Client-Side State-based Control for Multimodal User Interfaces

David Junger <tffy@free.fr>
Supervised by Simon Dobnik and Torbjörn Lager of the University of Göteborg

Abstract

This thesis explores the appeal and (ease of) use of an executable State Chart language on the client-side to design and control multimodal Web applications. A client-side JavaScript implementation of said language, created by the author, is then discussed and explained. Features of the language and the implementation are illustrated by constructing, step-by-step, a small multimodal Web application.

Table of Contents

Acknowledgments
Introduction
1. More and more modality components
    1.1 Multimodal Output
    1.2 Multimodal Input
2. Application Control
    2.1 Multimodal Architecture and Interfaces
    2.2 Internal application logic
    2.3 Harel State Charts
    2.4 SCXML
    2.5 Benefits of decentralized control
3. JSSCxml
    3.1 Web page and browser integration
    3.2 Implementation details
    3.3 Client-side extras
    3.4 Performance
4. Demonstration
    4.1 The event-stream
    4.2 The user interface
    4.3 Implementing an output modality component
    4.4 Where is the “mute” button?
5. The big picture
    5.1 Related work
    5.2 Original contribution
6. Future Work
References

Acknowledgments

I learned about SCXML (the aforementioned State Chart language) thanks to Torbjörn Lager, who taught the Speech and Dialogue course in my first year in the Master in Language Technology. Blame him for getting me into this.

Implementing SCXML was quite a big project for me. I would never have considered starting it on my own, as I have a record of getting tired of long projects and I was afraid that would happen again. So I must thank Torbjörn who encouraged me and provided some extra motivation by giving me opportunities to talk about this project with other interested people.

Those other people include Johan Roxendal, who was also very helpful in giving me access to early W3C tests and his insight as the developer of PySCXML. I also met Stefan Radomski and Dirk Schnelle-Walka, who enlightened me about the somewhat arcane MMI-arch recommendation and the ways its paradigm differs from my Web developer's perspective. And let us not forget Jim Barnett and the other active contributors on the VBWG mailing list.

The implementation of SCXML was partially funded by the Dialog Technology Lab of the Centre for Language Technology.

And I must thank Olivier Bonami and Karl Erland Gadelii, of the Université Paris-Sorbonne, who recommended the University of Göteborg and helped me get there as an exchange student in 2011. And Martin Kaså and Peter Johnsen who ensured I could stay here despite some administrative hiccups.

I could go on back to the beginning of the universe, but I'm sure you also would like to read about SCXML. Here we go.

Introduction

Multimodal applications available on personal computers and devices are typically — if not always — managed and controlled from a server with specialized software. But now that Web pages can access much of the device's hardware (including camera(s), motion sensors, microphone…) for multimodal input and output, and the computational power to process most of that information can be found on the device itself, it becomes tempting to create multimodal applications that run mainly on the client.

However, the complexity of designing and programming such an application can prove a great challenge with current tools. State Charts are known to be very good at design and management of complex applications [UIwithSC], and SCXML, an upcoming W3C recommendation for executable State Charts, was an ideal candidate for integration in Web browsers.

So I created JSSCxml, a client-side, JavaScript implementation of SCXML, enabling a new way of designing and implementing Web applications.

The thesis will begin by reviewing the relevant improvements in device and Web browser capabilities that enable multimodal applications. Then it will show why those applications benefit from client-side SCXML. Last but not least, I will present my implementation and illustrate it.

1. More and more modality components

Until recently, widely available hardware had very limited sensory capabilities and little power to process their input; programs in Web browsers were even more limited in how they could get that input, and limited as well in how they could produce content for output. Thus, Web applications required plug-ins and heavy server-side processing.

Because the Web as a platform is very attractive, those sacrifices were made anyway, and led to a generation of Web applications that suffer from poor interoperability, costly development, closed-source bugs and vulnerabilities, and lack of standardization and practical solutions to enable accessibility, forward-compatibility, and internationalization among other things. Multimodal applications have been rare. The complexity of developing them in addition to the intrinsic complexity and frailty of the available tools certainly bears some of the blame for that.

But that generation is over, thanks in part to new (and upcoming), standardized, native interfaces that client-side code can use and rely on to perform input and output.

1.1 Multimodal Output

Multimodal output capability has been around since Pong, at least in the form of video and audio. More to the point, Web applications are about to gain full access to it, natively, thanks to modern browser APIs as shown in this list.

Figure 1. Output Modalities available (soon) to Web applications

Text
: Web pages' basic form of expression, text can be generated, styled, and rendered in rich detail with modern CSS, especially for visual media but also to some extent as braille or speech on devices that support it (see Speech below). Text tracks [WebVTT] synchronized to audio/video streams no longer even require scripting.

Speech
: While most browsers support declarative HTML-to-speech to some degree, they do so in their own way and have little or no support for aural CSS [AuralCSS]. The standard Web Speech Synthesis API [WebSpeech], however, is being implemented and promises interoperability and finer control to application developers.

Graphics
: Modern browsers offer several ways to render rich animated graphics, declaratively with HTML and SVG, or programmatically with the Canvas. The latter supports 3D rendering with WebGL [WebGL], which enables declarative 3D markup through libraries (the plug-in era is ending [future of O3D]).

Audio/Video
: All major browsers support native audio/video playback [HTMLMedia] in a standardized (but not interoperable due to disagreements on codecs) way. Moreover, some browsers are rolling out an API for manipulating the audio stream [WebAudio] directly, which, among other things, promises better performance for custom speech synthesis.

Haptics

A W3C community group has been started to standardize APIs allowing Web applications to access the haptics capabilities of devices that have them. It is reasonable to expect that, if haptic feedback touchscreens spread widely, that API will be ready sooner rather than later. Start trembling already, for the Vibration API [Vibration] is widely implemented on mobile browsers!

1.2 Multimodal Input

The situation is not so advanced for multimodal input, which has stagnated for decades during which only proprietary plug-ins and cumbersome Java applets could access some of the hardware. Now is an interesting time, as new interfaces are being standardized or implemented.

Figure 2. Input Modalities available (soon) to Web applications

Text

Text entry is nothing new on the Web (or on computers in general). Any device whose physical input is translated into text can be used in decades-old HTML form elements. That means keyboard entry, but also platform-specific speech recognition, handwriting recognition, barcode scanners, etc.

Keystrokes

Generated by keyboards, gamepads, mice, trackpads, but also eye-blinks, voice commands or hand gestures on platforms that recognize them. As long as it comes out as a keystroke or click event, Web applications can handle it as DOM Events [DOMEvent].

Gamepads

Speaking of gamepads, there is a generic API [Gamepad] for accessing them, with normalized floating-point values for both their axes and buttons, and events when a gamepad is (dis)connected.

Pointing

Web applications support pointing (and scrolling) devices as long as they behave like a mouse, or, more recently, as a touchpad. There is no support for pointing in 3D, but multiple simultaneous fingers can be tracked, and support for more generic multipointer [PointerEvents] is on the horizon.

Gestures

It is possible to implement video-based body tracking in the browser but the results are only at a proof-of-concept level. The dedicated hardware and software required for high precision tracking have gotten into consumer hands recently, and in the near future that this thesis anticipates, the processed input will definitely be available to Web browsers and the technology will be more widespread. In the meantime, the only well-supported gestures are finger gestures on touchscreens.

Geolocation

Location information is available through a standardized API [Geolocation] for devices that support it, regardless of the method of location (usually GPS or Wifi-based). It is widely used by commercial websites.

Acceleration

Many devices come with motion sensors that can tell when the device is moved or rotated. The DeviceOrientation and DeviceMotion [DeviceMotion] DOM Events allow Web content to know all about it.

Audio

Live audio capture [MediaCapture] and manipulation [WebAudio] have standard APIs and are being rolled out while I'm writing my thesis.

Speech

Speech recognition also has a standard API [WebSpeech] that browsers are beginning to support. The recognition engine is platform-specific but, if the API is followed by implementers, Web applications will enjoy interoperability anyway. It is not equivalent to text entry because, unlike platform-based speech recognition, the application gets a lot more than just a string: it can specify recognition grammars, gets a list of alternative recognition strings as well as quick intermediate results, and possibly some prosodic information.

Video

Browsers are beginning to support live video capture [MediaCapture] from the device's camera. The stream is encoded by the platform but can be decoded by passing it to a `<video>` element. It could be used to implement various image analysis and recognition algorithms.

More

An exhaustive list of all APIs under development would be outdated the moment it was written. Instead, have a look at the Device APIs Working Group's roadmap.

Web developers are used to keyboard and mouse input (plus touch input in recent years). Voice developers are used to text and audio. Some games and accessibility applications use special hardware that can, for example, detect body movements, brain waves, or perform eye tracking. Even that hardware is becoming more and more commonplace, while the software to process their raw data gets better.

Not all those components have a standardized API yet (or any API at all) that Web applications can use to access them. In fact, as the previous lists show, the existing APIs are not yet fully implemented by most major browsers. But considering all the items in the previous lists that would not have been there two years ago, we can assume that this situation is only temporary.

If the current “HTML5” trend continues, all the input from common devices will shortly have their Web browser API and/or DOM Events. Now if only we could make Web applications smart enough to handle all that for powerful multimodal interfaces…

That is what the Multimodal Architecture and Interfaces specification tries to address, and where SCXML comes in.

2. Application Control

2.1 Multimodal Architecture and Interfaces

The Multimodal Architectures and Interfaces [MMI-arch] recommendation, approved in 2012, provides an event-based framework for the design of multimodal applications.

