Proceedings of the 1st EICS Workshop on Engineering Interactive Computer Systems with SCXML

Dirk Schnelle-Walka, Stefan Radomski, Torbjörn Lager, Jim Barnett, Deborah Dahl, Max Mühlhäuser (eds.)

Fachbereich Informatik
Telekooperation
Prof. Dr. Max Mühlhäuser

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Table of Contents

PREFACE

The W3C MMI Working Group suggests the use of SCXML [1] to express the dialog control of multimodal applications. The overall approach has already been shown to be suitable i.e. to decouple the control flow and presentation layer in multimodal dialog systems [5]. It has been used in several applications to express dialog states [2] or to easily incorporate information [4] from external systems.

As SCXML approaches formal W3C recommendation status and more applications employing SCXML start to appear, we gathered experiences and in general areas where clarification, further standardization or extension are needed or open new perspectives, like [3].

The workshop provided a forum to discuss submissions detailing the use of SCXML, in particular, multi-modal dialog systems adhering to the concepts outlined by the various W3C standards in general and related approaches of declarative dialog modeling to engineer interactive systems.

Our goal was to attract a wide range of submissions related to the declarative modeling of interactive multi-modal dialog systems to leverage the discussion and thus to advance the research of modeling interactive multi-modal dialog systems.

These proceedings contain the keynote from Jim Barnett and six submissions around the different aspects of engineering interactive systems with SCXML.

Format

The workshop was conducted as a two-tiered event: i In the first part the scientific contributions with regard to application and extensions of SCXML were presented, while ii the second part was in the format of an open-panel discussion, where suggestions that arose during the first part were detailed and elaborated.

ORGANIZERS AND PROGRAM COMMITTEE

The organizers are early adaptors of SCXML as well as leading experts from the SCXML working group.

Dirk Schnelle-Walka leads the "Talk&Touch" group at the Telecooperation Lab at TU Darmstadt. His main research interest is on multimodal interaction in smart spaces.
Stefan Radomski is a PhD candidate at the Telecooperation Lab at TU Darmstadt. His main research interest is about multimodal dialog management in pervasive environments.
Torbjörn Lager is professor of general and computational linguistics at FLoV, University of Gothenburg. His main research interests include computational logic, web technology and state machine technology for building web-based multimodal systems.
Jim Barnett is a software architect at Genesys, a contact center software company. He is the editor of the SCXML specification.
Deborah Dahl is the Principal at Conversational Technologies and the Chair of the W3C Multimodal Interaction Working Group. Her primary technical interest is practical applications of speech, natural language and multimodal technologies.
Max Mühlhäuser is full professor and heads the Telecooperation Lab at TU Darmstadt. He has over 300 publications on UbiComp, HCI, IUI, e-learning and multimedia.

The list of program committee members is as follows:

- **Rahul Akolkar** (IBM Research, USA)
- **Kazuyuki Ashimura** (W3C, Japan)
- **Stephan Borgert** (TU Darmstadt, Germany)
- **Jenny Brusk** (University of Skövde, Sweden)
- **Sebastian Feuerstack** (Offis, Germany)
- **David Junger** (University of Gothenburg, Sweden)
- **Stephan Radeck-Arneth** (TU Darmstadt, Germany)
- **David Suendermann-Oeft** (DHBW Stuttgart, Germany)
- **Raj Tumuluri** (Openstream, USA)

ACKNOWLEDGEMENTS

The 1st EICS Workshop on Engineering Interactive Systems with SCXML was an interesting experience where participants with all their different backgrounds had lively discussions about their applications and research regarding SCXML. If you contributed to it in any way, we are grateful for your involvement. We wish that these proceedings are a valuable source of information in your efforts. We hope that you will enjoy reading the following pages. We would like to thank the organizers and the program committee for all their work.

The Technical University of Darmstadt's efforts around SCXML have been partially supported by the FP7 EU Large-scale Integrating Project SMART VORTEX co-financed by the European Union.

REFERENCES

1. Barnett, J., Akolkar, R., Auburn, R., Bodell, M., Burnett, D. C., Carter, J., McGlashan, S., Lager, T., Helbing, M., Hosn, R., Raman, T., Reifenrath, K., and Rosenthal, N. State chart XML (SCXML): State machine notation for control abstraction. W3C working draft, W3C, Feb. 2012. http://www.w3.org/TR/2012/WD-scxml-20120216/.

