BEcool: Towards an Author Friendly Behaviour Engine

Nicolas Szilas

TECFA, FPSE, University of Geneva,
CH 1211 Genève 4, Switzerland
Nicolas.Szilas@tecfa.unige.ch

Abstract. Virtual agents, to be expressive, not only need algorithms for displaying the subtleties of human behaviour, but also require environments and tools so that people can author them. Because powerful algorithms are sometimes difficult to author, a compromise has to be found between algorithmic sophistication and authorability. Our approach for providing expressive characters at the behavioural level is based on such a compromise. This paper provides a model for describing behaviours which is author focused, while enabling some interesting algorithmic features such as parallelism and inter-agent coordination. The model has been implemented and simulation results are displayed.

Keywords: Virtual characters, behaviour engine, expressiveness, authoring, authorability, interactive drama.

1 Agents' expressiveness

The visual representation of agents in virtual worlds has raised the question of expressiveness. These agents are not only rational agents, being able to take decisions according to their goals and the environment [8], but they also need to exhibit some lifelike attitude, and in particular display emotions within their behaviours [10].

   Within this general goal consisting of increasing the expressiveness of virtual agents, our research has followed a specific approach based on the two strong following assumptions:
–   Expressiveness can be improved at a level which is independent of the visual realism of agents. An extreme but illustrative case is the art piece called "Pixel blanc" [9], which displays the movement of one single pixel on a screen. The programming of the movement is such as the pixel seems to have life, "hesitate" before moving then suddenly "takes the decision" to move forward, etc. The visual representation of the agent consists of one pixel yet it is expressive, and a similar algorithm could be applied to a more realistic agent, a virtual dancer for example. This higher level of expressiveness, that we denote the behaviour level, is the focus of this paper.
–   To the question "where does the expressive behaviour come from", there are two answers: from the agent itself or from the human who created it. We chose the

latter. In that sense, the role of the agent is to mediate some expressiveness between two sets of humans: the creators of the agents and the users of the virtual world containing the agent. Of course, this is quite a specific mediation, which requires sophisticated algorithms for the agents. But it should not be omitted that to be effective, these agents need some creative people to use them to express themselves. We call these people "authors". One could argue that many existing systems do not use authors. They are designed by researchers and they are efficient. However, in these cases the researchers or engineers do play the role of authors, explicitly or implicitly.

Given these two assumptions, our goal is to design a behaviour engine in such a way that it is easily handled by authors, who obviously are not necessarily fluent in programming. The authoring issue is considered, in this research, as a primary requirement for the design of a behaviour engine, which contrasts with more classical approaches in which the most efficient engine is created before considering how an author could use it.

The rest of the paper is organized as follows. In the next section, the notion of behavioural level is more precisely explained. Then a review of existing behaviours engines is presented. Then we introduce *BEcool*, the proposed behaviour engine, its implementation, examples and future research.

2. The Behavioural Level

From the various systems using virtual agents that have been developed so far, a general scheme emerges regarding the software architecture. The systems tend to be hierarchically structured into three components as far as movement is concerned: animation, behaviour and reasoning. Animation includes body part movements, facial animation, path planing, gaze control. The animation level is informed by the upper levels to generate appropriate animations. Note that the animation level is usually combined with the other modalities such as speech. Behaviours are larger units which contains one or several animations. Behaviour management includes triggering the animations, running animations in parallel, blending animations, synchronizing animations between several characters, managing failures, managing priorities between competing behaviours. Reasoning is related to higher levels of intelligence such as strategic planing, decision making, affective reasoning. It is highly dependent on applications. In our current research, reasoning is performed by a central narrative engine [12], but other applications could use AI-based agent architectures.

The division into three levels is not always clear cut. For example, mechanisms for re-planing a path after a failure are usually included at the animation level, but they could also be managed at the behavioural level. It is however quite useful to modularize the architecture because each module can be developed and worked on separately. Furthermore, in terms of authoring, various skills are required for different levels. For example, graphical skill is needed for the animation level, not for the upper levels. Drawing an analogy with the field of drama, animation requires the skill of an actor, behaviours the skill of a director and reasoning the skill of a screenwriter.

