruby, Smalltalk, Python, and even Javascript or Java). They have been applied to automaton programming, cooperative threads, compilation techniques, and have lastly raised interest in web application programming. This paper shows how this concept happens to be especially useful and elegant to program agent behaviors (or behavioral components), while increasing code readability and ease of writing. The proposed approach especially facilitates modular interaction protocol implementation, one of the main difficulties in conversational agents engineering.

Keywords. Continuations; conversational multi-agent systems; agent oriented software engineering; behavioral component; continuation-based automatons

1. Introduction

Multi-agent systems (MAS for short) are a programming paradigm especially well suited to model complex systems. In MAS, agents usually interact by means of an elaborate, normalized interaction model. The interaction model used will condition numerous prized MAS properties, such as agent autonomy, interoperability, robustness, and self-organization, as mentioned in Luck et al. [7].

However, these elaborate interaction models are also a significant source of difficulties in conversational MAS implementations, as shown in former research [6]. To address these difficulties, various multi-agent platforms propose componential approaches. Agents are then provided with reusable behavioral components, that define partially or completely their behavior with respect to a given conversation type.

In this paper we show how a well established programming concept, the continuations, can significantly facilitate the writing of such behavioral components, while bringing some flexibility and better performance in their implementation.

Section 2 introduces continuations, their applications and variants. Section 3 identifies implementation issues in conversational MAS, and the classical solutions proposed by multi-agent platforms. We present our continuation based approach in section 4, followed in section 5 by the results of an example test, and then conclude on its benefits, limitations, and integration into existing MAS platforms.

2. Introduction to continuations

Continuation are a well established programming concept, described for example by Strachey et al. [13] in the early seventies, which were originally present in functional programming languages such as Scheme or ML, in the actor model of Hewitt [4], and in various process calculi.

The principle consists in capturing the currently executing program state into a variable or artifact that can be manipulated programmatically, called the continuation, in order to be able to resume again the program from this state, by activating the continuation. More precisely, we designate by *execution context* the program state, in reference to the execution context of a thread or process, and by *continuation* the object or artifact allowing to activate (i.e. switch to) this execution context.

Some informal definitions of continuations are found in the literature, such as "the rest of the program", or "`goto` with parameters", however none of them are really satisfactory: the first implies a linear execution of the program, that should be unwind beforehand, and the second does not express that local variables and call stack are captured in a continuation.

Although there exist variants and restrictions of continuations, such as coroutines and generators (see section 2.2); continuations remain the most general form, as shown by Haynes et al. in [5] and Allison in [1].

Continuations are somehow related to threads, since they capture all, or part of, the stack, including local variables, in addition to the "program counter"; however their activation is done programmatically rather than by an operating system scheduler.

2.1. Implementations and examples

For illustrative purpose, we will borrow examples from various languages that support continuations natively, such as Scheme or Ruby, as well as a restricted form of continuation, to start with: Python's generators.

```
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
```

Fig. 1. Simple Python generator example: on the left the generator definition; on the right the code to display the successive values returned by the generator, and the console output in italic.

As figure 1 suggests, a generator behaves the same way as an iterator, written however as a function that returns successive values, using the `yield` keyword, while memorizing its current execution state between each call (see Mertz [8]).

The advantage of generators is that the programmer does not have to explicitly manage the iterator state, using object attributes. Indeed, generators make it possible to use the flow control structures of the language (if, loops, etc.) to manage transitions between the states of the underlying automaton, as shown by Mertz in [9].

In object based languages, capturing the current execution context has the side effect of interrupting the control flow, to take it back to the caller of the "continuable" function or object, as does `yield` in a generator. In a sense, the continuation is relative to this callable object. In functional language however, and in Ruby, the continuation is "global" to the program, and is created implicitly by a primitive that calls a target function or lambda expression, and pass it the continuation as parameter.

```
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
```

Fig. 2. Continuation example in Ruby, with the console output on the right (in italic)

The `callcc` primitive (call with current continuation), in figure 2, calls the anonymous closure defined by the next bloc between brackets, and passes the continuation as the parameter `cont`. The continuation activation, `cont.call()`, results in the premature exit of the two `for` loops, by taking the control flow just after this bloc.