2. Brusk, J., Lager, T., Hjalmarsson, A., and Wik, P. DEAL: dialogue management in SCXML for believable game characters. In *Proceedings of the 2007 conference on Future Play*, ACM (2007), 137–144.

3. Radomski, S., Schnelle-Walka, D., and Radeck-Arneth, S. A Prolog Datamodel for State Chart XML. In *SIGdial Workshop on Discourse and Dialogue* (Aug. 2013).

4. Sigüenza Izquierdo, Á., Blanco Murillo, J. L., Bernat Vercher, J., and Hernández Gómez, L. A. Using scxml to integrate semantic sensor information into context-aware user interfaces. In *International Workshop on Semantic Sensor Web, In conjunction with IC3K 2010*, Telecomunicacion (2011).

5. Wilcock, G. SCXML and voice interfaces. In *3rd Baltic Conference on Human Language Technologies, Kaunas, Lithuania* (2007).

SCXML: Current Status and Future Prospects

Jim Barnett
Genesys
jim.barnett@genesyslab.com

INVITED TALK

The W3Cs Voice Browser Group started work on SCXML as part of its VoiceXML 3 effort. One problem with VoiceXML 2.x was that it was difficult to re-use markup because it mixed user interaction with flow control. For example, someone might write a form (or series of forms) to collect the callers credit card number, but the markup that interacted with the caller was tightly coupled with the logic that decided where to go next in the application, so it was difficult to incorporate those forms in another application. We therefore decided to separate flow control from user interaction, and developed SCXML as a pure flow control application. Given the composition of the Voice Browser Group, most of the people who have worked on SCXML have had backgrounds in speech recognition or natural language processing, but our intent was to keep voice-specific constructs out of SCXML, and we have been pleased to see the diversity of applications to which it has been put, including a current effort to use it for modeling the spread of infectious diseases.

The two obvious candidates for a flow control language were state machines and logic programming (or goal decomposition), both of which are widely used for dialogue control. We chose state machines because Harel had produced an excellent formulation of them, and because VoiceXML 2 already had a state machine embedded inside it in the guise of the Form Interpretation Algorithm. There is ongoing interest in combining SCXML with logic programming, for example by defining a Prolog datamodel. Since Harel has produced several versions of state charts, we chose UML state machines as our baseline, particularly because the industry has a lot of experience with them, and Harel consulted on their definition. SCXML has attempted to adhere to the UML state machine definition except where we had good reason for deviation, and we have intended for it to be easy to translate a UML state machine into SCXML.

One significant from traditional Harel State Charts is that SCXML must define concrete datamodels and event delivery mechanisms. This requires pinning down a lot of details that can be left vague in a graphical notation. The most important distinction, however, is that we chose to make the runtime behavior of SCXML as deterministic as possible. Harels work, for example, does not specify the order in which transitions in parallel regions are taken. This is a justifiable decision, since it is unlikely that a well-designed application would need to rely on such details. Nevertheless, the SCXML group decided to specify them, in part because most of us work for enterprise software companies, and our customers demand a high level of predictability and control. The last thing that a large bank wants is for the software to suddenly decide to do something different this time around.

One interesting question is why it has taken so long to finish SCXML, given that we had Harels work and UML as a starting point. One reason is that W3C process requires a high degree of consensus so that decisions take much longer than they do in a single company development project. Another problem is that people join and leave the working group as the work progresses. Each time a new person joins, the group ends up revisiting previous decisions to re-build consensus with the new member. One significant technical issue we had to deal with was the need to abstract away from the concrete datamodels and event processors. It took a significant amount of time to draw the boundaries correctly. Finally, it has taken us a long time to debug the interpretation algorithm that is included in the specification. The big problem was that there was no easy way to test changes to the algorithm. It might have saved time to create a reference implementation, even though W3C process does not call for one.