There has been a lot of work about the control of agents' behaviours, often derived from previous work on robotics. We focus here on some systems that emphasize the role of the authoring.

An early system for authoring characters' behaviours is the system called *Improv* [7]. It allows the description of behaviours in terms of scripts. Scripts are sequences of simple actions (animations). Scripts are described in text form, making it easy for an author to write behaviours. The scripting language allows for non-deterministic behaviours, parallel behaviours and conditional choice between animations.

Other systems such as *Hap* use a hierarchy of goals [5]. Each goal contains a series of simple actions or subgoals, triggered if some conditions are met. Actions or subgoals can be triggered in parallel. Hap has been later extended into *ABL*, to carry out joint behaviours, that is the coordination of behaviours involving several characters [6]. *Hap/ABL* requires writing a list of goals with sophisticated parameters. It is a form of programming, making it unsuitable for authors. *ABL* was used to write an Interactive Drama [11], which demonstrated its usability for a large scale project, but also confirmed the fact that it requires proficient programming skills.

Other systems are based on finite state state machines [1][2][4]: a behaviour is represented as a node (state) in a graph, while transitions between behaviours are represented by arcs. More advanced systems use hierarchical finite state machines, which allow a behaviour to be described in nodes which can themselves be represented by an entire graph. This simplifies the representation of behaviours and enables reuse of sub parts of behaviours. Several finite state machines can run in parallel.

In terms of authoring, despite the visual representation (graphs), these systems require programming. In [1] for example, the graph structure is written in a dedicated language called HTPS, which is itself compiled into C++. HTPS is far more usable than C++, but it stills requires programming skills.

Commercial systems tend to focus more on the authoring aspect of behaviour authoring. A graphical environment for authoring hierarchical finite state machines was released ten years ago by a company named *Motion Factory* (the technology is now part of the *Softimage* software). The system not only provides a graphical editor for drawing hierarchical finite state machines for characters, but it also enables real time monitoring of the execution of the finite state machines, highlighting which states are active. However because the system is quite generic (for example any node or any transition could launch an animation or send a message), it remains difficult for the user (the author) to easily coordinate several animations beyond the simple case involving a sequence of animations.

Virtools' is another example of an authoring tool for 3D applications which includes a behaviour engine. Behaviours are described as flowcharts: building blocks are connected through a data flow. However, once again, *Virtools* is hard to use for non experts, because the charts cover all aspects of programming, including complex calculation, environment sensing, user interaction.

This short review illustrates that most behaviour engines developed so far have been focused on performance rather than authorability. In the following, we describe *BEcool*, an authoring tool that was designed with the intention that it be easy to use by an author.

3. BEcool

3.1 General specification

Our goal is to find a compromise between authorability and performance. In other words, *BEcool* is a behaviour engine which aims at being easy to author, departing from systems based on language programming, while proposing features beyond the simple sequencing of events. *BEcool* is based on three main principles:

– Behaviours are represented by oriented graphs, where each node in the graph is an animation and each arc is a transition. These graphs are meant to be visualized by an author.

– Two nodes belonging to two disconnected subgraphs can be active at the same time (parallelism).

– Animation coordination is managed by events generated by animation nodes.

Note that a similar approach is proposed in [15], but for the management of the entire narrative. The behaviour engine coordinates with two other modules, receiving data from the first, and feeding data into the second. The first module, in our case the narrative engine [14], launches the behaviours by providing the name of the behaviour and its parameters while the second module, typically a game engine, displays the animations triggered by the behaviour engine. The game engine is also responsible for sensing the environment by sending events to the behaviour engine.

