A Prolog Datamodel for State Chart XML

Stefan Radomski
TU Darmstadt
Telekooperation Group
Hochschulstrasse 10
radomski@tk.informatik
.tu-darmstadt.de

Dirk Schnelle-Walka
TU Darmstadt
Telekooperation Group
Hochschulstrasse 10
dirk@tk.informatik
.tu-darmstadt.de

Stephan Radeck-Arneth
TU Darmstadt
Telekooperation Group
Hochschulstrasse 10
arneth@rbg.informatik
.tu-darmstadt.de

Abstract

SCXML was proposed as one description language for dialog control in the W3C Multimodal Architecture but lacks the facilities required for grounding and reasoning. This prohibits the application of many dialog modeling techniques for multimodal applications following this W3C standard. By extending SCXML with a Prolog datamodel and scripting language, we enable those techniques to be employed again. Thereby bridging the gap between respective dialog modeling research and a standardized architecture to access and coordinate modalities.

1 Introduction

Deploying multimodal applications has long been an activity of custom solutions, each with their own access to modalities, approaches to sensor fusion and fission and techniques for dialog modeling. With the advent of the W3C MMI architecture (Bondell et al., 2012), the W3C proposed a standardized approach to ensure interoperability among its constituting components (Schnelle-Walka et al., 2013; Dahl, 2013).

The architecture proposed by the W3C decomposes a multimodal application into a nested structure of *interaction managers* for dialog control and *modality components* for in- and output. An application is conceived as a set of control documents expressed in SCXML (Barnett et al., 2012) or CCXML (Auburn et al., 2011) for the interaction managers and a set of presentation documents with modality-specific markup for the modality components. A topmost root controller document describes the global dialog and instantiates modality components as required. Each modality component can, in turn, again be an interaction manager, handling more fine granular concerns of dialog control, such as error correction or even sensor fusion/fission.

As one proposed XML dialect for control documents, State Chart XML (SCXML) is given the responsibility to model an applications dialog behavior. SCXML as such is a markup language to express Harel state charts (Harel and Politi, 1998) with nested and parallel machine configurations. The transitions between configurations are triggered by events delivered into the interpreter either from external components or raised by the interpreter itself. Whenever an event arrives, the SCXML interpreter can perform actions described as *executable content*. This includes invoking or sending events to external components, processing data or updating the datamodel via an embedded scripting language.

SCXML has been proven to be suitable to decouple the control flow and presentation layer in dialog management (Wilcock, 2007). It has been used in several applications to express dialog states (Brusk et al., 2007) or to easily incorporate external information (Sigüenza Izquierdo et al., 2011). However, SCXML seems to be suited *only* to implement finite state or frame-based/form-filling dialogue management approaches. Applications using theses dialog techniques are oftentimes inflexible as they lack grounding and reasoning. In this regard, Fodor and Huerta (2006) demand that dialog managers should feature: (i) a formal logic foundation, (ii) an interference engine, (iii) general purpose planners and (iv) knowledge representation and expressiveness.

Most of these requirements are addressed by employing Prolog. Embedding it as a scripting language into SCXML allows multimodal applications in the W3C MMI Architecture to employ the more elaborate dialog management techniques, resulting in more natural and flexible interaction. In this paper we describe our integration of Prolog as an embedded scripting language in an SCXML

Proceedings of the SIGDIAL 2013 Conference, pages 127–131,
Metz, France, 22-24 August 2013.

datamodel. All of the work described here is implemented as part of our uSCXML interpreter[1] by embedding the SWI Prolog implementation.

2 The Prolog Datamodel

Datamodels in SCXML are more than simple repositories for storing data. With the exception of the `null` datamodel, they provide access to embedded scripting languages. The datamodels already specified by the SCXML draft are the `null`, the `xpath` and the `ecmascript` datamodel. Prolog itself is a declarative language for logic programming in which facts and rules are used to answer queries. The result of a query is either a boolean value or the set of valid assignments for the queries variables.

