Chapter 12

Web Prolog and State Chart XML

> Content is king. — *Bill Gates*
> Conversation is king. Content is just something to talk about. — *Cory Doctorow*

When proposing a new web technology such as Web Prolog, it is important to show how it relates to already existing technologies and standards for the Web. As we have suggested in Chapter 10, since Web Prolog works really well with RDF and friends, we may want to think of it as a Semantic Web programming language. In this chapter, we shall argue that in combination with another web standard called State Chart XML, Web Prolog can also be seen as a language for programming what might be referred to as the *Conversational* Web.

State Chart XML (SCXML) is based on STATECHARTS. Invented by David Harel and described in (Harel, 1987), STATECHARTS is a graphical notation for specifying reactive systems in great detail, and is, in Harel's own words, a "fully executable visual formalism intended for capturing the behaviour of complex real-world systems". On top of the concept of finite-state automata, the STATECHARTS formalism introduces hierarchy, parallelism (a.k.a. orthogonality), a notion of broadcast communication, and more. STATECHARTS, just like formal logic, is a tool for thinking which comes with a precise semantics (Harel and Naamad, 1996). But STATECHARTS, in contrast to logic, is a *visual* tool – a tool for "tapping the potential of high band-width spatial intelligence, as opposed to lexical intelligence used with textual information" (Samek, 2002).

STATECHARTS have found many uses in, for instance, embedded systems (Samek, 2002), user interface design (Horrocks, 1999), conversational systems (Skantze and Moubayed, 2012), game design (Brusk and Lager, 2008), simulation (Cicirelli et al., 2009) and protocol design. (Recall, for example, that a statechart was used in the specification of the PCP protocol in Section 6.1.) A variant of the statechart notation also appears as a tool for model-based software development in the context of the Unified Modelling Language (UML).

In the context of the Web, the World Wide Web Consortium (W3C) has proposed SCXML as the basis for future standards in the area of (multi-modal and spoken) conversational systems. Indeed, SCXML can be seen as an attempt to *webize* STATECHARTS. For details of SCXML, see the document *State Chart XML (SCXML): State Machine Notation for Control Abstraction* (Barnett et al., 2015).[1]

[1] `https://www.w3.org/TR/scxml`

For sophisticated conversational systems, SCXML by itself does not suffice. While SCXML may be able to handle reactive aspect of such systems, proactive aspects such as planning, decision making, and many other forms of problem solving and "intelligent" behaviour must also be programmed. Therefore, in addition to interactional capabilities, sophisticated conversational systems require knowledge representation and reasoning capabilities. In its current form, SCXML does not provide developers with sufficient such means.

The problem lies with the data model and the scripting language that is normally used with SCXML. The most commonly used profile of SCXML employs JavaScript for this, which should come as no surprise since JavaScript has found such uses in HTML, SVG, and in several other web standards. However, JavaScript is not exactly known for its capabilities for knowledge representation and reasoning. In this chapter, we propose that Web Prolog should be used instead, since it is much more suitable language for data modelling in particular, due mainly to its roots in Prolog (rather than Erlang).

The idea is not new. A proof-of-concept implementation of a system using Prolog as a data model and scripting language for SCXML has previously been developed by a team at the University of Darmstadt in Germany (Radomski et al., 2013). The authors of the cited paper were concerned with multi-modal conversational systems, so the paper is well worth a read because it shows that others than we think (or thought – they may have changed their minds) that SCXML and Prolog form a combination that fits the needs of developers of such systems. However, we are taking a slightly different, and, we believe, a simpler and more lightweight approach to the design of the interface between the two languages. Since we are aiming for a version of SCXML which harmonises well with Web Prolog and the architecture of the Prolog Web, our motivation is also different. In other words, we are trying to adapt our proposed SCXML profile to Web Prolog and the Prolog Web, rather than the other way around.

12.1 A playful introduction to STATECHARTS and SCXML

In its simplest form, a statechart is just a deterministic finite-state automaton, in which state transitions are triggered by events appearing in a global event queue. The simple statechart in Figure 12.1 serves as out first example. The labelling of one of the transitions with the symbol 'play' suggests that the example is drawn from the field of game development:[2]

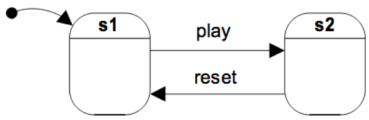

Figure 12.1: A simple statechart suggesting a game.

