Decalog 2007: Proceedings of the 11th Workshop on the Semantics and Pragmatics of Dialogue, pages 99–106.
Trento, Italy, 30 May – 1 June 2007. Edited by Ron Artstein and Laure Vieu.

Implementing the Information-State Update Approach to Dialogue Management in a Slightly Extended SCXML

Fredrik Kronlid
Department of Linguistics and GSLT
Göteborg University
S-405 30 Göteborg
kronlid@ling.gu.se

Torbjörn Lager
Department of Linguistics
Göteborg University
S-405 30 Göteborg
lager@ling.gu.se

Abstract

The W3C has selected Harel statecharts, under the name of SCXML, as the basis for future standards in the area of (multimodal) dialogue systems. The purpose of the present paper is to show that a moderately extended version of SCXML can be used to implement the well-known Information-State Update (ISU) approach to dialogue management. The paper also presents an experimental implementation of Extended SCXML, accessible from a user-friendly web-interface.

1 Introduction

The W3C has selected Harel statecharts (Harel, 1987), under the name of SCXML (Barnett et al., 2007), as the basis for future standards in the area of (multimodal) dialogue systems – replacing a simple and fairly uninteresting "theory of dialogue" (the form-based dialogue modelling approach of VoiceXML) with a theory neutral framework in which different approaches to dialogue modelling could potentially be implemented.[1]

One interesting and influential framework for dialogue management that has evolved over the past years is the so called Information-State Update (ISU) approach, based on the notion of an information state and its update via rules. The purpose of the present paper is to show that, if properly extended, SCXML can be used to implement the ISU approach to dialogue management.

[1]The present paper is based on the February 2007 SCXML working draft.

2 SCXML = State Chart XML

SCXML can be described as an attempt to render Harel statecharts in XML. In its simplest form, a statechart is just a deterministic finite automaton, where state transitions are triggered by events appearing in a global event queue.

Just like ordinary finite-state automata, statecharts have a graphical notation. Figure 1 depicts a very simple example.

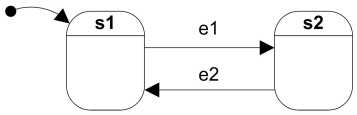

Figure 1: Simple statechart

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 statechart in Figure 1:

```
<scxml initialstate="s1">
  <state id="s1">
    <transition event="e1" target="s2"/>
  </state>
  <state id="s2">
    <transition event="e2" target="s1"/>
  </state>
</scxml>
```

The document can be executed by an SCXML conforming interpreter, an approach aiming at greatly simplifying the step from specification into running dialogue system application.

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

Hierarchy Statecharts may be hierarchical, i.e. a state may contain another statechart down to an arbitrary depth.

Concurrency Two or more statecharts may be run in parallel, which basically means that their parent statechart is in two or more states at the same time.

Broadcast communication One statechart S_1 may communicate with another statechart S_2 (running in parallel with S_1) by placing an event in the global event queue that triggers a transition in S_2.

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.

For our ISU implementations, we will find uses for all of these features, but will sometimes find is necessary to add a few novel ones as well. Fortunately, SCXML is designed with extensibility in mind (Barnett et al., 2007), and our own investigations suggest that there is indeed room for simple extensions that would increase the expressivity of SCXML even further.

3 The Information State Update Approach to Dialogue Modelling

Simplifying somewhat, the ISU approach to dialogue modelling can be characterized by the following components:

1. An *information state* representing aspects of common context as well as internal motivating factors
2. A set of *dialogue moves* that will trigger the update of the information state
3. A set of declaratively stated *update rules* governing the updating of the information state

The idea of information state update for dialogue modelling is centred around the information state (IS). Within the IS, the current state of the dialogue is explicitly represented. "The term Information State of a dialogue represents the information necessary to distinguish it from other dialogues, representing the cumulative additions from previous actions in the dialogue, and motivating future action" (Larsson and Traum, 2000).