One goal of the MMI architecture is to reduce the complexity of multimodal development by modularizing the details of input and output away from the main application logic.

In that framework, input and output parts of the user interface (as opposed to, say, network I/O, which nonetheless benefits from the same abstractions) are known as *modality components*. Atomic components provide input events to the application or transform output events into something the user can see, hear, or touch. They can be used in more abstract modality components that *fuse* their inputs or *split* output events into individual components.

For example, a system capable of detecting where the user is pointing or looking at could have an abstract component that fuses spatial input with the input from speech recognition, tagging deictic expressions. The top-level application would not handle low-level signals, but only receive more meaningful input from that speech + deixis component.

Communication between components is based entirely on events, which makes it robust, asynchronous, and nearly as easy to do over a network as locally.

That is also true of communication between modality components and the application logic, which the MMI architecture calls the *Interaction Manager*. Actually, it is a good name for those internal controllers that manage abstract modality components, but the top of the pyramid does more than manage interaction; thanks to the abstraction provided by the modality components, User Interface (UI) management may be a very small part of the top-level controller. Therefore I will not use that term to refer to application logic.

Regardless, encapsulating modality components and using events for inter-component communication is a good start to make an effective multimodal development platform.

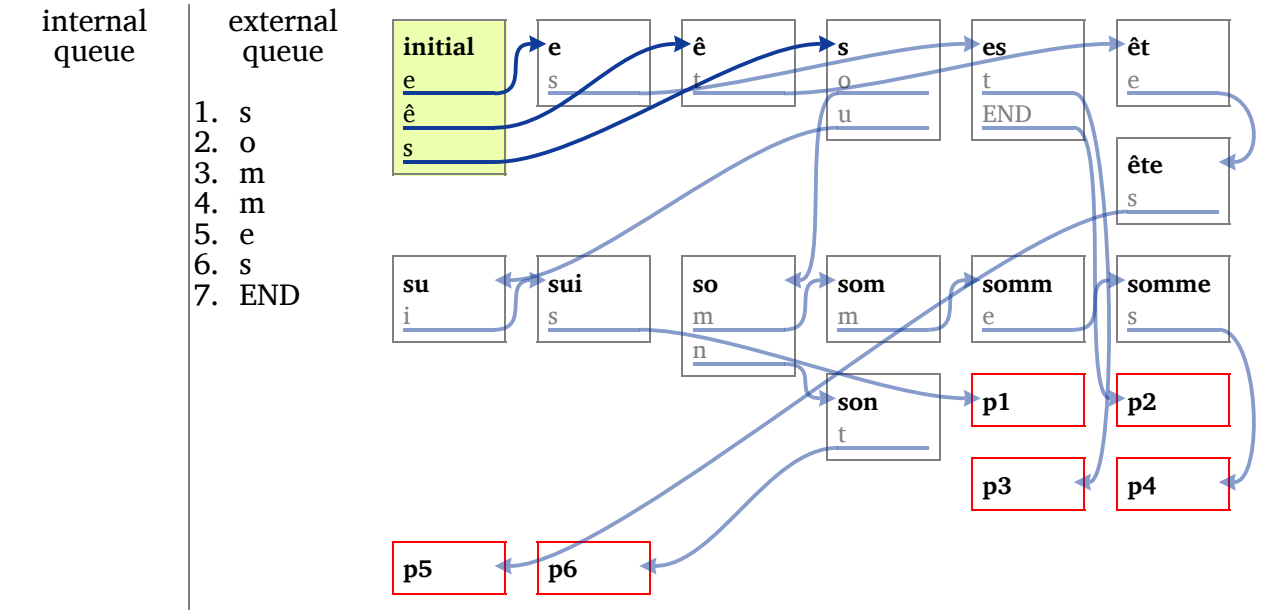
2.2 Internal application logic

UI in existing Web applications in fact already uses events, at least for input, in the form of DOM Events. That is good news, as Web developers will be familiar with the concept. But DOM Events are specialized, with a rather rigid structure, propagation and capture process that makes them unsuitable to serve as more than glue between input elements and the JavaScript code that controls the application.

Yet, event-based coupling provides the same advantages to other parts of an application as it does for modality components in the MMI-arch, and some other components (network communication, subprocess management, local I/O) with DOM events. Internal application logic could use that too, if the code was made of small logical units communicating with events rather than direct function calls, and with the right event distribution system.

A common paradigm for event-driven processing is the *State Machine*. It is a directed graph of *states*, where the system begins in one (or several) initial state and takes *transitions* thence to other states in reaction to *events*. In its simpler form, the *Finite State Automaton*, only one state is active at a time and the only output from the system is whether or not it reaches a final state and which one. FSA are very widely used in computer science as well as in computational linguistics, in part because of how easy it is to visualize them. Simple "dialog trees" (which are seldom strict trees in the mathematical sense) are instances of (sometimes extended) FSA that are still used for dialog management in the latest video games and in many voice application platforms.

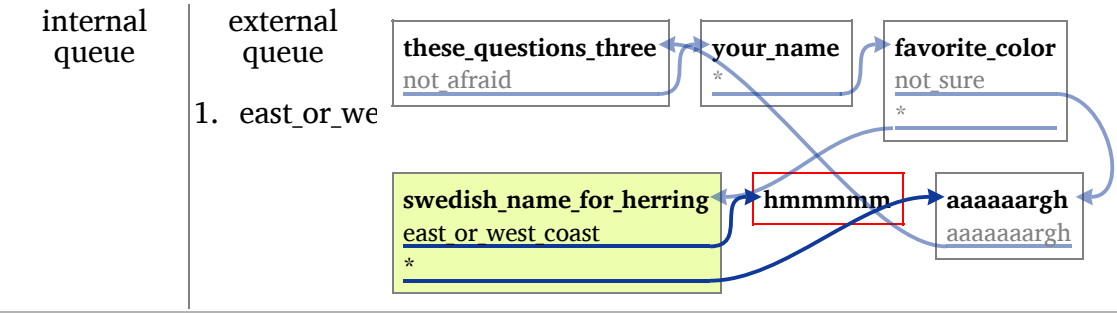
Figure 3. A Finite State Automaton



Boxes represent states, with their name at the top, followed by the list of their transitions (with arrows pointing to each transition's target and the text indicating which event will trigger it). Active states have a green background, and final states have a red border.

This sort of FSA is commonly used for pattern-matching on strings, but it can do a little more. The above example not only matches the present forms of the verb "to be" in French (equivalent to the regular expression `est?|êtes|s(?:uis|o(?:mmes|nt))` in PCRE syntax or `es(t)|êtes|s[uis|o[mmes|nt]]` in standard mathematical notation), it can tell which person it is by ending in the appropriate final state.

Figure 4. A small dialog tree



But, while FSA are generally good at handling events, and can be easily extended with a data model, their expressive power decreases very fast as the system they attempt to represent becomes more complex. That is even true of some more powerful state machines or flow charts that allow the execution of code. Those are used for example in the Lego NXT-G graphical programming language (which is of special interest because the Lego Mindstorms environment has access to unusual modality components) or in VoiceXML (a current standard for voice applications).

While those programming languages are indeed able to easily express useful event-driven programs, they lack certain features to make large applications a reasonable endeavour. And yet they sometimes acknowledge the issues themselves: VoiceXML for instance introduces “global” transitions in an effort to limit the need to duplicate identical behaviour into many different states. But that solution is only a special case of hierarchical states, which would be far more flexible.

Another issue, and one that is especially relevant on the Web because of its hostility towards fragmenting, proprietary platforms, is that all those existing state-based languages are platform-specific, each having its own features and idioms. If you want to program a Lego robot, you cannot use your knowledge of, say, the Automator utility in Mac OS X (it is an IDE for a flowchart-based, extensible graphical language using AppleEvents internally). A good solution for the Web must have a precise definition, and be easy to use, effective, and versatile, so that is has an actual chance of becoming a standard and not just one more niche platform.

2.3 Harel State Charts

The first step towards creating that solution was finding a state-based formalism that is powerful enough to be used to design applications of any size, yet simple and flexible enough that authors can get started quickly and produce reusable resources.

Features of Harel State Charts [HarelSC] and their merits for application design have been discussed for decades (largely by David Harel himself). Here is a summary of those features, introducing some important vocabulary and showing how they improve on FSA.

- State Charts have a *data model*. They can store and access data, and the events they process and send can contain a data payload.
- States in a State Chart can contain states of their own. That is, the system being in one state *S* means it is also in the parent state of *S* if it has one, and in one sub-state if *S* has any. States that do not have any sub-states are called *atomic* states, and other states are called *compound* states. The State Chart is thus a forest of state trees, rather than a flat list of states.
- Moreover, states declared as *parallel* activate all their direct sub-states simultaneously instead of just one at a time. All active states, the set of which is the State Chart's *configuration*, may respond to events in parallel as long as their reactions do not conflict (the reaction of a state deeper in the hierarchy preempts its ancestor's reaction in that case).