Here, `s1` is the initial state. If an event `play` appears in the event queue, a transition to `s2` is taken. For any other event, the machine will remain in `s1`. Clearly, this is just an event-driven state machine of the kind we saw in Chapter 11.

[2]We are borrowing some of our examples from (Brusk and Lager, 2008).

Any statechart can be translated into a document written in the linear XML-based syntax of SCXML. Here, for example, is a document that captures the statechart in Figure 12.1:

```
<scxml datamodel="web-prolog" initial="s1">
    <state id="s1">
        <onentry>return('IDLE')</onentry>
        <transition event="play" target="s2"/>
    </state>
    <state id="s2">
       <onentry>return('PLAYING')</onentry>
       <transition event="reset" target="s1"/>
    </state>
</scxml>
```

Note that two `<onentry>` elements, the significance of which are not shown in the statechart, have been added to the document. They contain what in SCXML terminology is called *executable content* – instructions which allows the state machine to *do* things such as modifying its data model or interacting with external entities. In our proposed SCXML profile, executable content always consists of Web Prolog source code. In the case of the above example, it ensures that an event message – either `'IDLE'` or `'PLAYING'` – is returned to the spawning process each time the automaton is entering `s1` or `s2`. Such code – which may also appear as the content of `<onexit>` and `<transition>` elements – must succeed deterministically, and to ensure that it does, it is always executed as if it was wrapped in `once/1`. If an error relating to Prolog is thrown by such code, an internal SCXML error event will be sent to the mailbox of the underlying actor, where it may trigger transitions in the state machine which is run by the actor.

An SCXML document can be authored by hand, perhaps with the help of a dedicated XML editor supported by an XML Schema. There are also tools that can render documents as statecharts. Even better, there are tools that allow developers to draw statecharts on a canvas and save the resulting diagrams as SCXML documents.[3] Most importantly, as a way to greatly simplify the step from specification into a running application – our little "game" in this case – a valid SCXML document can always be executed by an SCXML conforming processor.

12.2 A statechart is an actor in disguise

A major feature of our proposal is that *we are encouraged to think of an SCXML process as an actor running on a node on the Prolog Web*. Indeed, to create an SCXML session, we do not need any special purpose predicate, but can simply use `spawn/3` as is, like so:

```
?- spawn(run([]), Pid, [
       type(scxml),
       src_uri('http://sources.org/game.scxml'),
       monitor(true)
   ]).
Pid = '8dca3c82-e24b-11e6-8f24-67ab7e08ee12'.
```

[3] See e.g. `https://en.wikipedia.org/wiki/YAKINDU_Statechart_Tools`

```
?- flush.
Shell got 'IDLE'
true.
?-
```

The goal in the first argument calls a built-in predicate `run/1` which tells the node to fetch the source at `http://sources.org/game.scxml`. The value `scxml` of the `type` option (`'web-prolog'` by default) informs the system that SCXML source is to be expected at this URI. When `run/1` is called the source is transpiled into Prolog and the process is spawned. The first argument of `run/1` can be used to transfer a list of clauses to be added to the dynamic data model of the statechart. (We will get to the significance of the data model further down.)

As we saw above, when `spawn/3` was called it came back with a pid. Also, as specified by the content of the `<onentry>` element in its initial state, the statechart actor sent us an event message 'IDLE'. Equipped with the pid we can now use `!/2` to send an event message to the process which will change its state to 'PLAYING'. In general, an event message can be any Web Prolog term except a bare variable, but in this case the value of the attribute `event` in the relevant `<transition>` tells us that an atom `play` will do:

```
?- $Pid ! play.
true.
?- receive({What -> true}).
What = 'PLAYING'
?-
```

At this point we may want to send the statechart actor an event message `reset` to make it switch back to the state `s1` again, and so on. At any point during the interaction with the actor, we may choose to kill the process by sending it a signal using `exit/2`. Since the `monitor` option of `spawn/3` is set to `true`, the process would deliver a `down` message to its parent the moment before it dies, exactly as in the case of an ordinary actor.