Dialogue moves are meant to serve as an abstraction between the large number of different messages that can be sent (especially in natural language) and the types of updates to be made on the basis of performed utterances (Larsson and Traum, 2000, p. 5). Dialogue moves trigger non-monotonic updates of the IS. Thus, user utterances (or other kinds of user input) are matched against a set of possible update rules that change the IS in the appropriate places (e.g. a new value is entered into a slot). A single user utterance may unleash a whole chain of updates, allowing for generalisations beyond monolithic utterance updates.

The ISU approach should be seen as a rather abstract and relatively "empty" framework that needs to be filled with theoretical content to become a full-fledged theory of dialogue. For example, Larsson (2002) develops and implements a theory of Issue-Based Dialogue Management, taking Ginzburg's (1996) concept of Questions Under Discussion (QUD) as a starting point. QUD is used to model raising and addressing issues in dialogue (including the resolution of elliptical answers). Issues can also be raised by addressing them, e.g. by giving an answer to a question that has not be explicitly asked (question accommodation).

Two well-known implementations of the ISU approach to dialogue management are TrindiKit (Larsson and Traum, 2000) and DIPPER (Bos et al., 2003). Implemented/embedded in Prolog and relying to a large extent on properties of its host language, TrindiKit was the first implementation of the ISU approach. DIPPER is built on top of the Open Agent Architecture (OAA), supports many off-the-shelf components useful for spoken dialogue systems, and comes with a dialogue management component that borrows many of the core ideas of the TrindiKit, but is "stripped down to the essentials, uses a revised update language (independent of Prolog), and is more tightly integrated with OAA" (Bos et al., 2003). Other implementations exist, but TrindiKit and DIPPER are probably the most important ones.

4 Implementing ISU in SCXML

We suggest that most systems implementing the ISU approach to dialogue management can be reimplemented in (Extended) SCXML, exploiting the mapping between the ISU components and SCXML elements depicted in Table 1.

Of course, we cannot really prove this claim, but by taking a simple example system and reim-

The ISU Approach	SCXML
Information state	Datamodel
Dialogue move	Event
Update rule	Transition

Table 1: From ISU into Extended SCXML

plement it in SCXML we hope to be able to convince the reader of the viability of our approach. We choose to target the IBiS1 system from (Larsson, 2002), and thus most of our discussion will be comparing TrindiKit with SCXML, but we also hint at how DIPPER compares with SCXML. As we shall see, our conclusion is that SCXML could potentially replace them both.

4.1 Information states as datamodels

The expressivity of the SCXML `<datamodel>` is perfectly adequate for representing the required kind of information structures. A typical IBiS1 information state may for example be represented (and initialised) as follows:

```
<datamodel>
  <data name="IS">
    <private>
      <agenda>{New Stack init}</agenda>
      <plan>{New Stack init}</plan>
      <bel>{New Set init}</bel>
    </private>
    <shared>
      <com>{New Set init}</com>
      <qud>{New Stack init([q])}</qud>
      <lu>
        <speaker>usr</speaker>
        <move>ask(q)</move>
      </lu>
    </shared>
  </data>
</datamodel>
```

Here, the datamodel reflects the distinction between what is private to the agent that 'owns' the information state, and what is shared between the agents engaged in conversation. Note that `IS.shared.qud` points to a stack with `q` on top, indicating that it is known by both parties that the question `q` is "under discussion".[2]

4.2 Dialogue moves as SCXML events

The closest SCXML correlate to a dialogue move is the notion of an *event.* An SCXML event has a *name*, and an optional *data payload*. The (current) SCXML draft does not represent events formally, but for the purpose of the present paper we will represent them as records with a label (for representing their name) and a set of feature-value pairs (for representing the data payload). An ASK move where a speaker `a` is asking a question `q` may thus be represented as: `says(speaker:a move:ask(q))`

4.3 Update rules as transitions

A TrindiKit ISU-style update rule consists of a set of *applicability conditions* and a set of *effects* (Larsson and Traum, 2000, p. 5), and a collection of such rules forms what is essentially a system of condition-action rules – a *production system*. While SCXML is easily powerful enough to implement such a system, the expressivity of the language for stating the conditions is not adequate for our purpose, since there is no mechanism in place for carrying information (i.e. information dug up from the IS) from the conditions over to the actions. This is where we are suggesting a small extension. We propose that a `pcond` attribute be added to the `<transition>` element, the value of which is 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>`.