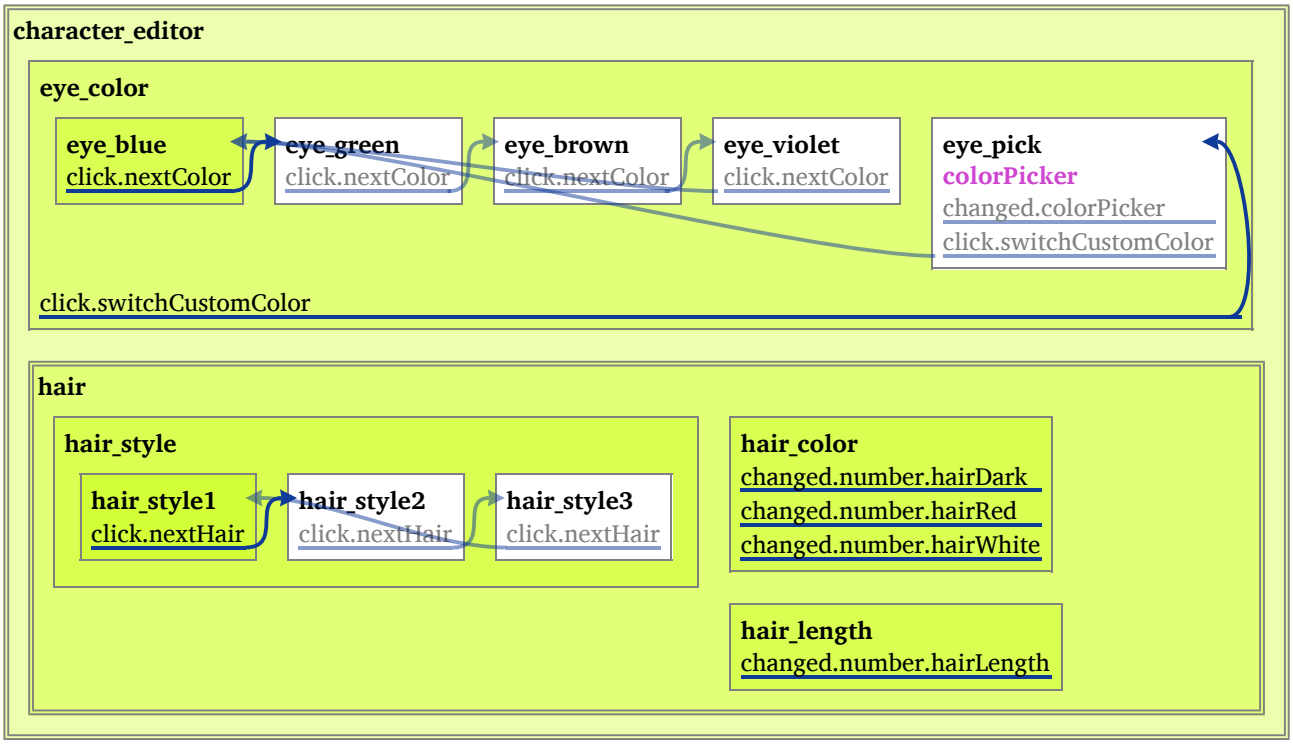
- Transitions between states can target special *history* pseudo-states, which remember the internal configuration of their parent state when it was last exited, and restore it. They can be set to restore only the immediate sub-state configuration, or go all the way to the atomic descendants. By the way, the set of atomic active states is the *basic configuration* and it is sufficient to rebuild the full configuration.
- The system can execute punctual *actions* when entering and exiting states, and when taking transitions (the latter may happen without entering or exiting any state when the transition is *targetless*). But State Charts also bring the notion of *activities*, which are sub-systems that can be executed asynchronously as long as the system is in a particular state.

Let us see some of that in action:

Figure 5. Controlling a character editor UI

This state chart manages user interaction with UI elements in a Web page, updates a data model representing the user's choices, as well as a graphical representation of the model.

… except the UI and the graphical representation do not exist, but that is the beauty of it: this controller can be designed early in the development process, with the expectation that the other, loosely coupled components will integrate with it easily. Every component can be reliably tested as a black box with event input/output, including the controller which serves both as a human-readable design document and a functional part of the application.



The double-bordered boxes represent *parallel* states: each of their sub-states deals with a logically separate part of the interface and model, and is generally able to do so without interfering with its sibling states. Adding control for a new aspect of character customization would most likely translate into a new sub-state, no matter how internally intricate.

The purple items inside states represent invoked *activities*. In this example, when the user wants to pick a custom eye colour instead of one of the default choices, a colour picker is invoked. It will remain invoked, and send events to the parent state chart, as long as the `eye_pick` state remains active, and will be automatically cancelled if the user wants to go back to default colours.

Many transitions in this state chart are *targetless* (graphically, they have no arrows): they do not react to user input by going to any state. Instead, they (would, if this were used in a real application) perform *actions* to update the data model and send events to the graphical component to reflect those updates visually.

One event, "click.switchCustomColor", has two transitions in its name. One in the top-level `eye_color` state, which is always active, and one in its `eye_pick` sub-state. When the latter is also active, both transitions will be enabled for that event. Since they are in conflict, however, only the more specific (the one in `eye_pick`) will trigger in that case.

The above example could, of course, become a module invoked by a larger application whenever the user wants to customize their character.

Hierarchical, parallel states and *invoked* activities in particular enable modular development and offer superior scalability. Activities (also known as *invocations*) are a key part of the MMI-arch. Not only can Harel State Charts be used for top-level application control and in modality components, they can also invoke and manage sub-components on their own, with the state chart semantics taking care of the underlying mechanisms for starting, stopping, and communicating with those sub-components.

Harel State Charts are still limited in that they are finite. It is possible and easy to produce smaller or bigger state charts on demand (state renaming aside, you can simply paste a state chart into a state of another state chart, or cut off a state and its descendants, without breaking the system). But a given state chart, once started, does not scale.

Despite that rigidity at runtime, and the presence of a possibly niche feature (history pseudo-states have their uses, but could be replaced by a more versatile mechanism, at the cost of some predictability), Harel State Charts were selected by the W3C to produce a new, general-purpose language that satisfies our needs quite well. And we will see how its formal limitations can be overcome.

2.4 SCXML

[SCXML] is a State Chart description language using the XML syntax and designed to be interpreted by machines as well as to provide a graphical, human-readable representation.

➢ Background

SCXML is being developed (at the time of this writing, it is a Last Call Working Draft waiting for Implementation Reports and solving its remaining issues to become a Candidate Recommendation) by the W3C's Voice Browser Working Group, which is responsible for the dialog management languages [CCXML] and [VoiceXML].

While there have been previous implementations or use of Harel State Charts, none has become a generic and standard tool for both design and programming. That is, most uses of the State Charts have been limited to design specification (most importantly in UML 2 [UML2]) and lacking direct machine interpretability. Some applications or documents, like the RPC spec [RPC-SC], had to define their own State Chart formalisms that were not reused, demonstrating the appeal of such representations and the need for a standard.

In a departure from its older VoiceXML and CCXML cousins, SCXML has no default voice-specific features such as grammars, speech, and form definition. However, it is extensible and those features (and more) can be added when relevant. On the other hand, the core control elements and State Chart semantics are more powerful and adaptable than in its predecessors.

➢ Features

Let us now focus on what SCXML brings to the table beside the State Charts formalism, whose features were illustrated in the previous section (here is the source code for figure 5).

For one thing, SCXML is on the way to become a W3C recommendation. It already has a dozen implementations, including several that are free, open-source and highly conformant (see table below). It is one of the preferred languages in the very recently approved recommendation for Multimodal Architectures and Interfaces [MMI-arch].

Figure 6. current general-purpose SCXML implementations

This table contains only SCXML implementations that have been updated in the current decade. Also, implementations specialized for and baked into larger products are outside the scope of this thesis, except in that their existence indicates an industrial interest in SCXML.

name	platform	conformance status
PySCXML	Python 2.6+	very good; aims to be the reference
SCION	ECMAScript	uses incompatible semantics
scxml4flex	Adobe Flex (ActionScript)	partial and outdated port of PySCXML
Commons SCXML	Java	poor and outdated
uSCXML	C++	very good
JSSCxml*	Web browsers (JavaScript)	very good, some client-side limitations

* this is my implementation

SCXML is also compatible with UML 2 State Charts so that existing UML diagrams can be easily converted to SCXML, and UML 2 editing tools should be able to handle SCXML with minimal effort. Besides, dedicated SCXML editors and debuggers will become available and, as I am developing one of them, I can safely predict that there will be at least one good, free, Web-based editor for SCXML.

When they are available, SCXML interacts nicely with DOM events, which enable loosely-coupled interaction with all sorts of HTML and XML-based applications in a way that authors are already very familiar with.

With all that, core SCXML is lightweight. It can be embedded in Web pages even on somewhat low-powered devices like tablets and set-top boxes. That is a big improvement over the monolithic VoiceXML and CCXML that simply had to run on a server (also using expensive, proprietary software on that server).

Those features should encourage widespread adoption of SCXML, particularly in Web applications.

2.5 Benefits of decentralized control

Assuming that there is indeed an appropriate technology to do it (and this thesis intends to demonstrate just that), there are several reasons to move Web application control to the client.

➢ scalability

Keeping a complex application state for hundreds (of thousands) of clients is expensive, as is running the application logic. On the other hand, keeping and managing a complex application state for one client is well within the capabilities of even the cheapest smartphone. So let them do it! Not only does it reduce infrastructure costs, it may enable more complex designs than would have been possible with server-side control.

➢ latency

Under some circumstances, network latency could degrade the responsiveness of server-controlled Web applications to unacceptable levels. Not so if the client handles user interaction. Even when communication is necessary, a powerful controller can remain responsive while blocking only the parts of the user interface that strictly require a server response, or simply marking them as unsynchronized without blocking. Using local storage, the application could function offline and synchronize whenever network becomes available.