Note that this program could be translated in Scheme very easily, by using the Scheme primitive `call-with-current-continuation`. Other programming languages support continuation natively in their most complete implementations: we can mention for example Smalltalk, Haskel, and Perl 6.

More recently, continuations have been added as libraries or extensions to common object based imperative language like Javascript or Java. Since this type of extension is related to flow control and low level stack manipulation, it requires either a modification of the interpreter, class instrumentation and manipulation techniques, or source code transformations, as shown by Pettyjohn et al. in [11]. Good examples are:

- Flowscript[1], a modified version of the Rhino Javascript interpreter implementing continuations, part of the apache Cocoon framework since version 2;
- The RIFE web application framework[2], which allows limited continuations;
- And the Javaflow framework[3], which implements continuations in Java, using a dynamic class instrumentation technique, which we will use hereafter.

[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(); }
}
```

Fig. 3. Basic implementation of Python style generators in Java, using Javaflow continuations

Figure 3 gives a simple implementation of Python style generator in Java, using Javaflow. On the contrary to Ruby or Scheme, the continuation represents here the execution state inside the `Runnable` object passed to the continuation capture routine, and not the execution state in the calling function. In this example, it is necessary to temporarily store the return values of the generator in the `result` attribute, since Javaflow does not handle continuation parameters and return values like Ruby.