This extends to other uses of options. In the case of our example, the actor will run locally, but if we want to run it on a remote node instead, we only need to pass the option `node` with a URI pointing to the node, similar to what we would do with an ordinary actor. Also, the SCXML source code is fetched from a URI in the example, but using the `src_text` option would work too, if set to a string or an atom in valid SCXML syntax. However, note that the `src_predicates` and `src_list` options are not valid when `type` is set to `scxml`.

All of this would be rather pointless if it was the whole story, since after all, as we saw in Chapter 11, simple state machines can easily be implemented in raw Web Prolog. However, Harel (1987) introduced a number of (at the time) novel extensions to finite-state automata which are present also in SCXML. Here they are, organised into two columns:

- Hierarchy and history states
- Parallelism (a.k.a. orthogonality)
- Broadcast communication
- Process invocation
- Inter-process communication
- Data model and scripting language

In the sections that follow we shall illustrate the use of these extensions. As it turns out, the majority of the features in the left column are those that STATECHARTS (and the core of

SCXML) provides, while Web Prolog bears most of the responsibility for the features in the column to the right.

12.3 Hierarchy and history states

Statecharts may be *hierarchical*, i.e. a state may contain another statechart down to an arbitrary depth. From a methodological point of view this may be useful, as 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). The importance of support for refinement and clustering should not be underestimated. Refinement allows a developer to design the top layer of an application first, perhaps using stubs to simulate the behaviour of the application when in each of the top layer states, and the behaviour when the state changes. When supported by appropriate tooling, this way of working could make a world of difference. The fact that SCXML is closely aligned to statechart theory and UML will help those using model driven development methodologies.

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. (In fact, in STATECHARTS as well as in SCXML there are two kinds of history states: shallow history states, and deep history states. We shall gloss over this distinction here, and for our example it does not make a difference.)

The statechart in Figure 12.2 exemplifies both hierarchy and the history state by enhancing our simple game with a pause-and-resume functionality.

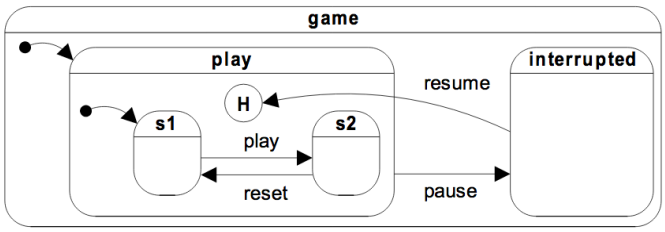

Figure 12.2: Pause and resume of our game.

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 then shows up in the queue, the system transfers to the history state `h`, which by the statechart semantics implies a transition back to `s2` again.

An SCXML document is structured so that it matches the topology of the corresponding statechart. A documents that captures the statechart in Figure 12.2 is shown below:

```
<scxml datamodel="web-prolog" initial="play">
    <state id="play" initial="s1">
        <history id="h">
            <transition target="s1"/>
        </history>
        <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>
```

12.4 Parallelism and broadcast communication

Another invention by Harel is that two or more statecharts embedded in a parent statechart may be run in *parallel*. This basically means that the parent statechart is in two or more states at the same time. This is an important mechanism for introducing modularity into a design.

Just to give a hint of how this might look like in STATECHARTS and in SCXML, we provide a simple model of the emotions and reactive behaviours of a non-player game character (NPC). Figure 12.3 depicts the statechart. Dashed lines mark the borders between parallel regions.

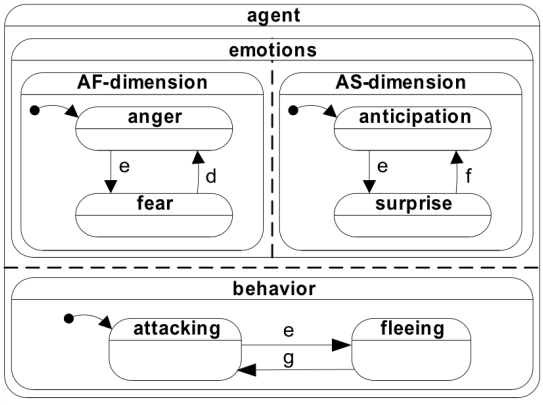

Figure 12.3: Emotions and behaviour of a non-player game character. [The event g should perhaps be changed to d]

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 behaviour. Then, if its friend dies (the event named `d`), fear will turn into anger, and it will attack instead. Note that if we did not have access to parallelism, we would have had to distinguish the *atomic* state `FearSurpriseFleeing` from 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)) that parallelism and to some extent also hierarchy are the means by which this often cited problem with traditional finite state automata – the exponential growth in the number of states and transitions – can be treated.