➢ security and privacy

Some users care about their privacy (and those who do not, surely would not complain if you respect theirs anyway). In particular, streaming every detail of user interaction to a server may range from undesirable to unacceptable. There could also be security risks involved if that information were intercepted. By handling the details of user interaction on the client, an application need not transfer all that information. Other information, such as preferences and documents, can be saved locally as well (which means that even more of the application logic moves to the client, reinforcing the other benefits of a decentralized architecture).

➢ openness

Distributing more application code to the client means that the user has more control over how it runs and what it does. They could build plug-ins and modifications to adapt the application to their needs, increasing the value of the application with no additional effort from the original developer.

Of course, some developers (and a greater number of their managers) consider openness a drawback rather than an opportunity. Hence the proposal for protected media playback in HTML that is hotly debated at the moment. It is easy to imagine that its proponents would push for a similar API to run encrypted versions of fundamentally open content such as SCXML and JavaScript to (try in vain to, if history is any guide) prevent that content from being reverse-engineered and modified, while still sending it to client devices.

3. JSSCxml

But there was no implementation for Web browsers. *JSSCxml* is a nearly complete, free and open-source SCXML implementation that I started writing in june 2012, in JavaScript, leveraging standard APIs built in modern browsers such as the DOM level 4, XmlHttpRequests [XHR], and the EventSource interface [EventSource]. It enables the use of SCXML documents to control application behaviour and manage user interaction directly in the client. Traditional, server-side approaches suffer from several drawbacks that make JSSCxml worthwhile, as seen earlier, and JSSCxml offers some unique advantages to developers and users.

3.1 Web page and browser integration

A SCXML interpreter can be embedded in a Web page either declaratively as an HTML element, or programmatically with a line of JavaScript. The latter allows some extra options to be passed to the constructor and precise control on when the interpreter starts running. For details on how exactly to do all that, look at the JSSCxml Web site (spoiler: basic usage takes 2 to 3 lines of code, only one of which must be repeated for additional instances). The Web site also details the API and DOM Events that can be used to control the interpreter, and browser-specific extensions that I have created for JSSC and will explain later.

JSSC does not require any third-party library and the uncompressed source file takes 41 kB. For comparison, the text (just the text, not the whole HTML) of this thesis uses 74 kB. However, JSSC is designed exclusively for modern, standards-compliant browsers and will not run if any required API is missing (authors may use polyfills, of course). Hopefully, the faulty browsers will improve (or die) before long. What JSSC tries to do is be forward-compliant. It does not use any deprecated or non-standard browser feature, it is easy to extend with multiple independent modules, and, even though it is not version 1.0 yet, it will not have to break existing code on the way there.

3.2 Implementation details

The SCXML specification and the Web browser environment impose constraints that JSSC had to deal with. There are also some additions I made to work around limitations in basic SCXML and to improve usability on the specific platform JSSC runs on.

➢ **Web Workers**

Web Workers are JavaScript threads, with their own execution context, that run in the background and communicate with their parent Web page through DOM events. That sounds ideal for a SCXML interpreter which is supposed to be a black box with only events going in or out, and this option was given some thought.

However, Web Workers in practice lack features that normal client-side code can rely on, most importantly the DOM. By using and abusing the DOM, JSSC can do very powerful XML manipulation “for free”, and enjoys native performance. Also, at the moment there is no good browser interface for debugging Web Workers, which is a serious issue when you are looking at writing a few thousand lines of code.

So JSSC was written as a regular JavaScript program, running in the Web page's own context. Well, almost (keep reading).

➢ Encapsulation

SCXML documents can contain executable ECMAScript code that is supposed to run in an encapsulated *data model*. But Web browsers do not let you just create a new JavaScript context (yet) to evaluate code.

Spawning a Web Worker just for the embedded ECMAScript was not an option either, because it would have made it a black box for the interpreter just as for the rest of the Web page. And the interpreter needs to access the data model very often (and preferably without the overhead of inter-thread communication).

The closest thing to a fresh JavaScript context that unprivileged code can have is a dedicated window object, instead of the page's main window object. So JSSC creates a hidden `iframe` (which counts as a “window” just like a tab does) for each SCXML session, and uses some rather unsightly tricks to make ECMAScript code from the SCXML document run transparently in that `iframe`'s window, and to hide the browser's environment from that code as much as possible (source code). Those tricks include:

- shadowing predefined browser properties and internal JSSC properties by dynamically adding an object to the scope chain [ScopeChain] of the execution of SCXML-embedded ECMASCript
- calling `eval()` from a function defined in the `iframe` itself, rather than the main window
- writing `<script>` elements in the `iframe` to execute the same elements from the SCXML document
- wrapping setTimeout calls so their callbacks are executed in the modified scope (the setInterval wrapper is not yet implemented)
- defining getter and setter methods allowing access to SCXML system properties that are supposed to be read-only, such as `_event`

It is not possible to create a perfect sandbox (not without implementing a full ECMAScript interpreter, anyway) in the existing browser environment. The `iframe`-based data model implementation is relatively modular, so it can be replaced in case that changes in the future. Until then, there are several unavoidable ways for SCXML-embedded ECMASCript to gain access to its parent window, the browser, and its own interpreter. Obviously, that violates the encapsulation principle of SCXML, but at least JSSC makes it hard to do so accidentally.

➢ Working with JavaScript structures

The reference algorithm from the SCXML recommendation uses blocking queues to implement the interpreter's event queues. Such structures are unavailable and unnatural in Web browsers. The JavaScript way is asynchronous calls and callbacks. Thus, JSSC's translation of the SCXML event loop is exited as soon as the SC reaches a stable configuration and has no more events to process. The loop can then be recalled by the interpreter's `fireEvent()` method when it accepts an external event.

Then there are structures which *could* be implemented in ECMAScript, but at a price. ECMAScript natively has only one data structure: the hash table. Everything is a hash table, except some C-like arrays that were added recently and cannot contain references. ECMAScript Arrays are just hash tables with numerical keys, with the performance you would expect for insertion/deletion.

As a result, if you wanted to implement, say, an actual linked list in ECMAScript, you would have to use plenty of small objects (which are hash tables) to represent the links, with a property referencing the next link. The code, memory and performance overhead of having lots of tiny hash tables instead of C-like structures and pointers do not justify it. It is a shame, because a linked list would have been wonderful to represent the event queues in SCXML interpreters. Instead, JSSC uses plain Arrays, which do come with queue-like methods, and trusts the JavaScript engine (they're getting smarter all the time) to optimize accordingly.

Yet another structure used in the reference algorithm is the ordered list. That one too would be inefficient to implement in any classic way in ECMAScript, but even more inefficient to *not* implement and just sort an Array every time it should be in order. That is particularly true because the order of the items in the list is not trivial to compute in the case for which SCXML uses ordered lists. That is, representing the state chart's configuration.

JSSC does keep a handy hash table with the unsorted configuration for the purpose of testing whether an ID is in the current configuration. But other tasks require the ordered configuration or can be optimized by representing the configuration in an ordered way. And when you want a mutable, ordered list, the underlying structure is typically some sort of tree or linked list. Yet, like the links in linked lists, tree nodes suffer from being represented as a hash table. Here, I wrote my own data structure.

➢ configuration tree and selection

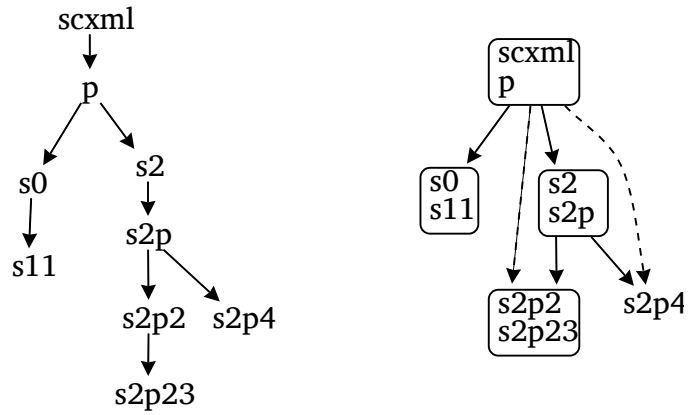
JSSC is designed to work directly on the parsed SCXML DOM. Some operations, however, can be optimized by compiling a configuration tree with shortcuts. They are defined in JSSC by the *CompiledTree* constructor (source code).

As long as there is no parallel state in a subtree, its active states form a *chain* that can be represented as a list of nodes instead of as a full-blown tree. JSSC's CompiledTrees therefore contain not states but lists of states. These lists contain a direct reference to both ends for fast access, as well as a property saying whether they end in an atomic state or a parallel one. The chains are immutable once constructed, so they do not suffer much from being implemented as Arrays.

Each CompiledTree node (that does not contain an atomic-ended chain) has a table associating the `id` of all descendant parallel states to the node containing the chain ending with that state. That way, when a new node has been constructed for any chain in the SCXML document, the algorithm can instantly attach it to its parent node by looking up the chain's parent's `id` in previously built trees (and vice-versa if the existing trees are not rooted to the `<scxml>` element). If two chains have been built that begin with the same state (e.g. because of conflicting transition targets) the algorithm will detect it when trying to attach the most recent chain and will resolve the conflict very quickly.

One perk of (fully-built) CompiledTrees is that they are unable to represent conflicting state configurations. As long as the building algorithm finishes (and it always will, unless you kill the browser or some targets did not even exist), the result is guaranteed to be a valid configuration even if the targets passed to it were in conflict.