```
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");
   }
}
```

Fig. 4. Simple generator example in Java, instrumented by Javaflow

Figure 4 takes back the example of figure 1, translated in Java, and using the generator class of figure 3. The console output is identical. To work properly, these classes need to be instrumented by Javaflow at compile or class loading time.

2.2. Variants and applications

The possibilities offered by continuations and closures are numerous: one can rewrite existing or custom flow control structures, for example simulating a `goto` in Scheme. However the most useful applications of continuation are:

- The implementation and composition of automatons (for example in syntactic parsers and validators), that we will further develop in section 4.1;
- Coroutines and cooperative threads. Coroutines are functions that retain their execution state when calling another coroutine. On the contrary to generators, coroutines never return after capturing its state, but explicitly call another coroutine that will in turn be activated. Coroutines thus allow defining cooperative threads, with explicit context switching. The main advantage is performance: cooperative threads are lightweight and fast compared to real preemptive threads. This technique is not suited when preemption is necessary, but is well suited to non preemptive concurrent models and event programming;
- Continuation based Web application programming, described by Tate et al. in [14], which has recently focused a lot of attention. Historically introduced by the Web application framework Seaside, this technique has been followed in other frameworks like RIFE or Cocoon. The reason is simple: the interaction between a Web server and a web browser usually takes the form of a stateful conversation, for which it is necessary to store the state on the server. The routines controlling the page flow can use continuations to store this state implicitly, relieving the programmer from this tedious and error prone task. Page flow is then described as a simple instruction flow in a script. This principle being simple and elegant, it has quickly been adopted by many Web application developers.

Remark. Such a behavior may be obtained using threads; however this is usually not feasible in practice, since it would require one thread for each possible conversational state during a session, which would result in too many threads, especially if the server implements the web browser back button behavior.

3. Conversational agent engineering

We define here a conversational agent as an agent whose behavior requires memorizing the local context of the ongoing interactions, which we refer to as *conversation*. This context may take various forms. In this work we focus on agents communicating by asynchronous messages, organized into inter-related message sequences, the conversations, between several participants, in the context of a collective task.

3.1. Problem definition

Experience shows that, though bringing many useful properties to the system that they compose, conversational agents are difficult to implement. In particular, we can identify the following difficulties related to conversation management:

- *Multi-party conversation parallelism*. In a multi-party conversation, it is necessary to combine several behaviors simultaneously, corresponding to the bilateral sub-conversations with each participant separately, and possibly synchronize them;
- *Internal parallelism*. A bilateral conversation with a single participant can itself comprise some form of parallelism: it may fork into several parallel or interlaced branches of the conversation;
- *Protocol error management*. A great part of the complexity implied by conversation management comes from the asynchronous error management: protocol errors, timeouts, etc. This aspect being transverse, it is difficult to modularize and reuse;

Remark. The programming languages used in MAS platform are not concurrent languages, and, thus, are not designed to handle elaborate parallel processing. Concurrent languages, such as Erlang for example, could address some of these issues; however they suffer from other limitations, and are unfortunately not popular enough to have been chosen in existing MAS platforms.

It is interesting to observe that most MAS platforms propose componential approaches in order to promote reusing of behavioral components, which confirms the importance of the difficulties identified above in MAS design and implementation.

The problem then becomes: how to define behavioral components in a modular way, that is to say, decoupled and encapsulated in reusable components; and how to combine and synchronize several behavioral components consistently in agents.

Considering for example FIPA interaction protocols, a FIPA compliant platform will typically propose abstract behavioral components implementing partially these protocols, and bound to the agent specific code using a platform specific composition technique (callbacks, inheritance, or event model).

Behavior component libraries help the programmer to deal with delicate aspects of conversation management, such as: timeouts, protocol error handling, following a group of participants in parallel, etc. These libraries are well developed and largely used in MAS platforms. Two strategies are possible to implement such components:

- Either a thread is assigned to each parallelizable branch of each behavioral component. A waiting state is then implemented as a blocking method on this thread, and the conversational state is defined by all the threads execution contexts. This option is not always feasible since, as we discussed it in the case of web applications, it may involve too many threads: the number of available threads is limited by the operating system, and too many threads may result in a performance loss due to an excessive scheduling with respect to the actual processing;
- Or the conversational state of the behavioral component is stored explicitly. This implies that the corresponding code be fragmented according to the transitions and structure of the underlying automaton, so that it may be executed step by step, by interrupting and resuming its execution at each state. Typically, distinct methods or objects will represent the various transitions, themselves associated to objects representing the states. This technique is the most widely used, because it gives more control on the activation of the behavior components, and on the number of threads allocated. The FIPA-OS platform, described by Poslad et al. in [12], uses a global thread pool to activate the behavioral components of agents (called *task* in FIPA-OS), whereas Jade, a MAS platform described by Bellifemine et al. [2], assigns by default one thread per agent.

3.2. Automaton behavioral components

In Jade, behavioral components are named *behaviors*, and are all inherited from the abstract `Behavior` class. Figure 5 shows a simple 3 states behavior example, illustrating the explicit state management, and resulting code fragmentation, in the `switch` statement. Although not obvious from first reading, this behavior is looping two times on the three states before termination.