Why do we expect this approach to provide an efficient answer to the issue of expressivity in behaviour authoring? First, while text-based scripts are intuitive for organizing sequences of events, they become programming as soon as parallelism is involved. Graphs on the other hand, with their two-dimensional nature, allow a more intuitive representation of parallelism. Furthermore, as argued by Wages et al. [15], graphs are becoming commonly used in software, making them more familiar to potential authors. Second, our approach does not try to represent any specific organization of animations within a behaviour. Contrary to other systems discussed above, which are substitutes for a general programming language, the behaviour description is highly constrained, allowing only a few types of node and links that the author can "play with" in order to describe a behaviour.

3.2 Behaviour description

In order to introduce the various features of *BEcool*, successive cases are presented.

Simple sequencing: the sequence of animations is simply represented by a chain of nodes (Figure 1). The plain arrow between two nodes means that the target node is activated when the source node is finished. When a node is activated, it triggers the animation attached to the node. More precisely, it sends a message to the animation engine, which executes the animation and then sends back a message when it has finished. One of the nodes is a start node, which means that it is activated as soon as the behaviour is launched. One of the nodes is an end node, which means that it sends a message that the behaviour is finished to the module that called *BEcool*. In Figure 1

and those which follow, the caption of the figure contains the command that is sent to the behaviour engine to execute a specific behaviour, where variable parameters are prefixed with an interrogation mark. In the figures themselves, each node contains an animation also described by variables, which are instantiated during runtime.
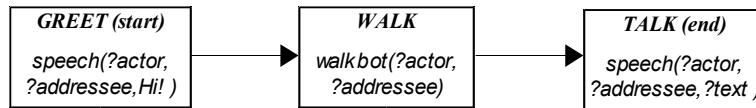

Fig. 1. Simple Sequencing. *Inform1(?actor , ?addressee, ?text).*

Branching: In Figure 2, at the start of the behaviour, one of two animations is triggered, walk or run, depending on the distance between two characters. The triggering of one of the transitions rather than the other depends now on events, associated to the transitions (arrows). These events (*far* and *close* in the example) are managed as follows: the behaviour engine sends not only the animation name and the associated parameters to the animation engine, but also a list of "sensors", that is a list of events, that have to be sent back when some conditions related to the 3D environment are met. In the example of Figure 2, the *Init* node asks the game engine to send *far* as an event when the distance between the actor and the addressee becomes greater than 5 meters. Note that this sensor is specific to the node it comes from. If the conditions of the *far* event mentioned above are met when another node is active, no event is sent by the animation engine.
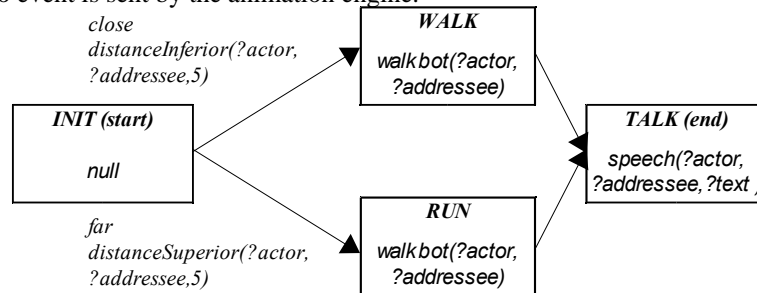

Fig. 2. Branching. *Inform2(?actor , ?addressee, ?text).*

Parallelism: In Figure 3, the mechanism of event management is used for the synchronizing of two parallel subgraphs. While the *Walk* animation is launched by the first node, the event *close* is sent back by the animation engine as soon as the other character is at a distance smaller than 2 meters.
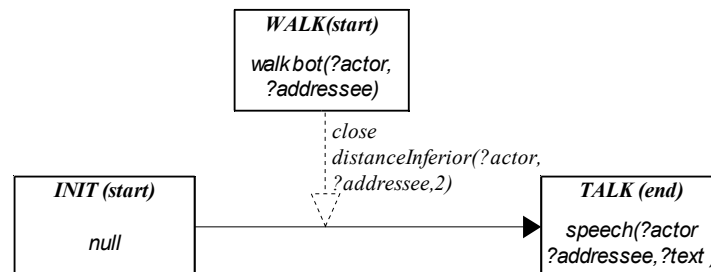

Fig. 3. Parallelism. *Inform3(?actor , ?addressee, ?text).*

When this happens, this triggers another transition, in another subgraph, to activate the node *Talk*. this is represented by a dash arrow, pointing to the regular transition. This means that the target node is activated if both the source node is finished (here *Init*) and the event (here *close*) has been triggered.