For example, an update rule written in the following way in the Prolog-based TrindiKit notation

```
rule( integrateSysAsk,
    [ $/shared/lu/speaker = sys,
      $/shared/lu/move = ask(Q)],
    [ push( /shared/qud, Q ) ] ).
```

may be written as follows in Extended SCXML:

```
<transition vars="Q"
            pcond="IS.shared.lu.speaker=sys
                   IS.shared.lu.move=ask(Q)"
            target="downdateQUD">
  <script>{IS.shared.qud push(Q)}</script>
</transition>
```

4.4 The update algorithm as a statechart

Dialogue management involves more than one rule, and the application of the rules needs to be controlled, so that the right rules are tried and applied at the right stage in the processing of a dialogue. Furthermore, we require three *kinds* of rules: 1) rules that perform unconditional maintenance operations on the datamodel (representing

[2] We use `q` and `r` here as placeholders for a question and a response, respectively.

the information state), 2) rules that enable events (representing dialogue moves) to update the datamodel, and 3) rules that when triggered by certain configurations of the datamodel updates it, i.e. changes its configuration. (The above example is of the third kind.)

Here is an example of the first kind of rule, responsible for first clearing the agenda, and then transferring to the `grounding` state:

```
<state id="init">
  <transition target="grounding">
    <script>
      {IS.private.agenda clear}
    </script>
  </transition>
</state>
```

(We shall return to the significance of the enclosing state further down.) For an example of the second kind of rule we offer:

```
<state id="grounding">
  <transition event="says"
              target="integrate">
    <assign location="IS.shared.lu.move"
            expr="Eventdata.move"/>
    <assign location="IS.shared.lu.speaker"
            expr="Eventdata.speaker"/>
  </transition>
</state>
```

This rule provides a bridge between the events representing dialogue moves and the datamodel representing the IS. If an event of the form `says(speaker:sys move:answer(r))` appears first in the event queue when the statechart is in state `grounding`, the rule will set `IS.shared.lu.move` to the value `answer(r)` and `IS.shared.lu.speaker` to `sys`, and then a transfer to the state `integrate` will take place. In this state, three transitions representing update rules of the third kind are available:

```
<state id="integrate">
  <transition vars="Q"
              pcond="IS.shared.lu.speaker=sys
                     IS.shared.lu.move=ask(Q)"
              target="downdateQUD">
    <script>{IS.shared.qud push(Q)}</script>
  </transition>
  <transition vars="Q"
              pcond="IS.shared.lu.speaker=usr
                     IS.shared.lu.move=ask(Q)"
              target="downdateQUD">
    <script>
      {IS.shared.qud push(Q)}
      {IS.private.agenda push(respond(Q))}
    </script>
  </transition>
  <transition vars="Q R"
              pcond="IS.shared.lu.move=answer(R)
                     {IS.shared.qud top(Q)}
                     {Domain.relevantAnswer Q R}"
              target="downdateQUD">
    <script>{IS.shared.com add(Q#R)}</script>
  </transition>
</state>
```

The transitions are tried in document order and given the current datamodel the last one will be the one chosen for execution. Its effect is that `q#r` (i.e. the pair of `q` and `r`, representing a proposition) will be added to the set at `IS.shared.com` i.e. the set of beliefs that the user and system shares (or "the common ground"). Thereafter a transition to the state `downdateQUD` will take place:

```
<state id="downdateQUD">
  <transition vars="Q R"
              pcond="{IS.shared.qud top(Q)}
                     {Domain.relevantAnswer Q R}
                     {IS.shared.com member(Q#R)}"
              target="load_plan">
    <script>{IS.shared.qud pop}</script>
  </transition>
  <transition target="load_plan"/>
</state>
```