Here is the statechart in Figure 12.3 translated into SCXML:

```
<scxml datamodel="web-prolog" initial="agent">
    <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="anticipation">
                    <transition event="e" target="surprise"/>
                </state>
                <state id="surprise">
                    <transition event="f" target="anticipation"/>
                </state>
            </state>
        </parallel>
        <state id="behavior">
            <state id="attacking">
                <transition event="e" target="fleeing"/>
            </state>
            <state id="fleeing">
                <transition event="d" target="attacking"/>
            </state>
        </state>
    </parallel>
</scxml>
```

One statechart S_1 may influence 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 be detected by any transition in any active state in the statechart, this is often referred to as *broadcast communication*. Compared to the communication between different actors this is different, as a message sent by actor is always sent to a particular address (in the form of a pid). Furthermore, what happens when we send a message to two or more actors running

concurrently is subject to indeterminism, whereas what happens when we broadcast a message over parallel regions in an SCXML statechart is completely determined by where in the document order of the SCXML source the regions appear.

Based on the concept of a statechart *configuration*, defined as the set of states and parallel regions that the running statechart is currently in, STATECHARTS and SCXML support a way for an executing statechart to detect what is going on in parallel regions. A predicate `in/1` is available, which takes an ID and succeeds if a `<state>` or `<parallel>` with this ID is a member in the configuration. The `in/1` predicate can be used in conditions that are checked before transitions are taking place. (We do not show an example of how this works here. We could, but we do not.)

Below, we exemplify parallel statecharts and the communication between their substates with an SCXML document featuring two states playing ping-pong. Note that broadcast communication implies that when (say) Pinger sends an event, it will be "seen" by both Pinger and Ponger (as they are both active), but only be acted upon by Ponger. In this sense, the example works quite differently from the ping-pong example in Section 5.4, where two different actor processes were involved.

```
<scxml datamodel="web-prolog" initial="start">
    <parallel id="start">
        <state id="Pinger">
            <onentry>
                raise(ping)
            </onentry>
            <transition event="pong">
                raise(ping)
            </transition>
        </state>
        <state id="Ponger">
            <transition event="ping">
                raise(pong)
            </transition>
        </state>
    </parallel>
</scxml>
```

Note the use of the *targetless* `<transition>` elements in this example. Those are transitions where the matching of an event and/or the checking of a condition results in the running of the transition's actions (executable content such as the goal calling `raise/1`) but not in any change of states (and therefore not in the execution of the content of any `<onentry>` or `<onexit>` elements). In Section 12.6 we shall take a look at another use for such transitions.

12.5 Process invocation and inter-process communication

For creating an instance of an external service (e.g. another SCXML session), the W3C standard offers the `<invoke>` element. In our design of a new version of SCXML, and for the purpose of greater conceptual and terminological compatibility with the rest of the architecture of the Prolog Web, we use an element `<spawn>` instead, with a novel set of

valid attributes. In contrast to `<invoke>`, the `<spawn>` element allows us to specify not only *which* SCXML source to run, but also *where* (i.e. on which node) to run it, either locally or remotely as determined by the value of the `node` attribute. Also, note that the reason for having a dedicated element in the first place, instead of just using raw Web Prolog to spawn an actor inside an `<onentry>` element, is that SCXML dictates that when the state that contains the invocation is left, the invoked process must terminate. Having a separate element for the invocation of an external process is likely the only way to implement this, in our opinion very reasonable and potentially very useful, behaviour.

A statechart actor is allowed to send events to external entities, including actors such as pengines and other statechart actors. For this purpose, SCXML normally offers the `<send>` element. However, since Web Prolog has `!/2` as well as several other predicates (such as `raise/1`, `return/1` and `pengine_ask/2-3`) that are defined on top of `!/2`, we have no need for the `<send>` element. Since `<send>` is a complex element with a lot of attributes this simplifies things. Since executable content consists of Web Prolog calls only, we do not need a `<script>` element either.