Inter-actors coordination (joint behaviours): in order to coordinate several actors, we made the choice of a centralized authoring. One single behaviour is directing several actors, as if these actors were one entity. This approach is less general than an autonomous agent architecture [6], but it highly simplifies the architecture. In particular, no complex plan sharing is needed between two autonomous actors. More importantly, such an approach is more intuitive for an author, because it shares similarities with the activity of a stage director, who coordinates several real actors. In Figure 4, the behaviour is composed of two subgraphs, one for each of the two characters involved in the behaviour. The first subgraph means that the actor calls the addressee and then talks to him/her (when s/he gets close). The second subgraph means that the addressee walks towards the actor when s/he is beckoned and then listens to the actor. These two subgraphs are linked by two events, *end* (to notify that the call is finished) and *close* (to notify that the addressee is sufficiently close to start to talk).
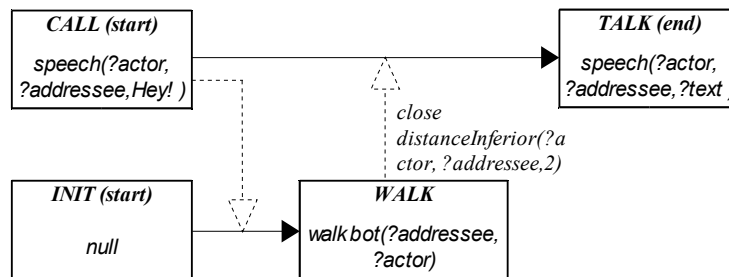

Fig. 4. Inter-actors coordination. *Inform4(?actor , ?addressee, ?text).*

3.3 Management of multiple behaviours

A behaviour is launched by sending a message to *BEcool* containing the name of the behaviour and a list of parameters.

When several behaviours are running at the same time, it might happen that the same actor is involved in two concurrent behaviours. A design choice has to be made between cancelling one of the behaviours, suspending one of the behaviours and restarting it later, blending the behaviours [3]. Again, a simple solution has been chosen, based on the priority affected to the behaviour when it is launched. When a character is involved in a running behaviour and a new behaviour involving the same character is asked to run (conflicting situation) the following rule is applied:

> *IF* the priority of the new behaviour is equal to or greater than the priority of the running behaviour,
> *THEN* cancel the running behaviour (*failure* message sent), without restarting and start the new one
> *ELSE* send a *failure* message for the new behaviour.

In the current implementation, the fact that a character is involved in a behaviour is computed by checking if this character is one of the parameters of the behaviour. This is a clear limitation because even if a character only plays a peripheral role in a

behaviour, it is considered as "busy" during the whole running of the behaviour. This could be improved in the future.

It has been mentioned above that there is no restart mechanism at the level of the behaviour engine. Thus BEcool does not take into account the continuous changes in the environments. But such mechanism can be implemented at the upper level, by the narrative engine in our case. In the whole architecture, that will be detailed below, the narrative engine might decide to relaunch the behaviour, if it is relevant from a narrative point of view (or a different point of view, if another type of module is managing the behaviour engine).

In some cases however, a behaviour might be interrupted by another behaviour, but it does not mean that the interrupted behaviour has failed. Suppose for example that John says to Mary: "I love you, I love you". If this behaviour is interrupted during the second utterance of the word "love", then an author should be able to decide that the behaviour is considered as successful, despite the technical failure. Thus we introduce two different messages:

– the *success* message, which might be sent before the end of the animations;
– the *end* message, which is sent when the last animation has succeeded.