```
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"); }
}
```

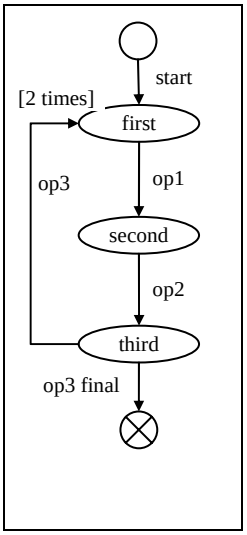

Fig. 5. Simple Jade behavior example: 3 states automaton, with the corresponding finite state automaton diagram on the right.

In this example the `SimpleBehavior` class does not contribute much to the behavior; however, other available behaviors, like `ContractNetInitiator`, or `AchieveREInitiator`, define abstract implementations of common interaction protocols, bound to the agent specific code by inheritance or callback. Some composite behaviors allow various types of composition of children behavior, for example `SequenceBehavior` or `ParallelBehavior`. Inter-thread synchronization is usually not necessary in Jade since transitions are executed sequentially.

Comparable techniques can be found in other MAS platforms. Let us mention for example the multi-plan automatons of the Bond platform, by Bölöni et al. [3]; the finite state automatons of the Zeus platform, by Nwana et al. [10]; the FIPA-OS tasks, by Poslad et al. [12]; or the automatons designed with SEdit in the MadKIT platform[4].

The multi-plan automatons of Bond, in particular, handle the parallelism using a simplified parallel sub-automatons composition model; whereas FIPA-OS tasks are by default executed in parallel: they may launch other sub-tasks, but this requires careful multithread synchronization.

[4] http://www.madkit.org/

4. Continuation-based approach

As we stated above, the need to manage parallelism and asynchrony without requiring dedicated threads implicates that behavioral components be manipulated like automatons, without blocking on threads. Besides, since existing MAS platforms comply with this strategy, our approach should also comply with it to facilitate integration.

On the other hand, being able to write stateful behaviors as simple routines, leveraging native control flow structures of the hosting language, as with the first strategy described in section 3.1, but without suffering from the usual inter-thread synchronization problems, would also make MAS programmers' life easier.

With respect to these issues, continuations offer a particularly elegant solution that combines the best of both worlds. Indeed, using continuations in MAS behavioral components for conversation management allows to:

- Capture the conversation or conversation branch state implicitly in the continuation, including call stack and local variables, so that the component may be used as an automaton, without having to explicitly manipulate its state;
- Get rid of inter-thread synchronization problems, since this model is not preemptive and transitions are executed sequentially.

4.1. Continuation-based automaton behavioral component

A continuation represents a program states, usually as an immutable object in object based language, but is not itself an automaton: it requires an encapsulating automaton object, similar to the generator implementation presented in section 2.1. Figure 6 shows a minimal automaton abstract base class, containing:

- a `current` attribute, storing a single current continuation for this automaton;
- the `activate()` method, the automaton step by step activation point, that updates the current continuation with a new one, corresponding to the new captured state;
- a continuation capturing method, `yield()`, meant to be used in implementations of the abstract `run()` method, or other methods called by `run()`, that interrupts the normal control flow and jumps to the `activate()` method;

```
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);
   }
}
```

Fig. 6. Simple continuation-based behavioral automaton abstract class, in Java, using Javaflow. Each time `activate()` is executed, the automaton advances one step. Each step terminates either by invoking `yield()`, or by the normal end of the `run()` method. In both cases this gives back control to `activate()`, which updates the continuation and returns normally.

In order to be usable in an actual agent implementation, however, these primitives are not sufficient: a way to consume events, like messages or timeouts, in the `run()` method, is necessary. In the case of a conversational agent platform, an agent is in principle provided with a message queue. A behavior component message queue may also be considered, associated to a message dispatch mechanism at the agent level. Such dispatch will typically be based on message response or conversation identification parameters, such as FIPA `conversation-id` message parameter.

4.2. Example and comparison

```
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();
      }
   }
}
```

Fig. 7. Continuation-based version of the 3 states automaton behavior of section 3.2, figure 5.

In comparison with the Jade component of figure 5, the version shown on figure 7 is much easier to read: it does not contain any object attribute to store and manage state information, nor does it require code fragmentation. The loop is materialized by a normal java `for` statement, increasing readability and reducing error proneness.

4.3. Pseudo parallelism and synchronization primitives

In order to address the parallelism requirements mentioned in section 3.1, it is necessary to introduce synchronization and pseudo-parallelism mechanisms in our automaton abstract classes. We use the term “pseudo-parallelism” since transitions will *in fine* always be executed sequentially. However, sub-automaton states corresponding to parallel branch of the conversation need to be stored and updated separately.

The case of multi-bilateral conversations (i.e. conversations involving an initiator and a group of participants), in particular, is quite common in protocols such negotiations or auctions. This case requires interacting in parallel with *n* participants, possibly synchronizing at key steps of the protocol, even if the various participants all play the same role and are thus governed by the same rules in the protocol.

We introduce two primitives to handle this kind of parallelism:

- `parallelize()`, a primitive allowing to switch to parallel mode, duplicating the current automaton in n sub-automatons, one per participant;
- and `join()`, the reverse primitive, usable only in parallel mode, that waits for all sub-automatons to reach this point (note that this does *not* entail blocking on a thread), before switching back to synchronous mode;

5. Testing framework

In the context of this work we have realized a testing prototype and framework based on Javaflow. Two abstract classes, `BilateralRole` and `MultiBilateralRole` (see table 1) implement the primitives mentioned in section 4.1 and 4.3. [5]

Table 1. Primitives provided by `BilateralRole` and `MultiBilateralRole`

Method signature	Description and side effect
`void activate()`	Automaton activation point
`void yield()`	Continuation capture, interrupts control flow
`Message receive()` `Message receive(Message.Type... types)`	Read the next event or message (possibly typed). Calls `yield()` if no event is available
`void parallelize()`	Switch to parallel mode, duplicate automaton
`void join()`	Switch back to synchronous mode

5.1. Auction and handshake protocol example