In the following sections, we will describe our integration of Prolog as a datamodel in SCXML. The structure of the description loosely follows the existing descriptions for datamodels already found in the SCXML draft.

2.1 Assignments

In an SCXML document, there are two elements which will assign values to variables in the datamodel. These are `<data>` for initial assignments and `<assign>` itself. In Prolog, variable assignment is only available in the scope of a query. To realize variable assignment nevertheless, we introduce the variables as predicates, with their assigned data as facts. Listing 1 exemplifies some assignments followed by their resulting Prolog facts.

```
<data id="father">
  bob, jim.
  bob, john.
</data>
% father(bob, jim).
% father(bob, john).

<data id="">
  mother(martha, jim).
  mother(martha, john).
</data>
% mother(martha, jim).
% mother(martha, john).

<assign location="">
  retract(father(bob, jim)).
  assert(father(steve, jim)).
</assign>
% father(bob, john).
% father(steve, jim).

<data id="childs">
  <child name="john" father="bob" />
  <child name="jim" father="bob" />
</data>
% childs([
%  element(child,
%    [father=bob, name=john], []),
%  element(child,
%    [father=bob, name=jim], [])]).

<data id="household">
  {
    name: "The Bobsons",
    members: ['bob', 'martha', 'jim', 'john']
  }
</data>
% household({
%   name:'The Bobsons',
%   members:[bob, martha, jim, john]}).
```

Listing 1: Assignments and their results in Prolog.

If given, the `id` or `location` attribute identifies the predicate for which the content is to be asserted as fact, otherwise the content is assumed to be a dot-separated list of prolog queries or expressions. The content might also be loaded per URL in the element's `src` attribute. In the context of SCXML, it is important to support XML and JSON data as shown in the last two examples. Not only enables this an application developer to load data from existing XML and JSON files, it is also important to support these representations for incoming events as we will see in the next section.

There is no standardized representation for XML DOMs or JSON data in Prolog. We pragmatically settled upon the structure returned by the SWI-Prolog SGML parser and the JSON converter as de-facto standards respectively.

With the Prolog datamodel, having an `id` or `location` attribute at assignment elements seems superfluous. We do keep them as the SCXML draft specifies these as required attributes.

2.2 Structure of Events

Whenever an event is received by the SCXML interpreter, it has to be transformed into a suitable representation in order to operate on its various fields and content as defined by the SCXML draft. We choose to represent an event as the single predicate `event/1` with its facts as compound terms reflecting the event's fields as shown in listing 2.

```
event(name('foo')).
event(type('external')).
event(sendid('s1.bar')).
event(origin('http://host/path/basichttp')).
event(origintype('http://www.w3.org/TR/scxml
    /#BasicHTTPEventProcessor')).
event(invokeid('')).
event(data(...)).
event(param(...)).
event(raw(...)).
```

Listing 2: Example facts for event/1.

This representation enables access to the events individual fields by simple queries such as `event(name(X))`, which will resolve X to the

[1] `https://github.com/tklab-tud/uscxml`

event's name `foo`. Whenever the interpreter is about to process a new event, all old facts about `event/1` are retracted and reasserted with regard to the new event.

The event's data field may contain a space normalized string as an atomic term, an XML DOM or, optionally, data from a JSON structure. The structure of JSON and XML DOMs is the same as with assignments in listing 1.

2.3 Scripting

The `<script>` element either contains Prolog expressions as they would be written in a Prolog file or references such a file directly via its `src` attribute. Together with `<assign>` and `<data>`, this element is the third available to load Prolog files into the SCXML interpreter. This is somewhat undesirable and we would propose to use (i) `<data>` to establish initial a-priori knowledge as facts, (ii) `<assign>` for subsequent changes and additions to facts and (iii) `<script>` to introduce new rules or load Prolog files containing primarily rules.

It is important to note that we do provide a full ISO-Prolog implementation at runtime. This enables an application developer to load arbitrary Prolog files with all their facts and rules.