If the *success* message is sent to the upper level, then the behaviour is considered as successful and the consequences of this can be computed, even if, later, the behaviour fails. If only the *end* message is sent to the upper level, then the behaviour is considered as both finished and successful. If the *failure* message is sent, then it is considered as a failure only if no *success* message has been received before. An example of such behaviour is depicted in Figure 5.
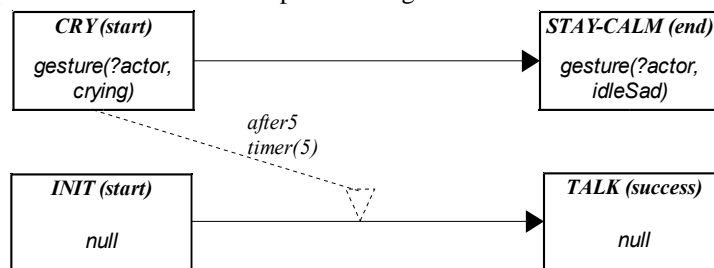

Fig. 5. Example of a behaviour which sends a *success* message before being finished.

3.4 Programming interfaces

A behaviour engine has two programming interfaces: one for the upper level (narrative engine for example) and the other for the lower level (animation engine).
At the upper level, a specific behaviour is launched by sending a launch message to *BEcool,* which contains the type of behaviour to be launched (*inform*, *gesture* for example) and the associated parameters. For example, the upper level would send the following data: *inform(john,mary,"Hi Mary!","did you know that Bill broke his arm?").* In this example, there are two text messages in the parameters because the behaviour has two speech bubbles, one for greeting, one for the actual content of the information. Note that the actual string syntax is different (see Fig. 8 caption). *BEcool*

then starts or tries to start the behaviour. During the execution, it sends back to the upper level a feedback among the following: *failure*, *success*, *end*.

At the lower level, *BEcool* sends messages containing the name of the animation (such as *walkbot*, *gesture*, *speech*, etc.), the associated parameters, which are either parameters of the behaviour sending the animation, or some hardcoded data and finally a list of events. For example, the following data could be sent: *walkbot(john,mary,(close,distanceInferior,john,mary,1))*, which means "launch the animation of john walking to Mary and during this animation send the *close* message as soon as the distance between john and mary is smaller than 1".
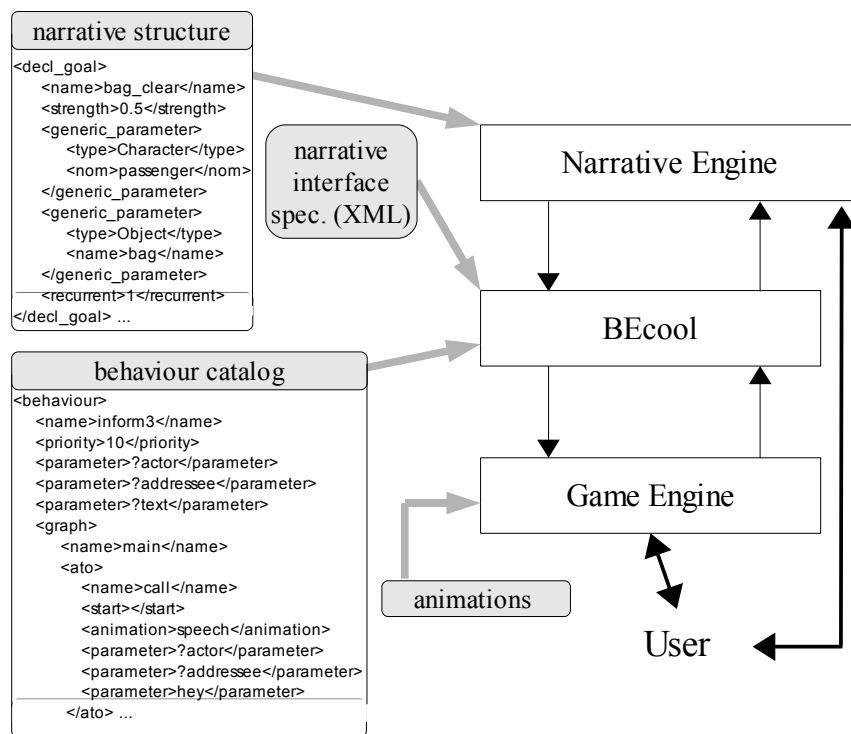

Fig. 6. Global architecture of a system using BEcool.

