Developing Natural Language Enabled Games in (Extended) SCXML

Jenny Brusk[1] and Torbjörn Lager[2]

[1] Department of Game Design, Narrative and Time-based Media, Gotland University College, Sweden
[2] Department of Linguistics, Göteborg University, Sweden
Email: jenny.brusk@hgo.se, torbjorn.lager@ling.gu.se

Abstract

The World Wide Web Consortium (W3C) has a standard in the pipeline – called SCXML – that may turn out to be very suitable for the design and implementation of games, in particular games featuring (possibly multimodal) natural language dialogue. We see three main reasons why SCXML may be a good fit for the game industry: 1) SCXML is all about statecharts – a powerful extension of finite-state machines – and we argue that statecharts has the right kind of expressivity for game design and development, 2) SCXML is an XML dialect (soon to be) endorsed by the W3C, and will thus become a part of a web infrastructure comprising speech technology and telephony, as well as other useful technologies for building games of certain genres, and 3) SCXML is designed for extensibility and it appears that it would be fairly straightforward – and very worthwhile – to build a game oriented extension ("profile") around the SCXML core. The paper also presents an experimental implementation of SCXML, accessible from a user-friendly web-interface at <http://www.ling.gu.se/~lager/Labs/G-SCXML-Lab/>.

1. INTRODUCTION

We want this paper to serve as a pointer to the fact that the World Wide Web Consortium (W3C) has a standard in the pipeline – called SCXML – that may turn out to be very suitable for the design and implementation of games, in particular games featuring natural language dialogue.[1] The W3C does not explicitly promote SCXML as a tool for building games, but rather as (part of) a framework for building (multimodal) dialogue systems. Still, there are three main reasons why SCXML may be a good fit for the game industry:

1. SCXML is all about statecharts – a powerful generalisation/extension of finite-state machines. We argue that statecharts has the right kind of expressivity for game design and development.
2. SCXML is an XML dialect (soon to be) endorsed by the W3C, and will thus become an integral part of a bigger picture – a web infrastructure comprising speech technology, telephony, and other useful technologies for building games of certain genres.
3. SCXML is designed for extensibility, and it appears that it would be fairly straightforward – and very worthwhile – to build a game oriented extension ("profile") around the SCXML core.

The first three sections of this paper deals with each of these points, followed by a section where an implementation (accessible online) is presented, and finally by a section where we summarize and draw some conclusions.

1 The present paper is based on the January 2006 SCXML working draft, but with a few syntactic simplifications.

2. SCXML = STATE CHART XML

SCXML can be described as an attempt to render Harel statecharts (Harel 1987) in XML. Harel developed his statecharts as a graphical notation for specifying reactive systems in great detail. In its simplest form, a statechart is just a finite state machine, where state transitions are triggered by events appearing in an event queue. Let us begin with a very simple example in the form of a 'game' where a display will show "YOU WON! Play again?" if the player pushes a Play-button, and then "Push to Play" if he pushes the Again-button. Not much fun, but it serves to introduce some notation.

Just like ordinary finite-state machines, statecharts have a graphical notation – for "tapping the potential of high bandwidth spatial intelligence, as opposed to lexical intelligence used with textual information" (Samek 2002). The statechart controlling our game could simply be:

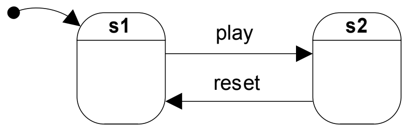

Figure 1: Game statechart

Any statechart can be translated into a document written in the linear XML-based syntax of SCXML. Here, for example, is the SCXML document capturing the above statechart, and thus the flow of our simple game:

```
<scxml target="s1">
  <state id="s1">
    <transition event="play" target="s2"/>
  </state>
  <state id="s2">
    <transition event="reset" target="s1"/>
  </state>
</scxml>
```

An SCXML document such as this can be executed by an SCXML conforming processor, greatly simplifying the step from specification into running game application.