```
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())
         send(bestOffer, a == winner? accept: reject, a);
   } while(nbAnswered > 1);
}
```

`propose`, `inform`, `refuse`, `cfp`, `accept` and `reject` are values of the enumerated type `Message.Type`

Parallel section: here the automaton is duplicated in n sub-automatons, managed by *n* continuations, until all of them reach `join()`, which switches back to synchronous mode

Fig. 8. *`run()`* method of the behavior component `EnglishAuctionInitiator`.

In order to represent both forms of parallelism related to conversation management, we have defined four behavioral components, corresponding respectively to: the initiator and participant roles of a simple multi-bilateral handshake protocol, consisting in a linear sequence of *inform* message exchanges; and the initiator and participant roles of an English auction protocol, comparable to FIPA-English-Auction.

[5] By lack of space, the algorithm used to implement `parallelize()` and `join()` are not detailed here. The corresponding Java code may be downloaded from the author's web page.

In figure 8 the whole management of this auction protocol role is handled in a few simple lines of code, which demonstrates the applicability and elegance of this approach. An equivalent behavior, implemented using classical explicit finite state automaton, would require numerous states, conditions and switches, and would result in a fragmented code, difficult to read and debug.

Note that the variables `bestOffer`, `nbAnswered` and winner are here attributes of the `EnglishAuctionInitiator` class. These attributes are necessary here to communicate between sub-automatons in parallel mode, since Javaflow does not support parameter passing (see the generator example in section 2.1).

The participant roles, quite symmetrical to the initiator roles, do not require of course the use of `parallelize()` and `join()`, since they don't comprise parallelism.

5.2. Performance

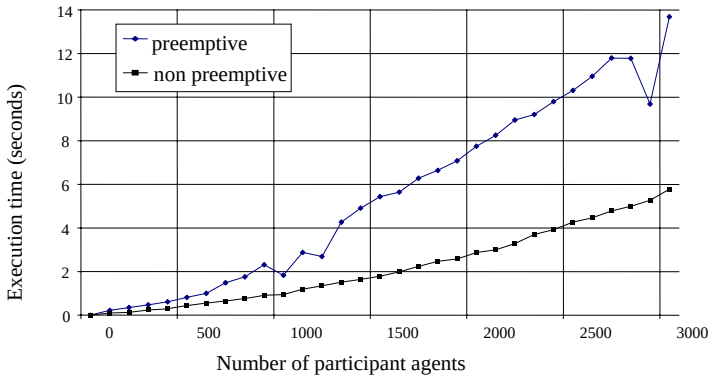

Fig. 9. Execution time, function of the number of participant agents. The upper curve corresponds to the preemptive version, the lower curve the non preemptive version.

Similarly to cooperative thread with respect to preemptive threads, a non preemptive activation of agents' behavior components, using continuation, give better performances. To evaluate this gain, we have compared the execution time of two versions of our prototype: a preemptive version with one thread per behavior component and per agent, and a non preemptive one, using round-robin activation and continuations. In this test the initiator and participant agents combine the handshake and auction behaviors. The results exhibit an average gain of 50% using Javaflow continuations.

6. Conclusion and future works

In this paper we have shown how to use continuations to facilitate conversational agents' behavioral components engineering. This approach allows an elegant, concise and intuitive coding of agents behavior dynamics, in the form of automatons written as "resumable" functions similar to coroutines or generators, while relieving the pro-

grammer from the explicit management of the automaton state and transitions. This approach also allows benefiting from the host language native flow control structure.