2.4 System Variables

The SCXML draft requires the datamodel to expose various platform specific values to the datamodel. These are the identifier of the current session, the name of the document and the available I/O processors to send and receive events. Following the approach of defining predicates to provide access to information in the datamodel, we introduced predicates as given in listing 3.

```
% name/1:
name("foo").

% sessionid/1:
sessionid("bar").

% ioprocessors/1:
ioprocessors(
  basichttp(
    location('http://host/path/basichttp'))).
ioprocessors(
  scxml(location('http://host/path/scxml'))).

% ioprocessors/2:
ioprocessors(
  name(basichttp),
  location('http://host/path1')).
ioprocessors(
  name('http://www.w3.org/TR/scxml/#
      BasicHTTPEventProcessor'),
  location('http://host/path1')).
...
```

Listing 3: Predicates for system variables.

Defining two predicates for ioprocessors is simply a matter of convenience as their short names (e.g. basichttp or scxml) are suited as functors for compound terms, where their canonic names are not. Therefore `ioprocessors/1` will only contain the short names, and `ioprocessors/2` contains both. This allows us to send events, e.g with the basichttp ioprocessor via:

```
<send type="basichttp"
      targetexpr="ioprocessors(basichttp(
          location(X)))"
      event="foo">
```

Listing 4: Sending ourself an event via basichttp.

2.5 Conditional Expressions

Conditional expressions in SCXML are used to guard transitions and as part of `<if>` and `<elseif>` elements in executable content. They consist of a single, datamodel specific expression that ought to evaluate to a boolean value. In the case of our Prolog datamodel, these expressions can take the form of an arbitrary query (see listing 5). If there exists at least one solution to the query, the conditional expression will be evaluated to *true*, and *false* otherwise.

```
% Is there someone who is not the father
% of Jim and older than bob?
<if cond="not(father(X, jim)),
          older(X, bob).">

% Was the current event received from an
% external component?
<transition
  target="s3"
  cond="event(type(X)), X='external'"/>

% Does the JSON structure in the event's
% data contain a  household whose name
% is 'The Bobsons'?
<transition
  target="s5"
  cond="event(data(household(name:X))),
        X='The Bobsons'"/>
```

Listing 5: Boolean expressions in `cond` attribute.

2.6 Evaluating as String

There are several situations in the SCXML draft, where an element from the datamodel needs to be represented as a string. These are usually attributes of elements that equal or end in `expr`, e.g. `log.expr` or `send.targetexpr`.

In these contexts, the interpreter will allow for queries with a single free variable that has to resolve to an atomic term. The actual value of the expression is then the string representation of the variable from the last solution to the query (see listing 6).

```
% This query only has a single solution
<log label="Event Name"
     expr="event(name(X))" />

% This query has multiple solutions, only the
% last is used when evaluating as string
<log label="Bob's youngest son"
     expr="father(bob, X)" />
```

Listing 6: Evaluating an expression as string.

2.7 Foreach

The `<foreach>` element in SCXML allows to iterate over values as part of executable content. Its attributes are `array` as an iterable expression, `item` as the current element in the array and `index` as the current iteration index.

In our Prolog datamodel, this element is available to iterate over all solutions of a query as shown in listing 7.

```
<foreach array="father(bob, X)"
         item="child"
         index="index">
  <log label="child" expr="child(X)" />
  <log label="index" expr="index(X)" />
</foreach>

% results in the following log output
child: jim
index: 0
child: john
index: 1
child: jack
index: 2
```

Listing 7: Foreach expressions.