Harel (1987) also introduced a number of (at the time) novel extensions to finite-state machines, which are also present in SCXML, including:

- Hierarchy
- History
- Concurrency
- Broadcast communication
- Datamodel (a.k.a. "extended state variables")

Hierarchy. Statecharts may be hierarchical, i.e. a state may contain another statechart down to an arbitrary depth. From a methodological point of view this is important, since it allows us to apply the principles of refinement (a top-down design process in which a state is refined into a number of substates and the transitions between the substates spelled out in detail) and clustering (a bottom-up design process in which a number of similar states are grouped together under the umbrella of a superstate). These principles are very general, and they are certainly relevant to game design as well.

History. A complex state may contain a history state, serving as a memory of which substate S the complex state was in, the last time it was left for another state. Transition to the history state implies a transition to S.

The statechart in Figure 2 exemplifies hierarchy and history by enhancing our simple game with a pause-and-resume functionality:

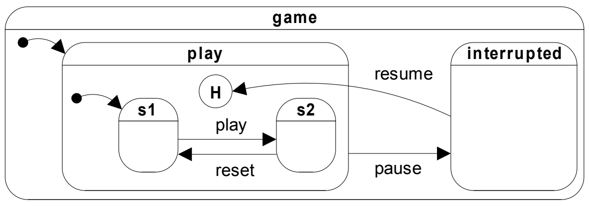

Figure 2: Pause-and-resume

Suppose that the current state is `s2`, and that an event "pause" appears in the event queue. The transitions leaving `s2` are tried from the inside and out, and since "reset" does not match the first event in the queue, but "pause" does, a transition to the `interrupted` state takes place. If a "resume" event shows up in the queue, the system transfers to the history state `h`, which implies a transition back to the `s2` state again.

The SCXML documents corresponding to Figure 2 would be:

```
<scxml target="play">
  <state id="play" target="s1">
    <history id="h" target="s1"/>
    <state id="s1">
      <transition event="play" target="s2"/>
    </state>
    <state id="s2">
      <transition event="reset" target="s1"/>
    </state>
    <transition event="pause"
                target="interrupted"/>
  </state>
  <state id="interrupted">
    <transition event="resume" target="h"/>
  </state>
</scxml>
```

Already at this point, it should be clear that statecharts, and thus also SCXML, are well-suited for the design and implementation of game flow, at least the kind of game flow where the progression between game states is more or less explicit, and perhaps even scripted.

Concurrency. Two or more statecharts may be run in parallel, which basically means that that their parent statechart is in two or more states at the same time. This is an important mechanism for introducing independency and orthogonality into a design. Concurrency may for example be useful when the flow of a game is not (only) modelled directly (or scripted), but when the NPCs[2] – their states-of-mind, states-of-body, as well as their (verbal and non-verbal) behaviors – are modelled in the hope that a good game will emerge from the interaction between different NPCs and between NPCs and human players. In such cases it makes sense to model each NPC as a separate statechart, running in parallel with each other, and running in parallel with a statechart modelling the environment.

Just to give a hint of how this might look like in SCXML, we give a high-level view of an architecture where an NPC's 'emotions' and its reactive behaviors are working independently. First as a statechart in the graphical notation:

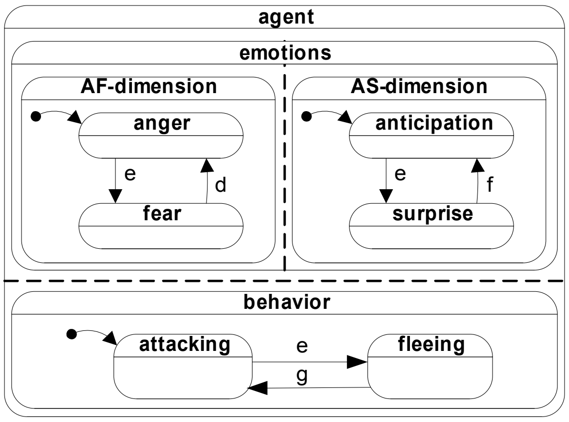

Figure 3: Emotion and behavior

and then translated into SCXML:

```
<parallel id="agent">
  <parallel id="emotions">
    <state id="AF-dimension">
      <state id="anger">
        <transition event="e" target="fear"/>
      </state>
      <state id="fear">
        <transition event="d" target="anger"/>
      </state>
    </state>
    <state id="AS-dimension">
      <state id="anticip">
        <transition event="e" target="surprise"/>
      </state>
      <state id="surprise">
        <transition event="f" target="anticip"/>
      </state>
    </state>
  </parallel>
  <state id="behavior">
    <state id="attacking">
      <transition event="e" target="fleeing"/>
    </state>
    <state id="fleeing">
      <transition event="g" target="attacking"/>
    </state>
  </state>
</parallel>
```

The idea is that in the event of (say) an explosion (the event named "e") the NPC will end up in the states of fear, surprise and flight behavior. Then, if his friend dies (the event named "d"), fear will turn into anger, and he will attack instead. Note that if we did not have access to concurrency, we would have had to distinguish the *atomic* state FearSurpriseFleeing from

2 NPC = Non-player character, i.e. a game character not controlled by the player.

other atomic states such as AngerSurpriseFleeing, AngerAttackFleeing and what have we. And as soon as we wanted to extend (say) the emotional dimensions from two into (say) three, we would see a large increase in the number of states and transitions required. It is well-known (cf Harel 1987; Horrocks 1999) that concurrency and to a some extent also hierarchy are the means by which this often cited problem with ordinary finite state machines – the exponential growth in the number of states and transitions – can be treated.

Broadcast communication. One statechart S_1 may communicate with another statechart S_2 (running in parallel with S_1) by placing, in the global event queue, an event that triggers a transition in S_2. Since the event can in principle be detected by any transition in any state in the statechart, this is often referred to as "broadcast communication".

We exemplify parallel statecharts and the communication between their substates with an SCXML document featuring two 'agents' playing ping-pong.

```
<scxml target="start">
  <parallel id="start">
    <state id="Pinger">
      <onentry>
        <send event="ping"/>
      </onentry>
      <transition event="pong">
        <send event="ping" delay="1s"/>
      </transition>
    </state>
    <state id="Ponger">
      <transition event="ping">
        <send event="pong" delay="1s"/>
      </transition>
    </state>
  </parallel>
</scxml>
```

Note the use of 'targetless' <transition> elements here, where the matching of an event results in the running of the transition's actions (executable content such as the <send> elements) but not in any actual transitions. Note also that we have delayed each sending of an event with one second, just to make the speed of the ping-pong game a bit more realistic.

Datamodel. SCXML gives authors the ability to define a data model as part of an SCXML document. A data model consists of a <datamodel> element containing one or more <data> elements, each of which may contain an XML description of data. The value of the `cond` attribute in a <transition> element may be an expression referencing the data, and transitions may thus be conditioned on the data. The <assign> element may be used in actions, or in <onentry> or <onexit> elements, to modify the data. As a simple illustration, the following state (which could be one of several parallel states) serves as a score counter, transferring to the "GameOver" state when the count is 3, and incrementing the count each time the "point" event shows up in the global event queue.

```
<state id="Scorer">
  <datamodel>
    <data name="Score" expr="0"/>
  </datamodel>
  <transition cond="Score==3" target="GameOver"/>
  <transition event="point">
    <assign name="Score" expr="Score+1"/>
  </transition>
</state>
```

3. SCXML IN THE BIGGER PICTURE

SCXML is meant to be used to control the flow of an application, be it a game or a (multimodal) dialogue system. It is not intended to manage lots of data, nor to directly interact with the user. In this section we will look closer at one particular kind of user interaction – dialogue using natural language.