Figure 7. normal tree vs. CompiledTree



The graph to the right represents the compiled version of the tree to the left. Dashed arrows stand for links to non-direct-child descendants. Rounded boxes are chains.

As the above figure illustrates with only a small configuration, CompiledTrees have less nodes: Only one for every `<parallel>` and every atomic state, instead of one for every state. That makes a greater difference in larger trees than the example, obviously. Less nodes mean less calls when using recursive functions on the tree.

One of the most performance-critical steps is transition selection. That is, looking over all the active states for transitions that match an event and current conditions. Many events can arrive quickly without triggering a state change or even without matching any transition at all, and the interpreter must be able to determine that before it can discard the event and dequeue the next one.

The selection process looks at all active states, and examines the transitions in each state. Under certain (frequent) conditions, some of the states or transitions may be skipped. The JSSC algorithm consists of the *CompiledTree*.select() method defined thus:

```
CompiledTree.prototype.select=function(test)
{
        var enabled=[]
        var allChildrenEnabled=true
        var enabledTargetedTransition=false

        if(!this.root.atomic){
```

```
                var childSelection
                for(var i=0; i<this.children.length; i++){
                        childSelection=this.children[i].select(test)
                        if(!childSelection.enabled.length){
                                allChildrenEnabled=false
                                continue
                        }
                        enabled=enabled.concat(childSelection.enabled)
                        enabledTargetedTransition |= childSelection.hasTargets
                }
        }
        else allChildrenEnabled=false

        if(!allChildrenEnabled){
                var t
                for(var p=this.root.path, i=p.length-1; i>=0; i--) if(t=test(p[i])){
                        if(!t.targets)
                                enabled.push(t)
                        else{
                                if(enabledTargetedTransition) break
                                enabled.push(t)
                                enabledTargetedTransition=true
                        }
                        break
                }
        }
        return {hasTargets:enabledTargetedTransition, enabled:enabled}
}
```

This uses the classic recursive pattern for *depth-first traversal*, with the first part (under the condition `if(!this.root.atomic)`) of the function applying itself to child subtrees and pooling the results. The second part selects results within the current subtree. Depth-first traversal is equivalent to *selection order*, where a state is looked at before is parent, and siblings are examined left-to-right. That is important because transitions must be preempted, and later executed, in that order.

Transition matching is done by the `test` argument, a function that returns the first positive match in a state (or nothing).

The first optimization simply comes with using the CompiledTree structure, by reducing the number of calls as mentioned earlier. Recent browsers on recent hardware, even without CompiledTrees, are at no risk of overflowing the call stack unless the state chart contains hundreds of thousands of states (worst case would be a chain of single child states, overflowing at a few dozen thousand states; with CompiledTrees that chain would be reduced to one node). But memory and complexity savings are interesting long before then.

The SCXML specification deals with conflicting transitions by picking the one deepest and leftmost in the tree among a set of conflicting transitions: the first in selection order. Also, only one transition (the first match) may be selected for each active atomic state. Note how, in the second half of the `select()` method, some or all of the code may be skipped. That is, if all subtrees have selected a transition, the current node does not even look at its own transitions (yellow mark). Otherwise, as soon as a transition is selected in the current node, the function returns (orange). And if that transition has targets, and a transition with targets has already been selected, then the new transition will conflict with the previous one and therefore it is not selected (green).

More optimization is possible, to skip all conflicting transitions instead of only the hierarchical type, but it would require knowing the transitions' LCCA (Least Common Compound Ancestor) in advance. Since JSSC allows transitions with dynamic targets, it computes the LCCA lazily and so cannot in general use the LCCA during transition selection (only transitions that are selected will have their LCCA computed).

➢ dynamic transitions

Dynamic targets mean that transitions in JSSC can have a `targetexpr` attribute instead of `target`. The attribute is evaluated just before taking the selected transitions in a microstep, and only if the transition was selected, so it will not show up as arrows on the graphical visualization.

Indeed, this feature makes the State Chart… not a State Chart, since it is no longer a connected graph at all times. In the general case, such structures lose some of their mathematical appeal. But dynamic transitions are an optional feature and, when used properly, do not make the SCXML document unstable nor unpredictable. If you can prove that the evaluation has a fixed set of outcomes, you can reason on the State Chart as usual despite that dynamic transition. And they are powerful and generic.

They could be used to implement `history` pseudo-states. Another example: In november, a user on the VBWG mailing list had a largely unrelated feature proposal that could also be implemented rather easily with dynamic transitions:

Figure 8. from the VBWG mailing list (archive here)

> suppose [a parent session used] `<invoke type="http://www.w3.org/TR/scxml/" src="http://example/other.xml#s2"/>`. The content of the [invoked sc]xml source would be:
>
> ```
> <scxml>
>   <state id="s1"/>
>   <state id="s2"/>
> </scxml>
> ```
>
> According to current draft the processor should start interpretation by entering s1 via (implicit) initial transition.
>
> Would not it be useful to allow start by entering s2 specifed in the frament?
>
> This would allow to chose the default entry state externally […]

Arguably, an external entity should not tell a SC what states to go to because a SC is supposed to be a black box. But that is beside the point. In this case, rather than add another specific mechanism for this specific case, a dynamic transition could use values passed to it by the parent session.

Yet another use, dynamic transitions can replace a long switch-like list of transitions such as:

```
<transition cond="next=='s1'" target="s1"/>
<transition cond="next=='s2'" target="s2"/>
<transition cond="next=='s3'" target="s3"/>
…
```

In that last case, using the dynamic transition `<transition targetexpr="next"/>` is not only concise, but saves time during selection.

Eventually, JSSC will be able to interpret a SCXML document that is being modified at runtime. To make that possible, JSSC computes very little in advance, but still tries to run fast, keeping costly results as long as they are reusable. That includes the compiled configuration tree, which is not recalculated every microstep if all the selected transitions were targetless. Also, when targets and LCCA are computed for a transition, they are stored and reused unless the transition had dynamic targets.

➢ scripting

All ECMAScript code embedded in SCXML attribute values is parsed at runtime. That is a constraint of working in the browser; only native SCXML support could avoid it. Modern browsers come with really good JavaScript optimization, but those do not apply to code evaluated by `eval()`, which JSSC uses for code in attribute values. Therefore, JSSC users are advised to put any performance-sensitive code in `<script>` elements (which are optimized on the fly, but there is an overhead for that), or even better, outside the interpreter so they can be optimized at load time and cached. JSSC provides a way for SCXML-embedded ECMAScript to transparently access libraries and other objects in the containing HTML document without resorting to platform-specific syntax inside the SCXML document:

Figure 9. Passing a library to a SCXML interpreter instance

```
var myLibrary = {…}

var sc = new SCxml("src.scxml", null, {lib1: myLibrary})
```

The *myLibrary* object defined in the parent HTML page will be accessible to script content in the SCXML document, under the name *lib1*

3.3 Client-side extras

Living in the browser is not just a matter of adapting to JavaScript structures and execution context limitations. A modern Web page is a rich environment with many I/O possibilities: modality components as we saw earlier, but also network and local storage. JSSC provides easy-to-use extensions to serve DOM events and HTTP communication to developers on a silver platter.

➢ DOM Events

Unfortunately, at this time JSSC is the only SCXML implementation that supports DOM events. DOM Event integration is part of the SCXML draft, but because there is no other implementation, it will most likely be removed from the candidate recommendation. Support for DOM events will therefore become a non-standard extension for some time, but that does not really change anything for JSSC.

JSSC's SCxml instances have a `fireEvent()` method to which external JavaScript can simply pass an existing DOM event, or a name and data if that is more convenient. Explicitly constructing a full `SCxml.ExternalEvent` is unnecessary in many cases.

On the output side, JSSC supports `<send type="DOM" …/>` nearly as the draft specified it, allowing SCXML to transparently send custom DOM Events to any element in the parent document. Both CSS selectors and XPaths are supported to specify the target, and I plan to add a mechanism to specify the target document according to my latest proposal on the VBWG mailing list.

➢ HTTP requests

The SCXML specification focuses on a symmetrical, fire-and-forget communication scheme called *Event I/O Processors* where both ends deal in events and both ends can send and receive to and from anywhere. That is, of course, impossible on a client machine; on the other hand, there are many existing and future services that Web applications can use over HTTP, and most do not understand SCXML events (even if the encoding of SCXML events were adequately specified for transport, which it is not).

JSSC fills the void left by the inability to implement remote Event I/O Processors by creating two extensions. The simplest is an executable content element (an *action*), `<fetch>` (full documentation), that lets a SCXML document send HTTP requests and receive events with the responses. It works much like XMLHttpRequest, and in fact it is implemented on top of it, simply wrapping it so the functionality fits into SCXML.

That gives SCXML good, generic, remote communication ability, but that is not all. Synchronous fetching of remote (and even local) resources is considered a very bad practice on the Web, more so for UI and high-level control which must be responsive all the time. The presence of the synchronous `<data src>` in SCXML is therefore an atrocity (which shall not be committed in JSSC). And it can be averted thanks, again, to `<fetch>`.

➢ server-sent events

XMLHttpRequests are commonly used to implement virtual persistent connections, using various third-party JavaScript libraries. But one method has given rise to an actual browser API and W3C candidate recommendation, and, considering that is is called “Server-sent *Events*”, I just had to provide a SCXML way to use it.

Technically, a server cannot initiate a connection with a client (to send it an event, for example). So what happens is that the client initiates the connection, and then both ends keep it alive. The browser transparently reconnects if it times out, and there is an optional facility in the message protocol to recover any event that should have been sent during the disconnection. The browser gives us all that for free, and the server requires relatively little work to fulfil its end, since it all happens over good old HTTP.