SCXML is currently at Last Call status, and we can move to full Recommendation status quickly if we get enough implementation reports. (In order to demonstrate inter-operability, W3C process calls for at least two independent implementations of each feature in a specification.) When we think of possible future work after SCXML 1.0, one obvious candidate would be the specification of new datamodels or event I/O processors, which could be developed as separate documents. Another possibility would be the addition of specific, limited, new features in the form of SCML 1.1. The choice of the features would depend on the participants, in particular on those who would commit to implementing them. A full overhaul of the language in the form of SCXML 2.0 would take several years and we would need a large group of committed people before we would even consider it.

EICS'14 Workshop, Engineering Interactive Systems with SCXML, June 17, 2014, Rome, Italy

Developing User Interfaces using SCXML Statecharts

Gavin Kistner
NVIDIA, Inc.
1350 Pine St.
Boulder, CO
gkistner@nvidia.com

Chris Nuernberger
NVIDIA, Inc.
1350 Pine St.
Boulder, CO
chrisn@nvidia.com

ABSTRACT
In this paper we describe NVIDIA Corporation's implementation of an editor and runtime for the SCXML statechart standard. The editor and runtime are used for both prototyping and production of user interfaces, targeted primarily for automotive in-vehicle interfaces. We show how state machines improve the simplicity and stability of application development, particularly when using the hierarchical and parallel states available in SCXML. We investigate the usefulness of statecharts in user interaction design. We further describe subtle additions and deviations from the SCXML standard, the motivations for these changes, and their benefits compared to a strictly standards-compliant implementation.

Author Keywords
SCXML; state machine; statechart; gui

ACM Classification Keywords
D.2.2 Software Engineering: Design Tools and Techniques: State diagrams

INTRODUCTION
Since 2003 we have developed a software product for creating 3D user interfaces. Since 2009 this tool has been known as NVIDIA's UI Composer Studio, or "Studio" for short.

In Studio all user interaction is handled through triggers known as "actions" that translate events occurring on objects in the scene to visual changes in the interface (Figure 1). Visual changes in Studio are most commonly specified as "slides", which control what aspects of the interface are visible along with animations and transitions. Conditional interactions—such as not responding to mouse clicks on a button when the button is disabled—are accomplished by placing actions only on specific slides for items in the interface.

EICS 2014 Workshop: Engineering Interactive Systems with SCXML, June 17, 2014, Rome, Italy

While actions have been effective at producing a functional interface, they have historically caused two problems:

1. Larger interfaces became hard to edit as interactivity was 'hidden' deep within specific slides of specific interface elements.
2. Combining interaction logic with the visual presentation made editing difficult whenever the interactivity needed to be changed independent of the presentation.

Visual States versus Logical States
It is often desirable in a software interface for changes in interaction to be paired with changes in the presentation. For example, when a text input is focused—accepting user input—it is beneficial to the end user for the visual appearance to reflect this and differentiate it from the case where the input is not focused. However, the visual state may not change along with the logical internal state.

One such example is the appearance of a modal dialog. Modal dialogs disable interaction with other visible content, but usually do not change the appearance of that content. In this case a single visual state (a slide) must be associated with multiple logical states.

A reversed example is when a transition animation is followed by a steady-state animation. In Studio such a situation is usually implemented using multiple slides. In this case we have the situation where multiple visual states are associated with a single logical state.

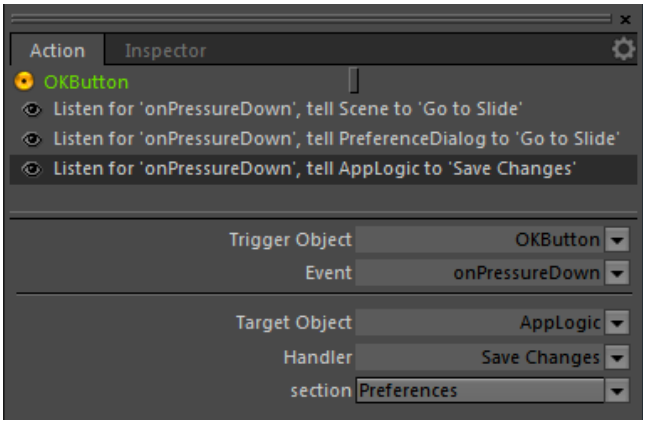

Figure 1: Scriptless Actions in UI Composer Studio

Separating Logic from Presentation

We set out to solve the problems described above by implementing the visual states independently from the logical interaction states. We further believed that this separation should provide additional benefits:

1. Interaction designers would be able to develop the logical states independently from graphical artists working on the interface, perhaps simultaneously.
2. Artists would be protected from accidentally breaking the interaction logic during development.
3. The interaction flow of the interface would be testable in an automated manner, independent of the interface.