There are several ways in which natural language dialogue may come into play in games. Assuming the commonly made distinction between game (G), player (P), player character (PC) and non-player character (NPC), and stretching the notion of dialogue somewhat, we may distinguish between:

- P in dialogue with G: Games may be 'voice controlled'. Instead of hitting the P button in order to pause a game, the player may just say "pause".
- P in dialogue with PC: Player is directing his player character using dialogue.
- P in dialogue with P: Player talking to player, using chat or voice.
- NPC in dialogue with NPC: The use of natural language for commenting on the states and the events of a game. Examples such as the conversations between NPCs in SIMS, and the soccer commentators in FIFA 200X games comes to mind.
- P in dialogue with NPC: For the purpose of letting NPCs provide the player with background story, quests and directions for progressing the game, but also in order to uphold 'social relationships' with NPCs. Dialogues will thus sometimes be task oriented, sometimes of a more socially motivated kind.

We believe that the most successful natural language enabled games will treat natural language dialogue as an integral part of the game, rather than something added on as an afterthought. We believe that in order to create such games, designers need to consider dialogue flow a part of the game flow, treat dialogue actions on par with other game actions, and think of the current game state as comprising also the state(s) of the dialogue(s) taking place at the current point in time. Indeed, we think that the best games will be built by those who are just as skilled at dialogue flow design as they are skilled at game (flow) design in general, and perhaps future books about game design patterns will have a chapter or two where good *game* dialogue design patterns are presented and explained.

To drive this point home even further, note that conversations between humans often are of the multimodal kind, and that realistic dialogue with NPCs therefore should be too. An NPC should be able to nod instead of saying "yes", or nod *and* say "yes" at the same time. Thus, the boundary between controlling the visual appearance and behavior of an NPC – how it looks and what it does – and its natural language capabilities – what it says – is not very clearcut. Thus, these things should better be controlled and synchronized using one and the same mechanism.

As we have indicated already, SCXML is not supposed to directly interact with the user. Rather, it requests user interaction by invoking a *presentation component* running in parallel with the SCXML process, and communicating with this component through asynchronous events. Presentation components may support modalities of different kinds, including graphics, voice or gestures. Concentrating on presentation components for spoken language dialogue (a.k.a. "voice widgets") we may assume that they include things like:

- A Text-To-Speech (TTS) component for presenting the player with spoken information.
- An Automatic Speech Recognition (ASR) component to collect spoken information from the player.
- A *combination* of TTS and ASR to implement something akin to a *field,* prompting for, and collecting, a value of one single parameter from the player.
- A form-filling algorithm (a.k.a. FIA) running over an (internal) datamodel, and using TTS and ASR for output and input, respectively, and thus implementing something akin to a *form,* collecting values for a *set* of parameters from the player.
- Other dialogue management components, tailored to particular conversational modes, e.g. social talk or negotiation.

Note that presentation components may be simple, as the first two components in the above list, or complex, as the last three. (The very last one might of course be *very* complex.) A complex component may in fact be made up from other (simple or complex) components (perhaps using SCXML for controlling the interplay between their parts, perhaps not), but for all intents and purposes their complexity is hidden from the developer, and the only way to communicate with them is through the global event queue that they share with the invoking SCXML document.

Example 1. We could enhance our simple game with *voice-controlled* pause-and-resume functionality by invoking an ASR component, like so:

```
<state id="s1">
  <invoke id="v"
          target="v3:grammar"
          src="play_or_pause.vxml"/>
  <transition event="*.play" target="s2"/>
</state>
```

Note that the semantics of <invoke> dictates that the ASR as well as the button are deactivated again as soon as the state s1 is left, i.e. as soon as either presentation component generates an event matching the pattern *.play.

Example 2. In order to collect a "Yes" or "No" from the player as a response to a question from an NPC, the question and the grammar for recognizing the response could be encoded in a VoiceXML document "yesno.vxml" and invoked from SCXML like so:

```
<state id="YesOrNo">
  <invoke id="yn"
          target="v3:field"
          src="yesno.vxml"/>
  <transition event="yn.yes" target="Yes" />
  <transition event="yn.no" target="No" />
  <transition event="yn.*" target="YesOrNo" />
</state>
<state id="Yes" ... />
<state id="No" ... />
```