3 Example

Listing 8 exemplifies some of the language features of the Prolog datamodel. We start by introducing two predicates with the `<data>` element, the first defined as dot seperated facts, the second one as inline Prolog expressions. In the first state `s1`, we iterate all children of bob and log their names. Transitioning to the next state is performed if bob and martha have a common child. In `s2`, we send ourself an event containing a XML snippet using the basichttp I/O processor. Then we transition to the final state if there is an element with a tagname of `p` in the received XML document. In the final state we print all facts we established via Prolog's `listing/1` predicate and the interpreter stops.

```
<scxml datamodel="prolog">
  <datamodel>
    <data id="father">
      bob, jim.
      bob, john.
    </data>
    <data id="">
      mother(martha, jim).
      mother(martha, john).
    </data>
  </datamodel>
  <state id="s1">
    <onentry>
      <foreach array="father(bob, X)"
               item="child"
               index="index">
        <log label="index" expr="index(X)" />
        <log label="child" expr="child(X)" />
      </foreach>
    </onentry>
    <transition target="s2"
                cond="mother(martha, X),
                    father(bob, X)"/>
  </state>
  <state id="s2">
    <onentry>
      <send type="basichttp"
            targetexpr="ioprocessors(
                basichttp(location(X)))"
            event="foo">
        <content>
          <p>Snippet of XML</p>
        </content>
      </send>
    </onentry>
    <transition
      cond="member(element('p',_,_), X),
          event(data(X))" />
  </state>
  <state final="true">
    <log label="Listing" expr="listing." />
  </state>
</scxml>
```

Listing 8: Example SCXML document.

4 Conclusion

Providing a Prolog datamodel for SCXML enables applications in the W3C MMI architecture to employ grounding and reasoning for facts established during a prior to a dialog. It even enables developers to load complete, existing Prolog programs to to be used during event processing. This extends SCXML to fulfill the requirements for dialog management as defined by Fodor and Huerta (2006).

There are multiple variations to the integration of Prolog and more experience is needed still to determine whether the approach presented here is optimal.

References

RJ Auburn, Paolo Baggia, and Mark Scott. 2011. Voice browser call control (CCXML). W3C recommendation, W3C, July. http://www.w3.org/TR/2011/REC-ccxml-20110705/.

Jim Barnett, Rahul Akolkar, RJ Auburn, Michael Bodell, Daniel C. Burnett, Jerry Carter, Scott McGlashan, Torbjörn Lager, Mark Helbing, Rafah Hosn, T.V. Raman, Klaus Reifenrath, and No'am Rosenthal. 2012. State chart XML (SCXML): State machine notation for control abstraction. W3C working draft, W3C, February. http://www.w3.org/TR/2012/WD-scxml-20120216/.

Michale Bondell, Deborah Dahl, Ingmar Kliche, Jim Larson, Brad Porter, Dave Raggett, T.V. Raman, Bertha Helena Rodriguez, Muthuselvam Selvari, Raj Tumuluri, Andrew Wahbe, Piotr Wiechno, and Moshe Yudkowsky. 2012. Multimodal Architecture and Interfaces. W3C recommendation, W3C, October. http://www.w3.org/TR/2012/REC-mmi-arch-20121025/.

Jenny Brusk, Torbjörn Lager, Anna Hjalmarsson, and Preben Wik. 2007. Deal: dialogue management in scxml for believable game characters. In *Proceedings of the 2007 conference on Future Play*, pages 137–144. ACM.

Deboraha Dahl. 2013. The w3c multimodal architecture and interfaces standard. *Journal on Multimodal User Interfaces*, pages 1–12.

Paul Fodor and Juan M Huerta. 2006. Planning and logic programming for dialog management. In *Spoken Language Technology Workshop, 2006. IEEE*, pages 214–217. IEEE.

David Harel and Michal Politi. 1998. *Modeling Reactive Systems with Statecharts: The Statemate Approach*. McGraw-Hill, Inc., August.

Dirk Schnelle-Walka, Stefan Radomski, and Max Mühlhäuser. 2013. Jvoicexml as a modality component in the w3c multimodal architecture. *Journal on Multimodal User Interfaces*, pages 1–12.

Álvaro Sigüenza Izquierdo, José Luis Blanco Murillo, Jesús Bernat Vercher, and Luis Alfonso Hernández Gómez. 2011. Using scxml to integrate semantic sensor information into context-aware user interfaces.

Graham Wilcock. 2007. SCXML and voice interfaces. In *3rd Baltic Conference on Human Language Technologies, Kaunas, Lithuania*. Citeseer.