To represent the logic of the system we chose to use a state machine.

State Machines and SCXML

Finite state machines (or simply "state machines") have been in use in a variety of technical fields since the 1950s. Traditional state machines have a single set of mutually exclusive states. The machine must be in exactly one state at any given time. Such systems are limited in their ability to efficiently express the interactions of a sophisticated software system. For example, describing a system of three independent buttons which have four possible states each—disabled, enabled, hovered, and active—requires a state machine with 64 states. These states represent the Cartesian product of the possible combinations of states for the buttons. We have been given (unsubstantiated) reports of such systems resulting in in-vehicle user interfaces with over 3,000 states. We would consider such a system to be unable to be easily tested, maintained, or even understood.

Harel statecharts [1] are a visual formalism of state machines. They provide three features that greatly simplify the description of a complex interface over a traditional state machine:

- The addition of orthogonal regions (also known as "parallel states") permits states from multiple sets to be active at the same time. This removes the combinatorial explosion problem described above; the machine requires far fewer states, instead authoring a simpler system that is better representative of the objects in the interface.
- The addition of hierarchical states allows a simple programming-by-differences methodology [2]. Child states can specialize a parent state, handling specific interactions as necessary or allowing the parent state to handle shared interactions. This reduces the number of transitions required in the machine, and in doing so it also reduces the chance of mistakes by reducing duplication of interaction logic.
- History states within hierarchical regions allow the state machine to record the active descendant state(s) when leaving them, and return to that same set of states later.

SCXML is an open standard [3] that uses XML instead of pictures to describe content similar to Harel statecharts. Some of the features of SCXML that are important to user interaction include orthogonal regions, hierarchical states, history pseudo-states, executable content on state entry/exit, executable content during transitions, and a formalized data model. SCXML also specifies a formal interpretation algorithm, with a wide test suite available to help ensure correctness.

While the feature sets of SCXML described above made it appealing as a choice for storing the user interaction, we felt that forcing users to understand the SCXML specification and type raw, syntactically valid XML would be both cumbersome and likely to cause mistakes. Once we decided to use SCXML we needed to write an editor to easily create valid markup, and then a runtime to support it.

SCXML VISUAL EDITOR

We created a new application named "Architect" to create and modify SCXML files. Editing is done visually, as seen in Figure 2. Our graphical editor provides many benefits over a text editor. It ensures that the user produces valid SCXML. It improves understanding [4, 5, 6] of the state machine. It allows executive stakeholders to review and approve interface logic without examining any 'code'. Finally, it hints at regions that are likely to be bug-prone.

We anticipated many of these benefits. In particular, using visual statecharts to express and discuss the user's navigation through application screens replaced a more cumbersome and error-prone exchanging of pictorial flow charts that were then translated to code with each change. We were able to replace this workflow by adding a feature

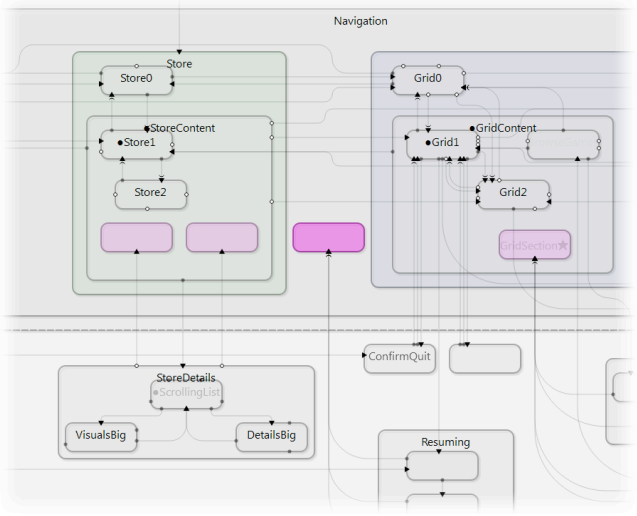

Figure 2: Portion of a production statechart created with Architect, with transition event labels hidden

that allows the user to filter the display of the statechart to only include transitions related to a subset of triggering events. This provides custom views appropriate for high level conversations, yet allows the same statechart to be analyzed under different contexts.