In this state, either the first of its transitions will trigger, first popping the QUD and then leading to the `load_plan` state, or else the second transition will trigger, also leading to `load_plan`, but this time without popping the QUD. That is, the state will *try* to downdate the QUD. Given the current configuration of the datamodel in our example, the first rule will trigger, the element on top of the stack at `IS.shared.qud` will be popped, and (the relevant part of) the datamodel end up as follows:[3]

```
<datamodel>
  <data name="IS">
    <private>
      <agenda>[]</agenda>
      <plan>[]</plan>
      <bel>{}</bel>
    </private>
    <shared>
      <com>{q#r}</com>
      <qud>[]</qud>
      <lu>
        <speaker>sys</speaker>
        <move>answer(r)</move>
      </lu>
    </shared>
  </data>
</datamodel>
```

Note how the underlying DFA 'backbone' controls when certain classes of rules are eligible for execution. In statechart notation, the relevant statechart can be depicted as in Figure 2.[4] By comparison, in TrindiKit the control of the application of update rules is handled by an *update algorithm* written in a procedural language designed for this purpose.

[3] Here, [] and {} indicate the empty stack and the empty set, respectively.

[4] The details of the `load_plan` and `exec_plan` states may be found in our web-based demo.

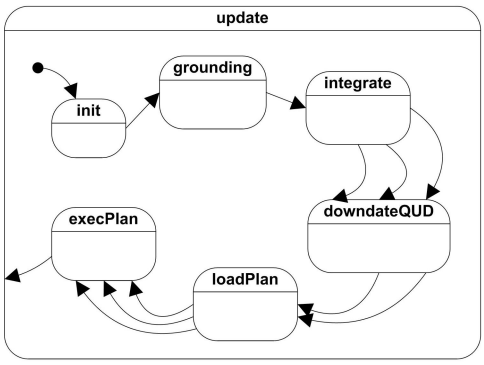

Figure 2: Update statechart

The update algorithm (or a version of it) used by IBiS1 is shown here:

```
if not LATEST_MOVES == failed
then ( init,
       grounding,
       integrate,
       try downdate_qud,
       try load_plan,
       repeat exec_plan )
```

Note that the statechart in Figure 2 does basically the job of this algorithm. Terms like "init", "grounding", "integrate", "downdate_qud", etc. refer to TrindiKit *rule classes*. In our statechart, they correspond to states.

4.5 Implementing modules as statecharts

The update statechart in Figure 2 basically corresponds to the *update module* in IBiS1, responsible for updating the information state based on observed dialogue moves. There is also a *select module* in IBiS1, responsible for selecting moves to be performed, which space does not allow us to go into detail about here (but see our web-based demo).

Together, the update module and the select module forms the *Dialogue Move Engine* (DME) – the dialogue manager proper. As can be seen in Figure 3, DME processing starts in the select state and then alternates between update and select.

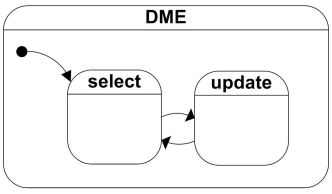

Figure 3: The Dialogue Move Engine

4.6 Interpretation and generation

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 TTS component for presenting the user with spoken information and an ASR component to collect spoken information from the user.