In the following example we show SCXML source for a hierarchical statechart which spawns a pengine, sends it a query, and collects the answers. When the statechart receives the last answer it transitions to the final state, which in accordance with the statechart semantics explained above means that the pengine is automatically exited (and note that this happens despite the fact that we have set `exit` to `false`):

```
<scxml datamodel="web-prolog" initial="spawn-ask-collect">
    <state id="spawn-ask-collect" initial="ask">
        <spawn type="pengine"
               node="http://remote.org"
               exit="false">
            p(a). p(b). p(c).
        </spawn>
        <state id="ask">
            <transition event="spawned(Pid)" target="collect">
                pengine_ask(Pid, p(X))
            </transition>
        </state>
        <state id="collect">
            <transition event="success(Pid,Data,true)" target="collect">
                return(Data),
                pengine_next(Pid)
            </transition>
            <transition event="success(_,Data,false)" target="f"/>
                return(Data)
            </transition>
        </state>
    </state>
    <final id="f"/>
</scxml>
```

This example also makes it clear that any Web Prolog term (except a bare variable) may serve as an event message. In a `<transition>`, the `event` attribute (if any) takes a pattern which is matched against the *whole* of any incoming event (similar to how `receive/1-2` works),

i.e. no distinction between event name and event data is made.

Finally, we note that SCXML does not define a mechanism for deferring events, so if an event is not handled by a transition in the current state (or in any of its ancestor states), it will simply be discarded.

12.6 Web Prolog as a data model and scripting language

SCXML gives developers the ability to define a *data model* as part of an SCXML document. In our SCXML profile, a data model consists of a `<datamodel>` element containing definitions in Web Prolog of a number of predicates, the clauses of which through code injection will eventually end up in the private workspace of the underlying actor where it will complement the node resident program (if any) that the node on which the document eventually will run makes available.

If present, the value of the `cond` attribute in a `<transition>` element is a Web Prolog query, and transitions may thus be conditioned on the data model, the current event and the current configuration (using the `in/1` predicate as explained above). We suggest that values of variables bound by such queries should be made available in the actions of the transitions, where executable content which can modify the data model and communicate with external entities can make use of them.

Note that, in order to avoid leaving unwanted choice points around, queries in `cond` attributes must run deterministically, as if wrapped in `once/1`. Of course, in contrast to executable content, they are allowed to fail though, in which case the transition is not taken.

Here is a simple example, implementing Euclid's algorithm for calculating the greatest common divisor of a set of numbers:

```
<scxml datamodel="web-prolog" initial="run">
    <datamodel>
        number(25).
        number(10).
        number(15).
        number(30).
    </datamodel>
    <state id="run">
        <transition cond="number(X), number(Y), X > Y">
            Z is X-Y,
            retract(number(X)),
            assert(number(Z))
        </transition>
        <transition cond="number(X)" target="stop">
            return(X)
        </transition>
    </state>
    <final id="stop"/>
</scxml>
```

Note that neither an `event` nor a `target` attribute is present in the first `<transition>` (in document order), and yet the document will do the job as advertised and send the computed value (`5`) to the process that spawned it and then terminate. In this way SCXML allows

us to create systems of forward chaining condition-action rules (also known as production systems). Although such systems may, in the form given here, lack sophistication, they might still be useful. With SCXML we get them for free, as part of a bigger package.

One might object to the use of clauses of dynamic predicates as shown above on the grounds that they are like mutable variables and therefore subject to the usual problems with such entities in the context of logic programming. Our response would be to argue that, in essence, SCXML is an imperative programming language where the use of mutable variables must be deemed acceptable. Indeed, we would claim that, in the above example, it is SCXML rather than Web Prolog that is in control and is using the dynamic clauses as mutable variables. Having said that, we would suggest that there may be ways to equip our SCXML profile with a kind of mutable variables which are more suitable as well as more efficient to update. In Section **??** we shall take a brief look at one such idea, based on the notion of a *frame*.

Another objection that might be raised is that there is nothing to stop us from inserting code with side effects in conditions. For example, nothing stops us from using the following `<transition>` in place of the second `<transition>` in the example above:

```
<transition cond="number(X),return(X)" target="stop"/>
```

However, as in the case of `when` conditions in receive clauses, we side with those who think that the responsibility for avoiding the abuse of side effects in situations like these should rest on the individual programmer. Calling `return/1` in a `cond` attribute is simply bad practice, a practice developers can easily learn to avoid.