3.5 Technical architecture

In this section, the current implementation of *BEcool* is described, as well as its integration into a global technical architecture (see also [14] fore details). *BEcool* is developed as an independent program written in plain Java. No particular programming formalism has been used in the implementation. This program communicates with the two other modules via sockets. The upper module is either a narrative engine previously developed by the author [12][13] or a "tester", a simple Java program allowing the user to manually enter commands to be sent to the behaviour engine. Figure 6 represents the architecture, with the narrative engine.

The narrative engine is the *IDtension* program, fully written in Java. *IDtension* generates high level narrative actions such as "John informs Mary that Bob want to steal the money from Greg", or "John gives a letter to Mary".

The animation engine is a customization of *Unreal Tournament 2004*, a commercial game engine delivered with the eponymous game. The customization consisted in adding the socket communication and the event management, as described above.

Behaviours, as depicted in Figures 1 to 4, are coded in an XML file called the behaviour catalogue. The main elements handled by the grammar are:

– ato: node in the behaviour graphs (*ato* stands for *atomic behaviours*),
– link: simple link between nodes,
– conditionalLink: event-triggered link between nodes,
– event: specific event generated by a node when certain conditions are met,
– condition: condition related to an event.

The communication between the narrative engine and the behaviour engine is also specified with another XML file, called the narrative interface specification, in order to match the type of high level actions generated by the narrative engine (inform, encourage, dissuade, perform, etc.) to the behaviours. Typically, two different narrative actions can be played by the same behaviour. This XML file enables independence between the modules: there is no need to hard code within the narrative engine the names of the behaviours, neither the usage of their parameters.

4. A complete example

In order to better illustrate the behaviour engine discussed in this paper, we detail hereafter a full example of a behaviour. This behaviour is used whenever a non player character wants to convey an information to another non player character. In natural language, this behaviour can be described as follows: "the first character walks towards the other one (or just turns towards him if they are close), greet him when arriving at 2 meters – which makes the second character turns towards him – starts uttering the main message when arriving at 1 meter, and finally stops in front of the second character" while the main message is finishing.