Example 3. The above SCXML snippet invoked a simple VoiceXML *field*, collecting only one value for a parameter from the user, but for a more complex interaction with the user, the developer may instead choose to invoke a VoiceXML *form*, consisting of several fields, that will be filled with information during the course of the interaction with the user and returned to the SCXML interpreter afterwards, in the form of an event with a payload representing the result. The invoking state may look as follows

```
<state id="buyTicket">
  <invoke target="v3:form"
          src="buyTicket.vxml"/>
  <transition event="buyTicket.done".../>
  <transition event="buyTicket.cancelled"
              target="chat".../>
</state>
```

and the invoked VoiceXML document like this:

```
<vxml version="2.1">
  <form id="get_from_and_to_planets">
    <grammar src="from_to.grxml"
             type="application/srgs+xml"/>
    <initial name="bypass_init">
      <prompt>Fly from and to where?</prompt>
      <nomatch count="1">
        Sorry dude, but I didn't get that.
      </nomatch>
      <nomatch count="2">
        I'm sorry, I still don't understand.
        <assign name="bypass_init" expr="true"/>
      </nomatch>
    </initial>
    <field name="from_planet">
      <grammar src="planet.grxml"
               type="application/srgs+xml"/>
      <prompt>From which planet?</prompt>
    </field>
    <field name="to_planet">
      <grammar src="planet.grxml"
               type="application/srgs+xml"/>
      <prompt>To which planet?</prompt>
    </field>
  </form>
</vxml>
```

Initially, a player is expected to respond with (say) "From Mars to Venus", in order to fill both fields in one shot. However, if the response is not recognized by the ASR (using the specified grammars) after two attempts, the NPC tries instead to split the initial question into two parts, expecting responses such as "Mars" or "Venus", filling one field at a time. Going into more detail about VoiceXML is beyond the scope of this papers. Suffice it to say that VoiceXML is a well-proven and mature technology for the design and implementation of task-oriented dialogue – a technology that a game developer willing to work with SCXML would be able to tap right into.

Example 4. But VoiceXML will most likely not work very well for 'social chat' kinds of conversations, and we therefore propose a strategy in which the player (or the NPC) is allowed to 'escape' from a task oriented dialogue into a 'social chat' kind of conversation. The idea would be to try to detect, from inside the VXML form, that the form based dialogue is not likely to succeed (e.g. by giving up after having made three attempts at understanding what the user is saying) and instead return an event that will cause a transfer to a state that will invoke a dialogue manager/ASR/TTS combination able to deal with 'social chat'.[3] After a while (say ten seconds

3 We use AIML as an example here (Wallace 2005).

of chat), or if the NPC gets some sort of indication that the user would like to buy that ticket anyway, we can always let the NPC return to the form-based, task oriented dialogue again, if we think that this would be important for the progression of the game.

```
<state id="chat">
  <invoke id=”c”
          target="AIML"
          src="chat.aiml"/>
  <transition event="c.done"
              target=”buyTicket.../>
  <transition delay=”10s” target=”buyTicket.../>
</state>
```

Example 5. SCXML+VoiceXML is not just a framework for building spoken dialogue systems, but also for controlling telephony – a framework in which technologies for voice recognition, voice-based web pages, touch-tone control, capture of phone call audio, outbound calling (i.e. initiate a call to another phone) all come together, creating a marvellous playground for building new and innovative games. As an example of such a game (which could have been built using SCXML but was not), we mention Electronic Arts' Majestic – a psychological web-based thriller released in 2001 that actually involved the players in the game, sending them e-mails, calling them on the phone and sending them Instant Messages when they least expected.

4. EXTENDING SCXML FOR GAME DEVELOPMENT

SCXML is designed with extensibility in mind (cf Barnett et al. 2006), and our own investigations (which also include some explorative implementation work that we will report on in the next section) suggest that there is indeed room for simple extensions that will increase the expressivity of SCXML considerably.