12.7 To conform or not to conform

Although SCXML was designed with extensibility in mind (cf (Barnett et al., 2015)), our approach has once again been to stick to KISS and to shrink more than we extend – an approach committed to already in Chapter 1 of this report. We have removed the elements `<data>`, `<script>`, `<assign>`, `<if>`, `<elseif>`, `<else>` and `<foreach>`, since they simply do not seem to mix well with a non-imperative language such as Prolog. The decision to use `!/2`, `raise/1` and `return/1` instead of a dedicated and very complex `<send>` element is another key to simplicity. We have also replaced the SCXML element `<invoke>` with the element `<spawn>` for reasons that we have explained above. We have kept the SCXML elements `<datamodel>`, `<state>`, `<initial>`, `<transition>`, `<parallel>`, `<history>`, `<final>`, `<onentry>` and `<onexit>` since they represent the core of statecharts. Although the proposed changes may seem radical, they still allow us to stick to the algorithm defined by the SCXML recommendation (as specified in Appendix D in (Barnett et al., 2015)) – the order in which transitions are tried and states are entered and exited, etc.

When designing the integration of Web Prolog in SCXML we have not cared much about conforming to the W3C recommendation, and we must admit that we have been more concerned with the adaptation of SCXML to Web Prolog than the other way around. Here are the main points:

- A statechart actor is an actor running on a node on the Prolog Web which has been programmed by means of SCXML with Web Prolog as a data model and scripting language.

- Conceptually, the data model is mapped to the combination of the program (if any) which is resident on the node and the contents of the private workspace of the underlying actor. Note that the database may well be an RDF triple store.
- When we spawn a statechart actor, we get a pid which can be used to send event messages to the actor by means of `!/2` (or by predicates that are defined in terms of `!/2`). As always, `receive/1-2` can be used to listen for messages arriving from the running actor.
- Any Web Prolog term (except a bare variable) may serve as an event message. The `event` attribute of a `<transition>` element takes a pattern which is matched against incoming events (similar to how patterns in `receive/1-2` works). No distinction between event name and even data is made, and the `event` attribute expects not just a name to be matched against the names of incoming event messages, but a whole term that can be matched against name *and* data at the same time.
- Actions are specified as Web Prolog text content in the `<onentry>`, `<onexit>` and `<transition>` elements.
- When present, the `cond` attribute of a `<transition>` expects a Web Prolog query which works exactly like a `when` guard in a receive clause. Also, just like in Web Prolog, the values of any variables bound by this query are available in the action of the `<transition>`.

12.8 Abstract ideas, concrete languages

To see how we might think about the role of STATECHARTS and SCXML in fairly abstract terms, consider the famous equation coined by Robert Kowalski in (Kowalski, 1979):

Algorithm = Logic + Control

The equation emphasises the difference between *what* (logic) and *how* (control) in implementing an algorithm in a computer program. There are many ways to apply control in Prolog: using the cut, modifying the default top-down, depth-first search regime, writing meta-interpreters or compilers, just to mention a few. However, we are not thinking about that *kind* of control here, but rather about the very high-level control facilities needed to build conversational systems with strong demands on fine-grained interaction, orchestration, choreography and game- and/or dialogue flow.

We suggest that it makes sense to complement Kowalski's equation with the following equation of our own making:

Intelligent conversation = Logic + Even more control

We believe that the implementation of conversational systems requires a lot more control than the implementation of (most) algorithms. To achieve this, we combine three programming models: logic programming, actor-based programming and programming with statecharts. Since STATECHARTS is almost entirely about control, its role can be seen to be to add more

control, and control of a different kind (e.g. complex orchestration and choreography), to actor-based logic programming.

Statecharts may or may not be the best possible paradigm for specifying reactive system, the actor model may or may not be the best possible model for programming concurrency and distribution, and logic programming may or may not be the best possible approach to knowledge representation and reasoning. But no matter what stand one takes on this, most people would probably agree that as programming paradigms they are all interesting, they are mature, they complement each other, and they are all formally grounded to at least some extent. If they can be used together, and it seems that on the Prolog Web they can, they should form a suitable foundation for a multi-paradigm web programming language.