For example, our interpretation module may invoke an ASR component, like so:[5]

```
<state id="interpret">
  <invoke targettype="vxml"
          src="grammar.vxml#main"/>
</state>
```

and our generation module may invoke a TTS component as follows:

```
<state id="generate">
  <invoke targettype="vxml"
          src="generate.vxml#prompt"/>
</state>
```

4.7 The dialogue system statechart

The TrindiKit architecture also features a *controller*, wiring together the other modules necessary for assembling a complete dialogue system, either in sequence or through some asynchronous (i.e. concurrent) mechanism (Larsson, 2002). We choose here to exemplify an asynchronous architecture, taking advantage of the concurrency offered by SCXML. The statechart corresponding to a full dialogue system might look like in Figure 4.

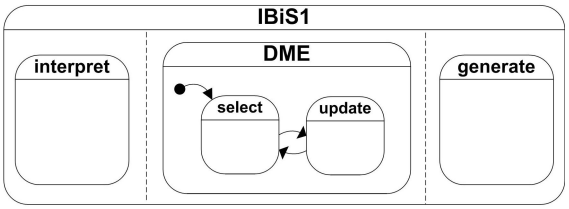

Figure 4: Parallel control

The dashed lines show – using standard statechart graphical notation – that the interpretation mod-

[5]Here we use VoiceXML for our example presentation components. This is not optimal, but we take comfort in the fact that the next major version of VoiceXML (known as V3) will be redesigned from the bottom and up with uses like these in mind.

ule, the DME and the generation module are run in parallel. In SCXML the full dialogue system may be sketched as follows:

```
<parallel id="IBiS1">
   <state id="interpret" .../>
   <state id="DME">
     <initial>
       <transition target="select"/>
     </initial>
     <state id="select" .../>
     <state id="update" .../>
   </state>
   <state id="generate" .../>
</parallel>
```

Communication between the modules of the system – between the interpreter, generator and DME – is performed in the broadcast style supported by SCXML, by letting one module place events in the global event queue – events to be picked up by another module. Comparing SCXML and TrindiKit, we note that the SCXML notion of an event queue seems to do the job of TrindiKit's *module interface variables* (MIVs), which is exactly this – to enable modules to interact with each other.

4.8 From TrindiKit to SCXML: a summary

In Table 2, we summarize the relevant correspondences between TrindiKit and our SCXML formalization of the ISU approach to dialogue management.

TrindiKit	SCXML
Information state	Datamodel
Dialogue move	Event
Module interface vars	Event queue
Update rule	Transition
Rule class	State (simple)
Update algorithm	State (complex)
Module	State (complex)
Control algorithm	State (complex)

Table 2: From TrindiKit into Extended SCXML

We note that SCXML is considerably more simple than TrindiKit, in that rule classes, update algorithms, modules and control algorithms are all represented as (simple or complex) states/statecharts.

4.9 From DIPPER to SCXML

(Bos et al., 2003) illustrate the DIPPER architecture and information state update language with an example which implements a "parrot", where the system simply repeats what the user says. These are the information state and the relevant update rules, in DIPPER notation:

```
is:record([input:queue(basic),
           listening:basic,
           output:queue(basic)]).

urule(timeout,
      [first(is^input)=timeout],
      [dequeue(is^input)]).

urule(process,
      [non_empty(is^input)],
      [enqueue(is^output,first(is^input)),
       dequeue(is^input)]).

urule(synthesise,
      [non_empty(is^output)],
      [solve(text2speech(first(is^output)),[]),
       dequeue(is^output)]).

urule(recognise,
      [is^listening=no],
      [solve(X,recognise('.Simple',10),
             [enqueue(is^input,X),
              assign(is^listening,no)]),
       assign(is^listening,yes)]).
```

Here is our translation into SCXML:

```
<scxml initialstate="process">
  <datamodel>
    <data name="IS">
      <input>{New Queue init}</input>
      <output>{New Queue init}</output>
    </data>
  </datamodel>
  <state id="process">
    <transition cond="{IS.input first($)}==timeout">
      <script>
        {IS.input dequeue}
      </script>
    </transition>
    <transition cond="{Not {IS.input isEmpty($)}}">
      <script>
        {IS.output enqueue({IS.input first($)})}
        {IS.input dequeue}
      </script>
    </transition>
    <transition cond="{Not {IS.output isEmpty($)}}">
      <send event="speak"
            expr="{IS.output first($)}"/>
      <script>
        {IS.output dequeue}
      </script>
    </transition>
    <transition target="listening"/>
  </state>
  <state id="listening">
    <onentry>
      <send event="recognise"/>
    </onentry>
    <transition event="recResult" target="process">
      <script>
        {IS.input enqueue(Eventdata)}
      </script>
    </transition>
  </state>
</scxml>
```