We did not anticipate the correlation between visually complex regions—such as states connected by many transitions, with the same events and different conditions—and the likelihood of bugs in that region. For example, the collection of states labeled `GridContent` in Figure 2 turned out to be the source of the most bugs in the application it was controlling.

Visual Representation

Our visual display of the statechart differs from Harel's in many ways. Most are designed to reduce visual clutter and improve understanding at first glance.

Harel shows the *default state* within a hierarchy (called the *default initial state* in SCXML) as a dot in the parent state with an arrow pointing to the default state. We simplify this to a single Unicode bullet prefixing the name of the default state, seen in the states `initial` and `a` in Figure 3.

While Harel's examples mostly use single-character event names labeled on the transitions, real applications often have multiple events with much longer names. For example, a particular transition in the application could be triggered by any of the events `dpad.right.down`, `touch.focusStore0`, or `bumper.right`. To avoid drawing large amounts of text on the statechart diagram we hide the name(s) of the event(s) that trigger a transition at the default zoom level. The user may zoom in to see event names exposed in the interface, or select a particular transition to see the triggering event(s).

A single parent state in SCXML can have multiple history states with different behavior (different initial targets). The circled **(H)** and **(H)*** notations used by Harel do not allow sufficient distinction between two or more `<history>` states within the same parent state. We instead display the full name of history states. We additionally append a star glyph to the name to visually differentiate them from normal states, with different glyphs indicating the type of history recorded ("shallow" versus "deep" recording).

We consider Harel's dotted separation for orthogonal regions to be a desirable visual representation, yet difficult to support for intuitive editing. Instead, we draw a SCXML `<parallel>` wrapper with a dotted border. In Figure 3 the states `X` and `Y` are orthogonal; both are active whenever the state machine is in the parallel state. This convention is convenient to edit, but has the disadvantage that it requires the consumer of our diagrams to understand this notation.

Organizing States

To help emphasize hierarchical placement we apply subtle shadows to states. This creates the perception of 3D stacking; child states appear to sit on top of their parent.

Some large applications developed with Architect have states hierarchically nested five or more levels deep. To better help visually distinguish the boundaries we allow the user to apply a background color to a state. The background of each state is semi-transparent, allowing the color of any parent state to be visible on each of its descendant states.

Adjusting the placement of child states within a parent state is constrained by conflicting requirements. We do not wish to allow a child state to be placed outside the boundaries of its parent state, since this would cause the visual diagram to no longer properly represent the internal hierarchy. Yet if we prevent a child from moving outside the boundaries of the parent a "claustrophobic" feel is introduced, forcing users to fight the system. Alternatively if we instead cause moving a child against the parent boundaries to resize the parent then we encounter additional problems, both with transition routing and the need to push sibling and parent states. A single errant child movement could destroy important layout of the states and transitions.

Our final solution was influenced by Alan Cooper's recommendation to allow users to "fudge" the system [7]. Users may temporarily create a visually 'invalid' statechart by placing states outside their parent, but we draw the outline of any states with invalid placement in a bright red color to indicate the visual error. This allows the user the freedom to move items around at will during editing, while still encouraging valid results. And, if the user truly wishes to change the parent of a state, we support alternative modes for dragging a child state into a new parent.

Display of Transitions

We believe that understanding the flow of control between states is most important in reading a statechart. We correspondingly expended a large amount of design and implementation effort on their appearance.

On each transition between states we draw a dot on the edge of the source state and a triangular head entering the target state. The source dot exists to make clear that a transition comes from that state and is not a line coming from another state that happens to go under this state. (The transparent background on states further prevents this problem, as any transition line going under a state is visible beneath it.) The dot is drawn smaller than the arrowhead and with reduced opacity to help distinguish them.