This example is depicted in Figure 7. It involves linear sequencing, branching, parallelism, and inter-character coordination. It contains 8 animation nodes grouped into three subgraphs: one for the choice between walking or just turning, another one for the first character main speech sequencing and the last one for the second character's behaviour (turning). The dashed lines from the *TURN* and *WALK* nodes are designed to make sure that the end of these two alternative initial animations, the next animation in the speaker character is launched. The *WALK* node triggers two types of events, *close* and *very_close*, so that the *GREETING* and the *INFO* nodes are successively activated according to the distance between the two actors, in parallel to the walk animation. Figure 8 reproduces screenshots of the simulation of this behaviour, in the case where the two characters are far from eachother.
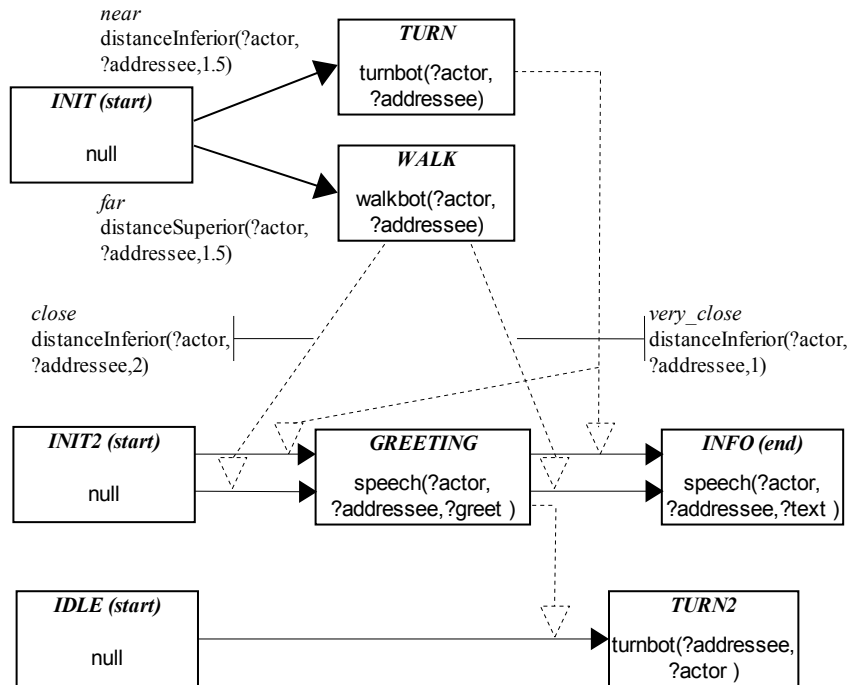
Fig. 7. A complete example of a behaviour: Inform_NPC(?actor, ?addressee, ?greet, ?text). This is an information transmission between two characters.
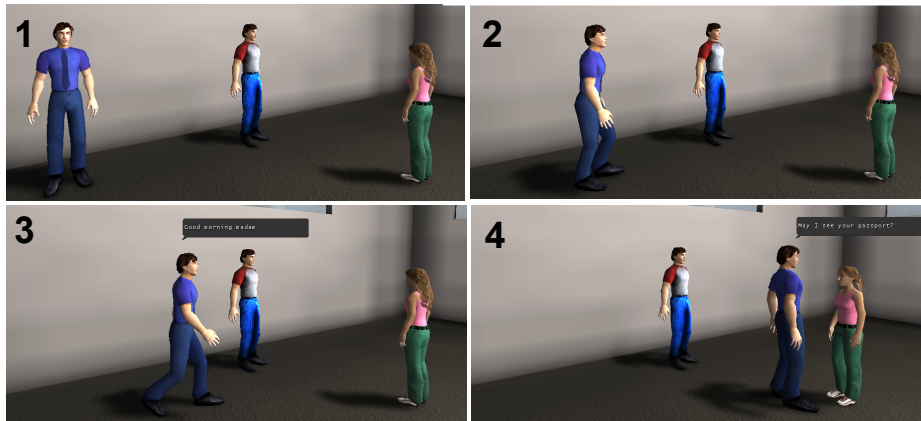

Fig. 8. Four successive screenshots of the execution of the behaviour depicted in Fig. 7. The string message sent to the behaviour engine is: "#launch::1234::inform_NPC::Bill::Kim::Good morning madam::May I see your passport?"

5. Conclusion and future work

In this paper, *BEcool*, an implemented behaviour engine has been presented. It has been designed to favour expressive authoring over agent intelligence. As a result, behaviours are fully described by visual graphs containing nodes for animations, arrows for sequencing, arrows' labels for environment's sensing (events) and dashed arrows for event-based animation triggering. This simple syntax allows sequencing, branching, parallelism and inter-characters behaviours.

The simplicity of authoring comes not only from the simplicity of this syntax, but also from the clear separation between levels. Behaviour authoring only involves the coordination of animations, not the "why" of the behaviours (reasoning), neither the "how" of the behaviours (animation level). In large scale production, these three levels would certainly involve three populations of authors.

The natural extension of this work is the development of a visual authoring tool. This tool would enable an author to directly draw the graphs within a dedicated software, without writing any XML line. This tool would produce the XML file needed by *BEcool* to run the behaviour (behaviour catalogue). The authoring tool development, that constitutes a considerable engineering work is a necessary step for the evaluation of the effective easiness of the proposed approach. This development has been initiated, using the Jgraph, a Java library for graph editing

Beyond the lack of a visual authoring tool, is *BEcool* fully usable for a non programmer author? The graph depicted in Fig. 7 for example is not that easy to design. During our own usage of the graphs, we found that:

–   It was easy to omit a case (a specific situation), resulting in a deadlock during the execution of a behaviour. Most of the time, we corrected the graph before the execution of the behaviour, but a regular user would certainly need to debug such cases.