We shall use this example as our point of departure when comparing DIPPER, SCXML and TrindiKit. First, we note that DIPPER uses the *solveables* of OAA for the purpose of enabling modules to interact with each other. In the case of the fourth rule above, a solvable is sent to the OAA agent responsible for speech recognition, which within 10 seconds will bind the variable `X` to either the recognition result or to the atom `timeout`. This value of `X` will then be added to the input queue. Our SCXML version works in a similar fashion. An event `recognise` is sent in order to activate

the speech recognition module, and a transition is triggered by the `recResult` event returned by this module. The `Eventdata` variable will be bound to the recognition result.

Secondly, in the DIPPER rule set, an information state field 'listening' is used (as we see it) to simulate a finite state automaton with two states `listening=yes` and `listening=no`. The idea is to control the application of the fourth rule – it is meant to be applicable only in the 'state' `listening=no`. The general strategy here appears to be to take advantage of the fact that a production system can easily simulate a finite state automaton. DIPPER can thus eliminate the need for an update algorithm in the style of TrindiKit, but at the expense of complicating the rules.

Note that the 'listening' field is not required in the SCXML version, since we can use two "real" states instead. Indeed, looking at TrindiKit, DIPPER and SCXML side by side, comparing TrindiKit's use of an update algorithm, DIPPER's DFA simulation 'trick', and SCXML's use of real states, we think that SCXML provides the neatest and most intuitive solution to the problem of controlling the application of update rules.

Finally, few (if any) extensions of SCXML appear to be needed in order to reconstruct DIPPER style dialogue managers in SCXML. This is mainly due to the fact that DIPPER does not make use of Prolog style conditions the way TrindiKit does. Whether the availability of Prolog style conditions in this context is crucial or not is, in our opinion, still an open question.

5 A More Abstract Point of View

In a recent and very interesting paper Fernándes and Endriss (2007) present an hierarchy of abstract models for dialogue protocols that takes as a starting point protocols based on deterministic finite automata (DFAs) and enhances them by adding a 'memory' in the form of an instance of an abstract datatype (ADT) such as a stack, a set or a list to the model. They show that whereas a DFA alone can handle only simple dialogue protocols and conversational games, a DFA plus a set can handle also for example the representation of a set of beliefs forming the common ground in a dialogue, a DFA plus a stack is required if we want to account for embedded subdialogues, questions under discussion á la Ginzburg, etc., and a DFA plus a list is needed to maintain an explicit representation of dialogue history.

Space does not allow us to give full justice to the paper by Fernándes and Endriss here. We only wish to make the point that since an SCXML state machine at its core can be seen as just a fancy form of a DFA, and since SCXML does indeed allow us to populate the datamodel with instances of ADTs such as stacks, sets and list, it seems like SCXML can be regarded as a concrete realization very "true to the spirit" of the more abstract view put forward in the paper (and more true to this spirit than TrindiKit or DIPPER). Having said this, we hasten to add that while we think that the DFA core of SCXML is well-designed and almost ready for release, the datamodel definitely needs more work, and more standardization.