In games, but not so often in 'serious' applications, a certain level of unpredictability can be an advantage. Since the W3C are designing SCXML with 'serious' applications in mind, they have not included any simple means for making choices based on chance. It can be done, but only in a rather clumsy way (witness appendix D in the January SCXML draft specification). As a first stab at this problem we propose that a new attribute `prob`, taking (an expression evaluating to) a value p between 0.0 and 1.0 and defaulting to 1.0, be added for the <transition> element, with the semantics that if all (if any) other conditions are satisfied, the transition in question is taken with probability p. For example, if the state `s0` in the following snippet is reached, either `s1`, `s2` or `s3` will immediately be entered, with an equal chance (33%) for each of the alternatives.

```
<state id="s0">
  <transition prob=”0.33” target=”s1”/>
  <transition prob=”0.5” target=”s2”.../>
  <transition target=”s3”.../>
</state>
```

Note that since the transitions are tried in document order, from top to bottom, it would *not* be correct to assign the probability 0.33 to each of them, if the intention was the above.

Our other suggestions deals with game AI. It should already be obvious that SCXML is more than powerful enough to implement conventional FSM-based game AI, but in this section we will show that simple extensions to SCXML would allow us to work with forward-chaining condition-action rules and decision trees as well, putting two other useful conventional game AI programming paradigms in the hands of the developer.

Our first suggestion is that the `cond` attribute of the <transition> element should accept a Prolog style query rather than an ordinary boolean expression, i.e. a query that evaluates to true of false (just like an ordinary boolean expression) but which will possibly also bind variables if evaluated to true. We suggest that the names of these variables be declared in a new attribute `vars`, and that the values of them are made available in the actions of the <transition>, as well as in the <onentry> element in the target state. A (possibly targetless) <transition> equipped with such a condition thus becomes a powerful forward chaining condition-action rule, and an ordered sequence of such transitions forms a so called production system, that can be used to implement the AI of a game. Here is a simple (non-game) example, implementing Euclid's algorithm for calculating the greatest common divisor of a set of numbers:

```
<scxml target="loop">
    <datamodel>
        <data name="S">[25 10 15 30]</data>
    </datamodel>
    <state id="loop">
        <transition vars="X Y"
                    cond="{Member S X}
                          {Member S Y}
                          X>:Y">
            <assign name="S" expr="{Del S X}"/>
            <assign name="S“ expr="{Add S X-Y}"/>
        </transition>
        <transition vars="X"
                    cond="{Member S X}"
                    target="stop">
            <log expr="'GCD = '#X"/>
        </transition>
    </state>
    <state id="stop" final="true"/>
</scxml>
```

Note that the strategy for selecting the next rule to fire is determined by the ordinary SCXML execution model – the first rule in document order with a condition evaluating to true will be executed. To avoid infinite loops, the programmer must ensure that the datamodel that satisfied the rule's conditions is actually changed by the actions of this rule. Alternatively, the next rule to be executed could be randomly selected, by means of the `prob` attribute – causing an unpredictability that could be just what we are looking for in a game.

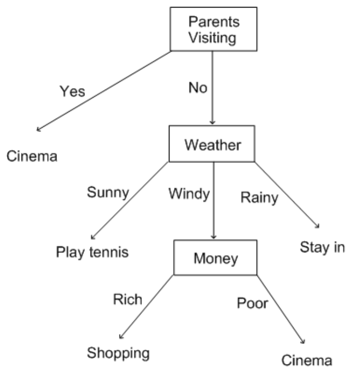

Figure 4: Decision tree inspired by SIMS

Our second suggestion is that the <transition> element should allow other <transition> elements as children, thus allowing a developer to encode decision trees directly in SCXML. For example, the (perhaps SIMS-relevant?) decision tree in Figure 4 could be rendered in SCXML in a way that clearly shows its structure, as follows:

```
<transition event="situation">
  <transition cond="Eventdata.visiting==yes"
              target="Cinema"/>
  <transition cond="Eventdata.visiting==no">
    <transition cond="Eventdata.weather==sunny"
                target="PlayTennis"/>
    <transition cond="Eventdata.weather==windy">
      <transition cond="Eventdata.money==rich"
                  target="Shopping"/>
      <transition cond="Eventdata.money==poor"
                  target="Cinema"/>
    </transition>
    <transition cond="Eventdata.weather==rainy"
                target="StayIn"/>
  </transition>
</transition>
```

Combined with the `prob` attribute, a probabilistic decision tree could be represented as well. The probabilities could be assigned using a machine learning method, and even be recalculated at runtime, allowing a form of adaptation to changing circumstances during a game to take place. A useful idiom might be to let one statechart monitor the activities of another statechart running in parallel to it, and adjust the values of its transition probabilities at runtime (this would work for delay times too).