–   There is several ways to describe the same behaviour. This might be seen as an advantage, in terms of flexibility, but we find it problematic in terms of easy authoring. Indeed, an author should not waste time hesitating between possibilities for expressing a behaviour.

These remaining authoring difficulties suggest to define some graph templates, that is predefined graph structures that authors could reuse when writing a behaviour. These templates would guide the author by providing animation structures that occur recurrently in behaviours.

Despite the current limitations mentioned above, *BEcool* appears to be a promising tool for behaviour authoring, because it allows a totally visual representation of rather complex behaviours. Furthermore, its representation with graphs is quite compatible with the practice of storyboarding in the movie making industry.

References

1. Donikian S.: HPTS: a behaviour modelling language for autonomous agents. In Proc. of the fifth int. conf. on Autonomous agents. ACM Press (2001) 401-4082.
2. Granieri, J., Becket, W., Reich, B., Crabtree J., Badler, N.: Behavioral control for real-time simulated human agents. In proc. of the 1995 Symposium on Interactive 3D Graphics, Monterey, CA (1995) 173-180
3. Lamarche, F., Donikian, D.: Automatic orchestration of behaviours through the management of resources and priority level. In proc. of AAMAS'02, Volume 3, Bologna, Italy. (2002) 1309-1317
4. Lau, M., Kuffner, j.: Behavior planning for character animation. In ACM SIGGRAPH / EUROGRAPHICS Symposium on Computer Animation. ACM Press (2005) 271-280
5. Loyall, A. B., Bates, J.: Hap: A reactive, adaptive architecture for agents. Technical Report CMU-CS-91-147, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA (1991)
6. Mateas, M., Stern, A.: A Behavior Language: Joint Action and Behavior Idioms. In: H. Prendinger & M. Ishizuka (eds.): Life-like Characters: Tools, Affective Functions and Applications. Springer (2004)
7. Perlin, K., Goldberg, A.: Improv: A System for Scripting Interactive Actors in Virtual Worlds. In: Proc. of SIGGRAPH 96, New Orleans, LA. ACM SIGGRAPH, (1996) 205-216
8. Russel, S., Norvig P.: Artificial Intelligence: a modern approach. 2nd edn. Prentice Hall, Saddle River, NJ (2003)
9. Schmitt, A.: Le Pixel Blanc. http://www.gratin.org/as/txts/lepixelblanc.html.
10. Smith, S., Bates, J.: Towards a Theory of Narrative for Interactive Fiction. Technical Report CMU-CS-89-121, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA (1989)
11. Stern A., Mateas M.: Integrating Plot, Character and Natural Language Processing in the Interactive Drama Façade. In: Göbel et al. (eds) Proc. TIDSE'03. Frauenhofer IRB Verlag (2003) 139-151
12. Szilas, N.: A Computational Model of an Intelligent Narrator for Interactive Narratives. Applied Artificial Intelligence, to appear (2007)
13. Szilas, N.: Interactive Drama on Computer: Beyond Linear Narrative. In Papers from the AAAI Fall Symposium on Narrative Intelligence, Technical Report FS-99-01. AAAI, Press Menlo Park (1999) 150-156
14. Szilas, N., Barles, J., Kavakli, M.: An implementation of real-time 3D interactive drama. Computers in Entertainment Vol. 5 , Issue 1 (Jan. 2007)
15. Wages, R., Grützmacher, B., Conrad, S.: Learning from the movie industry: Adapting production processes for storytelling in VR. In: S. Göbel et al. (eds.) Proc. of Technologies for Interactive Digital Storytelling and Entertainment (TIDSE 04). Lecture Note in Computer Science, Vol 3105. Springer Verlag, Berlin Heidelberg New York (2004) 119-125