6 An SCXML Implementation

We have built one of the first implementations of SCXML (in the Oz programming language, using Oz as a scripting language). A web interface to a version of our software – called Synergy SCXML – is available at <www.ling.gu.se/~lager/Labs/SCXML-Lab/>. Visitors are able to try out a number of small examples (including a full version of the SCXML-IBiS1 version described in the present paper) and are also able to write their own examples, either from scratch, or by modifying the given ones.[6]

7 Summary and Conclusions

We summarize by highlighting what we think are the strong points of SCXML. It is:

- **Intuitive**. Statecharts and thus SCXML are based on the very intuitive yet highly abstract notions of *states* and *events.*
- **Expressive**. It is reasonable to view SCXML as a multi-paradigm programming language, built around a declarative DFA core, and extended to handle also imperative, event-based and concurrent programming.
- **Extensible**. SCXML is designed with extensibility in mind (Barnett et al., 2007), and our own investigations suggest that there is indeed room for simple extensions that will

[6] Our implementation is not the only one. *Commons* SCXML is an implementation aimed at creating and maintaining an open-source Java SCXML engine, available from <http://jakarta.apache.org/commons/scxml/>. There are most likely other implementations in the works.

increase the expressivity of SCXML considerably.

- **Theory neutral**. Although it is clear that the framework is suitable for implementing both simple DFA-based as well as form-based dialogue management, the framework as such is fairly theory neutral.

- **Visual**. Just like ordinary finite-state automata, 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).

- **Methodologically sound.** The importance of support for refinement and clustering should not be underestimated. In addition, the fact that SCXML is closely aligned to statechart theory and UML2 will help those using model driven development methodologies.

- **XML enabled**. Thus, documents may be validated with respect to a DTD or an XML Schema, and there are plenty of powerful and user friendly editors to support the authoring of such documents.

- **Part of a bigger picture**. SCXML is designed to be part of a framework not just 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.

- **Endorsed by the W3C**. The fact that SCXML is endorsed by the W3C may translate to better support in tooling, number of implementations and various runtime environments.

We conclude by noting that despite the fact that SCXML was not (as far as we know) designed for the purpose of implementing the ISU approach to dialogue management, it is nevertheless possible to do that, in the style of TrindiKit (provided the proposed rather moderate extensions are made) or in the style of DIPPER. Indeed, we believe that SCXML could potentially replace both TrindiKit and DIPPER.

All in all, this should be good news for academic researchers in the field, as well as for the industry. Good news for researchers since they will get access to an infrastructure of plug-and-play platforms and modules once such platforms and modules have been built (assuming they *will* be built), good news for the industry since a lot of academic research suddenly becomes very relevant, and good news for the field as a whole since SCXML appears to be able to help bridging the gap between academia and industry.

8 Acknowledgements

A preliminary version of this paper has been presented in a seminar at the Centre for Language Technology (CLT) in Göteborg. We are grateful to the members of this seminar for reading and discussing the paper, and for proposing useful additions and improvements.

References

Jim Barnett et al. 2006. State Chart XML (SCXML): State Machine Notation for Control Abstraction. http://www.w3.org/TR/2007/WD-scxml-20070221/.

Johan Bos, Ewan Klein, Oliver Lemon and Tetsushi Oka. 2003. DIPPER: Description and Formalisation of an Information-State Update Dialogue System Architecture. In *4th SIGdial Workshop on Discourse and Dialogue*, ACL, Sapporo.

Raquel Fernández and Ulle Endriss. 2007. Abstract Models for Dialogue Protocols. In *Journal of Logic, Language and Information* vol. 16, no. 2, pp. 121-140, 2007.

Jonathan Ginzburg. 1996. Interrogatives: Questions, Facts and Dialogue. In S. Lappin (ed.): *Handbook of Contemporary Semantic Theory*, Blackwell.

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

Staffan Larsson and David Traum. 2000. Information state and dialogue management in the TRINDI Dialogue Move Engine Toolkit. In *Natural Language Engineering*, vol. 6, no. 3-4, pp. 323-340, 2000.

Staffan Larsson. 2002. *Issue-Based Dialogue Management*. Ph.D. thesis, Göteborg University.

Miro Samek. 2002. *Practical Statecharts in C/C++*. CMPBooks.