5. IMPLEMENTATION AND MORE EXAMPLES

We have built one of the first implementations of SCXML (in Oz, using Oz as a scripting language). A web interface to a version of our software – called G-SCXML – is available at <http://www.ling.gu.se/~lager/Labs/G-SCXML-Lab/>. Visitors are able to try out a number of small examples related to gaming (do not expect any full games though) and are also able to write their own examples, either from scratch, or by modifying the ones given.

6. SUMMARY AND CONCLUSIONS

This paper has presented a large number of reasons for why SCXML should be a good fit for the gaming industry. We would like to summarize our arguments as follows: At its core, SCXML has FSMs – and it has been shown, over and over again, that FSMs are useful for describing the flow of a game, i.e. the pace and sequence of its states and events, and the range of choices in its progression. As a result of its Harel Statechart heritage, SCXML also supports hierarchy and concurrency, and thereby avoids the most pressing problem with ordinary FSMs – the notorious state explosion problem. The presence of hierarchy furthermore allows the developer to describe game flow at different levels of granularity, and to apply the methodological principles of top-down refinement and bottom-up clustering. In addition, the fact that SCXML is closely aligned to statechart theory and UML2 will help those using model driven development methodologies.

The support for concurrency furthermore allows a game developer to model NPCs and their environment as separate statecharts, as well as allowing him to structure the of NPC statecharts into 'mind modules', thereby allowing NPCs to do more than one thing at the time – listening, 'thinking' and talking, for example.

The fact that SCXML is endorsed by the W3C may translate to better support in tooling, number of implementations and various runtime environments. In particular SCXML and VoiceXML forms a powerful combination, where SCXML is used for specifying and implementing the flow aspect of a dialogue system, and VoiceXML supplies the voice widgets required. This enables an approach to the development of natural language enabled games where natural language dialogue flow is seen as just an aspect of the overall game flow, and where SCXML is used for specifying and implementing (the major parts) of both kinds of flow.

Finite-state machines, condition-action rules and decision trees are perhaps the most commonly used AI programming devices used in games today. We have shown that SCXML could be extended to handle also the latter two in an intuitive and straightforward way, but more work here is certainly needed.

We conclude by stating that we see an opportunity here, for (the relevant part of) the game industry and the spoken dialogue industry to share, not only (core) standards and software infrastructure, but to also a (future) work force skilled in the use of such standards and software. Since it would of course not be reasonable to expect the latter industry to look after the interest of the former, we believe that the game industry ought to 1) keep an eye on where the W3C is heading with SCXML, and 2) start asking itself whether SCXML might not be a suitable point of departure for an effort to create an XML-based standard more tailored to its own particular needs.

REFERENCES

Barnett, Jim et al. (2006) State Chart XML (SCXML): State Machine Notation for Control Abstraction, W3C Working Draft 24 January 2006 <http://www.w3.org/TR/scxml/>.

Bartolomeo, Dave (2003) Screaming at the Machine: Using Speech Recognition as a Complement to Traditional Game Input Technique, In: *Game Developers Conference Proceedings*, 2003.

Harel, David (1987) Statecharts: A Visual Formalism for Complex Systems, In: Science of Computer Programming 8, North-Holland.

Horrocks, Ian (1999) *Constructing the User Interface with Statecharts,* Addison Wesley.

Houlette, Ryan and Fu, Dan (2003) The Ultimate Guide to FSMs in Games, In: *AI Game Programming Wisdom 2*, 2003.

Samek, Miro (2002) *Practical Statecharts in C/C++*, CMP-Books.

Wallace, Richard (2005) Artificial Intelligence Markup Language (AIML) Version 1.0.1, Working Draft 2005 <http://docs.aitools.org/aiml/spec/WD-aiml>