The EventSource interface was designed to receive DOM Events remotely, but it is trivial to adapt it for SCXML events. That leaves the design question: how do we use this the SCXML way? It cannot be an executable content extension, because it has a persistent effect. That goes against the spirit of all the existing executable content elements.

At first, the plan was to create a new element that would work very much like `<invoke>`, the element used to define *activities* for a state. It makes sense, since a persistent connection is a sort of activity that can be started (*opened*) and stopped (*closed*), and, while it is running (or rather *open*), it can send and/or receive events to/from its invoker. But invocations are a complicated part of SCXML, with hundreds of lines of code in many different parts of the code. Not something that should be duplicated lightly. So I designed an extension to `<invoke>` itself, allowing a slightly different behaviour (about nine lines of code out of three hundred, that's how slightly) to occur based on its `type` attribute (specification). Then, I designed the `event-stream` type (documentation) to fit into that. Then I implemented it just as specified, without touching anymore existing code, and it worked.

As part of the design process, I specified that invocations in my proposed extension could be asymmetrical (since the EventSource interface for server-sent events, as the name implies, only *receives* events). That led to Jim Barnett (editor of the SCXML recommendation) suggesting that might also become true of Event I/O Processors. So maybe JSSC will implement (half of) the HTTP I/O Processor after all. But let us get back to the present.

The `event-stream` type allows a client-side SCXML-based application to effortlessly open a connection to a remote server (it is not *that* easy on the server-side, unless you already have the right sort of server). As long as the connection is open, the server can push events to the client. The restrictions are the same as for the underlying EventSource object.

3.4 Performance

There remain several significant optimizations to implement in JSSC. Most importantly, the compiled tree is completely rebuilt each time the configuration changes. Instead, the same mechanism it already uses to avoid rebuilding branches when there are multiple targets in one microstep, could be extended to avoid rebuilding branches between microsteps. In large state charts, that could make a huge difference, bringing the speed of targeted microsteps much closer to that of targetless ones.

➢ test methodology

A series of tests were performed on the latest version of JSSC (as of 2 june 2013), using two devices, five state chart sizes, three types of microsteps, and two processing schedules. All the test charts except the first were generated by a recursive algorithm. They contain 1 parallel state for every 6 states, and most of their states have the same four transitions: two for “test1” and two for “test2”, with the first transition in each set having a `false` condition. The other transition for “test2” has a dynamic target expression which picks a random target state anywhere in the chart.

Green bars represent the time to process the event “test0”, which is not matched by any transition: thus, it only stresses the selection loop. Blue bars represent the time to process the event “test1”, matched by targetless transitions: the configuration stays the same; and purple bars stand for the time to process “test2”, which triggers randomly targeted transitions, so the configuration tree is rebuilt and the full microstep process is tested.

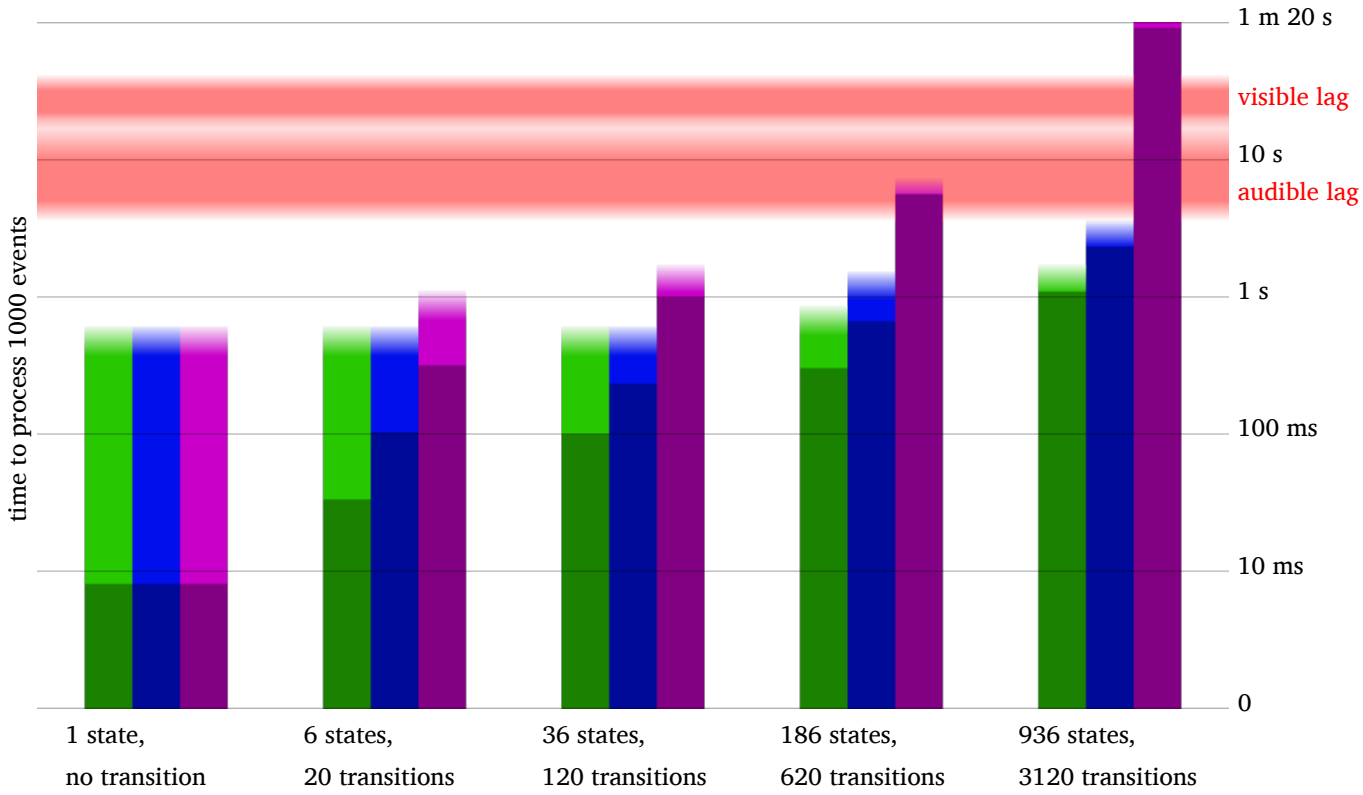
Unfortunately, JavaScript engines and profilers are unable to measure time more accurately than in milliseconds. Thus only the largest state charts would lead to non-zero processing times for one event. In order to obtain measurable times, a thousand events were added. Accordingly, the maximum acceptable time ranges (blurry red horizontal bars, because audio lag is more acceptable for violins than drums, and rendering frequencies can vary) were multiplied by 1000 (assuming that the time to process one event is roughly 1/1000 of the time to process 1000 events, which seems reasonable but could be wrong).

First, the events were all added before starting the interpreter and processed in batch (represented by the darker bar in the graph). CPU and compiler preference for batch-processing is obvious here. Then another thousand events were queued with the interpreter running, which leads to some wildly variable overhead (thus the blurry lighter bars — more tests will have to be performed to obtain proper statistical variation data).

➢ **results**

Figure 10. JSSCxml event loop performance on my MacBook Pro

These times were measured on a mid-2012 MacBook Pro (2.66 GHz Core i7) with plenty of free RAM, power savings mode disabled, and a full CPU core always fully available for the JavaScript thread alone (JavaScript being single-threaded anyway). There is no significant difference between Safari (JavaScriptCore) and Chrome (V8) on this test.