Its integration into existing platforms only depends on the support of continuations in the host language, and is quite straightforward. It gives to the designer a great flexibility in the mode of activation of agents and their behavior components.

In the general case, it allows to get rid of multithreading synchronization problems, but can still be combined with multithreading if necessary.

The main perspective to this work is the integration to Jade platform, by defining a `ContinuationBehavior`. If the adoption of continuations in the MAS community follows its adoption in the Web application programming community, they will have represent a promising new technology for agents, and become a common practice.

References

1. Allison, L.: Continuations Implement Generators and Streams. In The Computer Journal, volume 33(5), 1990. Oxford Journals, 460-465
2. Bellifemine, F., Poggi, A., Rimassa, G.: JADE – A FIPA-compliant agent framework. In proceedings of the International Conference and Exhibition on the Practical Application of Intelligent Agents and Multi-Agent Technology (PAAM 1999), London. 97-108.
3. Bölöni, L., Marinescu, D.: A Multi-Plane State Machine Agent Model. Acts of the International Conference on Autonomous Agents (AA 2000). Barcelona, Spain. ACM.
4. Hewitt, C.: Viewing control structures as patterns of passing messages. In Artificial Intelligence, volume 8(3), 1977. Elsevier. 323-364.
5. Haynes, C.T., Friedman, D. P., Wand, M.: Obtaining coroutines with continuations. In Computer Languages, volume 11(3), 1986. 143-153.
6. Jouvin, D., Hassas, S.: Role Delegation as Multi-Agent Oriented Dynamic Composition. In proceedings of the Intl. Workshop on Agent Technology and Software Engineering (AgeS 2002, collocated with NOD 2002), Erfurt, Germany.
7. Luck, M., McBurney, P., Preist, C.: Agent Technology: Enabling Next Generation Computing, A Roadmap for Agent Based Computing. Agentlink II report, 2003.
8. Mertz, D., Charming Python: Iterators and simple generators. IBM technical report, 2001. http://www-128.ibm.com/developerworks/library/l-pycon.html
9. Mertz, D., Charming Python: Generator-based state machines. IBM technical report, 2002 http://www-128.ibm.com/developerworks/library/l-pygen.html
10. Nwana, H., Ndumu, D., Lee, L., Collis, J.: ZEUS: A Toolkit and Approach for Building Distributed Multi-Agent Systems. In proceedings of the International Conference on Autonomous Agents (AA 1999), Seattle, USA. ACM press.
11. Pettyjohn, G, Clements, J., Marshall, J., Krishnamurthi, S., Felleisen, M.: Continuations from Generalized Stack Inspection. In proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2005), Tallinn, Estonia. ACM press.
12. Poslad, S., Buckle, P., Hadingham, R., The FIPA-OS Agent Platform Open Source for Open Standards. In proc. of International Conference and Exhibition on the Practical Application of Intelligent Agents and Multi-Agent Technology (PAAM 2000), Manchester, UK.
13. Strachey, C., Wadsworth, C.: Continuations: A mathematical semantics for handling full jumps. In Programming Research Group Technical Monograph PRG-11, Oxford, 1974. Re-edited in Higher-Order and Symbolic Computation, volume 13(1/2), 2000. 135-152.
14. Tate, B.: Crossing borders: Continuations, Web development, and Java programming, A stateful model for programmers, a stateless experience for users. IBM technical report, 2006 http://www-128.ibm.com/developerworks/java/library/j-cb03216/