But of course, if concrete syntax counts for something, what we have ended up with in this proposal is not *one* multi-paradigm language, but *two* languages – one Prolog dialect (Web Prolog) and one XML-based language (SCXML). However, in terms of cognitive ergonomics, this may not be much of a disadvantage as the two languages have been connected to each other in a fairly elegant way: as a scripting language to its "master language" in a style that is common on the Web (and here we are thinking of the role JavaScript has in relation to languages such as HTML and SVG). In this manner, and on the level of concrete syntax, the role of SCXML is to provide high-level facilities for control when using Web Prolog for programming the Prolog Web.

Interestingly, and for what it is worth, the approach sketched in this chapter might also help to explicate the relation between the RDF standard and the SCXML standard (our version of it). It seems that Web Prolog may be able to provide what is required to bridge them.

12.9 Constructing user interfaces with statecharts

A user interface allows users to interact with an application. Designing and developing user interfaces is difficult. We need to make sure that a user can only perform one out of a given set of operations at a given point of time, or else the application may end up in an inconsistent state, which is likely to frustrate the user.

Faced with interaction management difficulties, developers need to harness all the tools that they can get their hands on: tools for thinking, tools for coding, and tools for debugging. Logic is a wonderful and extremely useful tool for thinking, but so are statecharts, and for thinking about user interfaces, we would argue that statecharts are a lot more useful than logic, and that for actually *implementing* user interfaces, executable statecharts are more useful than executable logic. In his book *Constructing the User Interface with Statecharts*, Horrocks (1999) argues for the use of statecharts as a tool for designing GUIs. As far as we know, no books have been written about "constructing the user interface with logic".

SCXML was designed by the W3C Voice Browser working group which saw it as a component which promised to make it easier to control voice interactions as well as multimodal interaction. So how much is it used today for building voice user interfaces or multimodal user interfaces? It is difficult to say, but we do know that hierarchical state machines are one of the most popular methods for dialogue management in the relevant industry.

Developing voice user interfaces (VUIs) is difficult, but developing graphical user interfaces (GUIs) is not exactly easy either, and the development of web user interfaces is not an exception. The interaction management problem is the main concern here too, and it is

made even harder in the context of an asynchronous event-driven environments such as web browsers. How can we withstand the kind of unpredictability that occurs when a user pushes a button more times than expected or interacts with input widgets in an order not expected?

The popular approach to solving this problem seems to be frameworks (such as React or Vue) and libraries (such as Redux, MobX and Vuex). In contrast, Web Prolog and SCXML are not frameworks nor libraries, they are languages.

Some front-end web developers have started to use statechart for the design and development of complex web applications. This allows them to design visually, to enumerate the different modes a system might be in and map them to states, to limit possible transitions from one state to another, and to decouple the state machine from the code that leverages it. They seem to still be in the process of trying to figure out what patterns might be most useful in the context of front-end we programming, but seen from the side it does look promising indeed.

David Junger, in (Junger, 2014) was an early proponent of the use of SCXML for building web-based GUIs, and in recent years, David Khourshid, a Microsoft software engineer, tech author, international speaker, and open-source contributor, has given talks about statecharts at web developer conferences all over the world that seem to have resonated with front-end developers.[4] Khourshid is also the main developer of the XState system, a JavaScript implementation of statecharts compatible with SCXML.[5] User forums for discussions about STATECHARTS and SCXML have also appeared in recent years, some with members in the hundreds.[6] Many of them are advanced users of statecharts, and they blog about it.[7] A useful and fairly complete collection of links to resources are given at `https://statecharts.github.io/resources.html`. Indeed, we may want to start thinking about a statechart community of practice.

Of course, front-end web developers use JavaScript rather than Prolog as a data model and scripting language. Whether Web Prolog can succeed in this role probably depends on the future availability of Web Prolog in web browsers, as well as on if web developers see any use for Prolog. This, in turn, would probably depend on how the field of AI develops in the future, and to what extent the Web is going to be used as a platform for application of AI techniques and concepts.

[4] This is Khourshid's latest talk: `https://youtu.be/DrHccvns-L0`

[5] `https://github.com/davidkpiano/xstate`

[6] See `https://spectrum.chat/statecharts` and `https://gitter.im/statecharts/statecharts`

[7] See e.g. `https://medium.freecodecamp.org/how-to-visually-design-state-in-javascript-3a6a1aadab2b` and `https://medium.freecodecamp.org/how-to-model-the-behavior-of-redux-apps-using-statecharts-5e342aad8f66`