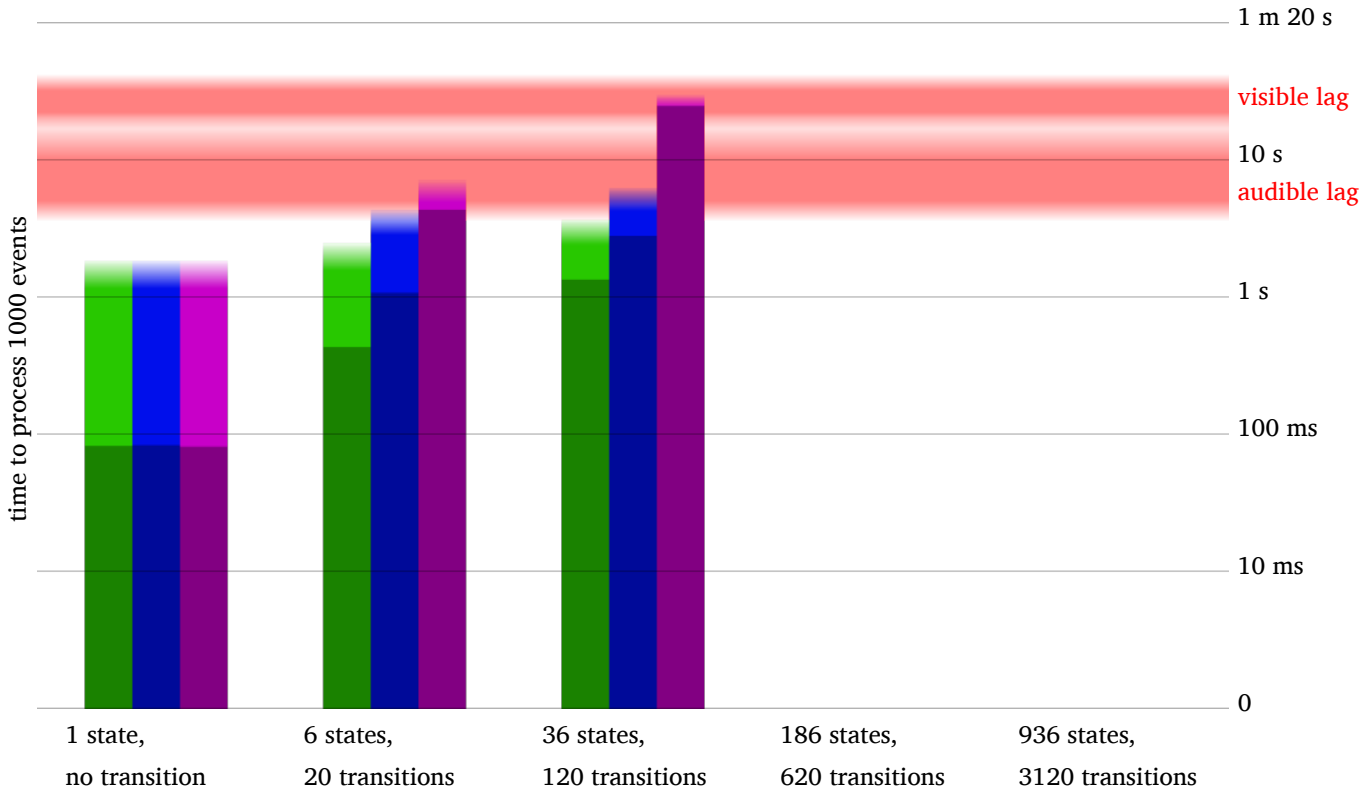


The graph shows that a chart with several hundred states and a mix of transition types can be expected to run fast enough in JSSC to cause no perceptible lag in the user interface on a relatively powerful computer.

It also shows that processing events as they arrive incurs an overhead of up to half a millisecond per event, compared to executing the exact same event loops without interruption. It should be possible to optimize JSSC to group event loops when many events arrive in a burst, reducing that overhead.

Figure 11. JSSCxml event loop performance on my iPod touch

These times were measured on a 4th generation iPod touch (1.0 GHz A4), with the latest Safari iOS, too little free RAM, but nothing big running in the background. This represents the low end of hardware that can run HTML5 applications.



The poor performance is unsurprising, and I stopped after the 36-state test SC. Still, the test shows that a state chart with a few dozen states (which is far more than any example in this thesis) running in JSSC can power Web applications with no perceptible lag on even an outdated smart-not-quite-phone.

4. Demonstration

This section will describe every step in building a simple Web-based CPU load monitor for a server, controlled with JSSC, with a tiny Node.js backend.

4.1 The event-stream

I will use the extended `invoke` type documented just above to send a steady stream of server load updates to the application. The events will be very simple: an event name (“update”) and data consisting of a JSON string representing an object with a “time” and a “load” properties:

Figure 12. a sample event from the stream

```
data: update
data:{ "time":1369817372833, "load":0.96435546875}
```

The server will send one event every 5 seconds to all clients that have opened a connection with the event-stream. That is implemented as a short Node.js server (code here), reverse-proxied by the nginx server for jsscxml.org so every part of the application has the same domain name. Kids, if you want to try this at home without sysadmin supervision, note that not all Web servers are able to proxy an event-stream properly because many of them will wait for the content to finish loading, with no way to configure them to just forward it as soon as it arrives.

Since it is served over HTTP we can view the raw output in a Web browser to make sure it looks as expected.

And that is all on the server side.

4.2 The user interface

One way to go about it would be to invoke the user interface from a top-level SCXML controller with something (that is not, but could be, defined in JSSC) like `<invoke type="HTML" …>`. But I prefer using the HTML document as the top of the MMI-arch pyramid, which means the HTML GUI will be written directly on the page rather than invoked. Either way, the JSSC instance would use DOM events to communicate with it.

Let us prepare a graph of server load over time in a 2D Canvas with some bits of text around it. That will serve as the main output component of the application, and giving it an `id="loadGraph"` makes it easier to target by DOM events. A little JavaScript to draw an axes overlay, and here we are:

Figure 13. server monitor UI

At this stage, with only one input and one output components, and no state-based logic, using SCXML as glue may seem like overkill. But look how little code it takes:

```
<scxml xmlns="http://www.w3.org/2005/07/scxml">
<state>
        <invoke type="event-stream" target="serverload.stream"/>

        <transition event="update">
                <send type="DOM" target="#loadGraph" event="draw">
                        <content expr="Math.round(_event.data.load*100)"/>
                </send>
        </transition>
</state>
</scxml>
```

We need to give the graph a DOM event listener so it will react to events sent by the controller. There will only be one event, arbitrarily called “draw”. The listener function gets the CPU load value from the event data, which resides in the `detail` property of DOM CustomEvent objects:

Figure 14. The “draw” event listener

Figure 14: The "draw" event listener

```
…
var C = graph.getContext("2d")
graph.addEventListener("draw", function(event){
        var y = event.detail
        C.drawImage(graph, 5,0,596,101, 0,0,596,101)
        …
        C.fillRect(596,100-y, 5,y+1)
}, true)
```

This simply moves the existing image to the left, then draws a vertical bar to the right of the canvas, at a height determined by the value passed in the DOM event.

4.3 Implementing an output modality component

Now let us implement another modality component, this time as a SCXML action (a.k.a. *executable content element*). In this example, it shall be a basic `<note>` action, which will allow me to show off the Web Audio API. Also, it will be pretty short. It will serve for a tuned "heartbeat" for the server monitor.

First, in order to do this properly, the custom element will need a custom namespace instead of the standard SCXML namespace. I'll reuse the JSSC namespace that `<fetch>` uses.

In the object that represents my namespace in JSSC, I'll now define a function named `note` (source code) that uses the `frequency`, `duration` and `volume` attributes of the element passed to it and creates a Web Audio chain with those parameters. Doing this is all it takes to make JSSC recognize the new element.

Now let us add it to the SCXML document, with the namespace and a little maths to make the range of values audible. Splitting each modality into its own sub-state makes it more modular, but is not required:

```
<scxml xmlns="http://www.w3.org/2005/07/scxml" xmlns:jssc="http://www.jsscxml.org">
<parallel>
        <invoke type="event-stream" target="serverload.stream"/>

        <state id="graph">
                <transition event="update">
                        <send type="DOM" target="#loadGraph" event="draw">
                                <content expr="Math.round(_event.data.load*100)"/>
                        </send>
                </transition>
        </state><state id="heartbeat">
                <transition event="update">
                        <jssc:note duration="250ms"
                                   frequency="Math.pow(_event.data.load*20, 2)+200"/>
                </transition>
        </state>
</parallel>
</scxml>
```

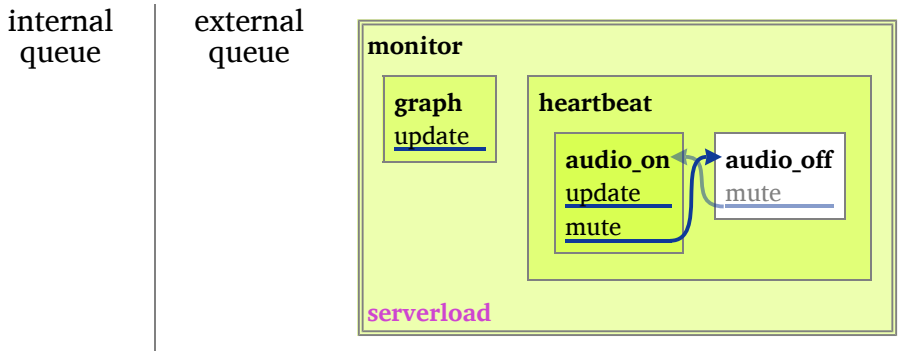
The `graph` and `heartbeat` states are in a *parallel* state so they are both active. But since both transitions are targetless, they do not conflict with each other. So when an "update" event arrives, both transitions are triggered and both effects take place.

Incidentally, this is a simple illustration of modality *split*. We get one event (from the server) and split it into a graphical output event and an aural output action. Why doesn't the `note` component use events for communication? Convenience. The price is that it is completely dependent on JSSC and very limited by the semantics of executable content. A sophisticated synthesizer should certainly be event-driven, in fact it would use MIDI events rather than DOM or SCXML events. Then, you would have to add a MIDI Event I/O Processor to JSSC. Which, by the way, would be awesome. But not in this thesis.

4.4 Where is the “mute” button?

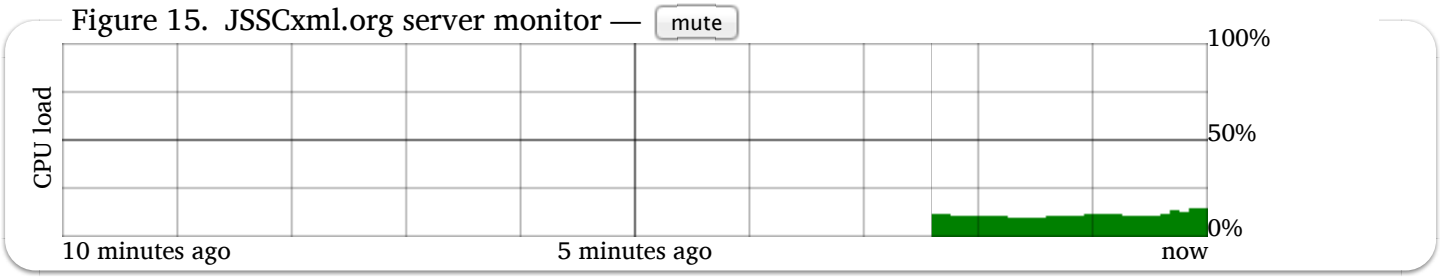
If you tried running the SCXML above, you would probably find the repetitive heartbeat sound annoying after a while. This thing needs a “mute” function. With SCXML, dropping in an extra state at the last moment is quite safe. Also, this is a chance to illustrate external event input, as the “mute” event will come from clicking a HTML button.

The `audio` state now has two sub-states. One, `audio_on`, has stolen away the transition for the `update` server-sent event, executing the `<note>`. Naturally, the other state, `audio_off`, will not have it. They also get a transition to each other, triggered by the “mute” event.



All we need now is a DOM event listener on this button, that will translate the DOM “click” event it receives when you click it into a “mute” event in the SCXML session. And *voilà*:

Figure 15. JSSCxml.org server monitor — mute



5. The big picture

5.1 Related work

This very year, there are two other Master students whose thesis centres around a SCXML implementation (uSCXML, by Stephan Radeck-Arneth) or a client-side State Chart implementation (SCION, by Jacob Beard). These happen to be the only general-purpose implementations under active development today beside my own JSSCxml, and PySCXML (which is also maintained by a Master student but with no thesis on the subject).

Having noted the coincidence, I must now tell the sad consequence: both publications most related to this thesis will be published at roughly the same time, and so, in-depth meta-analysis cannot happen now. But I can speak a little about the "competition" based on available documentation and my discussions with Jacob Beard (SCION) and the Darmstadt professors, Stefan and Dirk, supervising uSCXML.

uSCXML is very, very standard-compliant (passed all but three tests, last time I checked), and, being written in C++, noticeably faster than JSSCxml when running five hundred tests in a row. It is designed specifically to experiment with centralized MMI-arch concepts, with no ambition to set foot in Web browsers, and does not (but will, I hope) support DOM events. It does, however, provide a mechanism (which I helped define) to respond to HTTP requests as a Web server.

Either PySCXML or uSCXML would make, with the right extensions, an ideal server-side counterpart to JSSCxml. Exactly how that would work is a subject for future research, but the principle is quite natural.

SCION was designed as a multi-platform SC (but not quite SCXML) ECMAScript runtime. Its support for Harel SC semantics is lagging a bit behind, most importantly there are no invocations. SCION works by compiling a SCXML-like document into an ECMAScript object, then interpreting that object. Nonetheless, it provides the best available comparison with JSSC, since they both run inside Web browsers.

The author of SCION is now advocating specifying the SC directly as an ECMAScript object, since Web developers prefer working with JavaScript than XML. However, speaking of Web development, SCION has no direct support for DOM events or HTTP communications, relying on external code bound by callbacks to perform those tasks. Since they use different semantics, it is delicate to compare performance (many tests are incompatible), even in the same environment (also, unlike JSSCxml, SCION relies on multiple third-party JavaScript libraries, which may or may not be a burden depending on whether the application uses the same libraries otherwise).

SCION could be faster than JSSCxml due to its full compilation and more systematic optimization (and smaller feature set), but the difference would not be noticeable by the user. On the other hand, JSSCxml starts running sooner when SCION has to load all its support libraries, but again, it is not noticeable by the user ("sooner" here means in the order of one millisecond sooner). Also, SCION sticks more closely to the SCXML standard algorithm, which is less efficient than mine.

While I cannot deny that Web developers, myself included, do indeed prefer nice ECMAScript data to verbose XML, it tends to be different when working on large projects, integrating components together. Which is what this thesis is about. Not to mention that ultimately, Web-based graphical editors, and maybe other SCXML editors, should make it far easier to work with SCXML. The SC notation that SCION uses would have little chance of becoming a standard, and so would require authors to learn it just for SCION and to produce it without dedicated tools unless someone makes tools specifically for SCION. Jacob Beard is developing a SC viewer of his own, but I am unaware of any plans of his to turn it into an editor, and SCION's design would not easily support runtime modifications to the SCXML document.

5.2 Original contribution

I have created a working, rather efficient, and nearly complete, SCXML implementation for Web browsers. In the course of creating it I have contributed to SCXML development, in particular on the subjects of ECMAScript, remote communication, and DOM events.

I have developed SCXML extensions that allow browser-based SCXML to enjoy the usual client-server methods of communication, to compensate for the constraints of being in the browser in the first place, but also to enable more functionality.

For every feature I implemented that was not in the SCXML draft, I have published exhaustive documentation, and I have specified these features in a way that other SCXML implementations could adopt interoperably. uSCXML actually did adopt one already, and Jim Barnett has expressed interest in making them part of standard SCXML in the future.

I also created (or rather, am creating) a SCXML viewer based on and optionally bundled with JSSC. You saw it in action throughout this thesis, if you've been looking at the HTML version on a compatible browser.

6. Future Work

Much of the immediate future work is technical:

- finish JSSC 1.0
- implement the Web Sockets invoke type
- complete the SCXML debugger
- turn it into an editor
- make JSSC handle mutating documents
- publish reusable extensions that expose common modality components

But there is also much room for research and development.

- thoroughly evaluate the performance of JSSC vs. server-side SCXML
- what if SCXML is used on both client and server?
- compare client-side SCXML with client-side JavaScript (vanilla and with common libraries) and SCION

- see how a logic data model can be integrated with JSSC (PySCXML and uSCXML have an experimental Prolog data model, but it cannot work that way in a Web browser)
- develop new visualizations for SCXML documents
- build multimodal applications!
- explore what else JSSC can do that is not traditionally done on the client

Speaking of what can be done with JSSC, it is time to mention Spyderbrain's Dialog Web Lab [WebLab], which is an early, experimental attempt at using JSSCxml for dialog management. A less proof-of-concept dialog manager is certainly among the future applications of SCXML.

In fact, while SCXML's dialog management capabilities have not been the focus of this thesis, they represent an important application of this technology, most particularly on the client-side where many other dialog management platforms are unavailable.

References

[MMI-arch] Multimodal Architecture and Interfaces http://www.w3.org/TR/mmi-arch/ , W3C recommendation

[SCXML] State Chart XML (SCXML): State Machine Notation for Control Abstraction http://www.w3.org/TR/scxml/ , W3C Last Call working draft

[UIwithSC] Ian Horrocks (2009), *Constructing the User Interface with Statecharts* *http://dl.acm.org/citation.cfm?id=520870%5D*

[WebVTT] WebVTT: The Web Video Text Tracks Format http://dev.w3.org/html5/webvtt/ , W3C community group draft

[AuralCSS] CSS Speech Module http://www.w3.org/TR/css3-speech/ , W3C Candidate recommendation

[WebSpeech] Web Speech API Specification https://dvcs.w3.org/hg/speech-api/raw-file/tip/speechapi.html , W3C community group final

[WebGL] WebGL Specification https://www.khronos.org/registry/webgl/specs/latest/ , Khronos group specification

[future of O3D] Matt Papakipos, Vangelis Kokkevis (2010), *The future of O3D* *http://o3d.blogspot.se/2010/05/future-of-o3d.html* , posted on the O3D API Blog

[HTMLMedia] The `video` element http://www.whatwg.org/specs/web-apps/current-work/multipage/the-video-element.html , in the *HTML Living Standard*, WHATWG and W3C

[WebAudio] Web Audio API http://www.w3.org/TR/webaudio/ , W3C working draft

[Vibration] Vibration API http://www.w3.org/TR/vibration , W3C Last call working draft

[Gamepad] Gamepad http://www.w3.org/TR/gamepad/ , W3C working draft

[PointerEvents] Pointer Events http://www.w3.org/TR/pointerevents/ , W3C Candidate recommendation

[Geolocation] Geolocation API Specification http://www.w3.org/TR/geolocation-API/ , W3C Proposed recommendation

[DeviceMotion] DeviceOrientation Event Specification http://www.w3.org/TR/orientation-event/ (a misnomer, since it also defines DeviceMotion), W3C working draft

[MediaCapture] Media Capture and Streams http://www.w3.org/TR/mediacapture-streams/ , W3C working draft

[XHR] XMLHttpRequest http://www.w3.org/TR/XMLHttpRequest/ , W3C working draft

[EventSource] Server-Sent Events http://www.w3.org/TR/eventsource/ , W3C Candidate recommendation

[HarelSC] David Harel (1987), Statecharts: a visual formalism for complex systems http://www.sciencedirect.com/science/article/pii/0167642387900359 , in *Science of Computer Programming* volume 8, issue 3

[RPC-SC] The Open Group (1997), *DCE 1.1: Remote Procedure Call*, chapter 8: Statechart Specification Language Semantics http://pubs.opengroup.org/onlinepubs/9629399/chap8.htm

[DOMEvent] Events http://dom.spec.whatwg.org/#events in the *DOM Living Standard*, WHATWG and W3C.

[ScopeChain] Richard Cornford (2004), Identifier Resolution, Execution Contexts and scope chains http://jibbering.com/faq/notes/closures/#clIRExSc in the *comp.lang.javascript FAQ*.

[WebLab] David Junger, Torbjorn Lager and Johan Roxendal (2012), *SCXML for Building Conversational Agents in the Dialog Web Lab* in the SLTC 2012 Proceedings http://fileadmin.cs.lth.se/nlp/sltc2012/proceedings/sltc2012proceedings.pdf , page 42

This thesis is available from http://www.jsscxml.org/thesis.html.

It was successfully defended on the 10th june 2013.