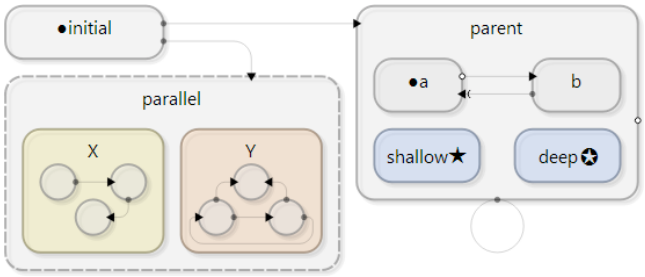

Figure 3: Examples of states and transitions in Architect

When a transition is guarded by a condition—dynamic code evaluated to determine whether or not to take the transition—we draw the circle for the source dot with a white background. This is demonstrated in the transition from state `a` to `b` in Figure 3. This visual differentiation helps to highlight to the casual observer that the transition may not always take place.

We draw transitions that have executable content uniquely to help highlight where side effects may occur in the statechart. As shown on the transition from `b` to `a` in Figure 3 these transitions have a small curved line added adjacent to the arrowhead. This visual style mimics a similar style (not pictured) that indicates when a state has executable content during entry or exit.

Certain transitions in SCXML may target the state that they originated from. We depict this as a circular transition, seen in Figure 3 at the bottom of the `parent` state. Other transitions in SCXML may not target any state at all. These "targetless" transitions are displayed without any line, as a single dot on the edge of the source state.

Transitions may be hand-routed by the user. Whenever a transition changes direction the corner is rounded. Beyond the aesthetic appeal, this helps to ensure that two transitions crossing each other at 90° angles cannot be mistaken for transitions that happen to turn at the same spot.

We draw the transitions with a semi-transparent line so that multiple collinear transitions are visually different from a single transition. For example, in Figure 2 the multiple transitions entering the pinkish states become darker than any individual transition line. We believe that subtle details like this—combined with our other work—result in a diagram that is both pleasing to the eye and that also is easier to examine and to understand.

Limitations in Graphically Representing SCXML

Our work at present does not yet allow the visual editing of all features offered by SCXML.

Architect sets a child state to be the default initial state by setting the `initial="…"` attribute on the parent state. SCXML alternatively permits an `<initial>` element to be created containing a transition with executable content on it. We do not provide a way to author such an initial transition. Users may instead create a state with that executable content on entry, and immediately transition from that state to the desired initial state.

We do not support the visual editing of transitions that target more than one state. Though this is reasonable to represent with some interim visual (similar to a UML Statechart "fork" node [8]) to date no statechart we have created has required this capability.

Finally, while we support the distinction between *internal* and *external* transitions in SCXML, we do not do so based on whether the transition's edge leaves the parent state as with UML Statechart *local* versus *external* transitions. Though this seems a good visual differentiation, we believe that it is not obvious enough for editing. It seems quite likely that an intended visual-only edit to the routing of a transition could accidentally result in a behavioral change.

CONNECTING LOGIC TO INTERFACE

Given a presentation authored in Studio and an SCXML state machine authored in Architect, we require a way to communicate between the two. Some changes to the logical state must be able to trigger a change in the interface, and some user interactions in the interface (such as tapping on a button) must be able to fire an event in the state machine.

Driving Presentation from States

To control the interface from the state machine, we need to be able specify interface-specific actions that may take place during any of the "executable" regions of SCXML: during the entering of states, the exiting of states, or during the activation of a transition.

Since SCXML is XML, we *could* specify the interface changes as executable content in a custom namespace. However, our automotive customers have asked to be able to re-use a single state machine with different presentation layers. For example, a high-end car may implement the interface using UI Composer Studio, while a less expensive model may use a simpler interface requiring cheaper graphics hardware. To support this we chose to specify the presentation control separate from the state machine.

To this end we designed an XML schema for a custom file (the "Glue" in Figure 4) that maps the entering and exiting of specific states, and the activation of transitions, to the desired changes in the presentation. While the format of this file is irrelevant to this discussion, its use highlights a limitation of the SCXML standard.

Referencing a state from this separate file is simple as the SCXML file contains a unique `id` attribute for each state. Referencing transitions, however, is not as simple: There is no such unique identifier present in the standard for transitions. To facilitate the reference, Architect adds a custom `uic:id="…"` attribute in a custom namespace to each transition. This value is editable by the user in order to apply a semantic and memorable label, but Architect ensures that the value entered is unique amongst all transitions. We hope that a future version of the SCXML specification may support unique identifiers on transitions.

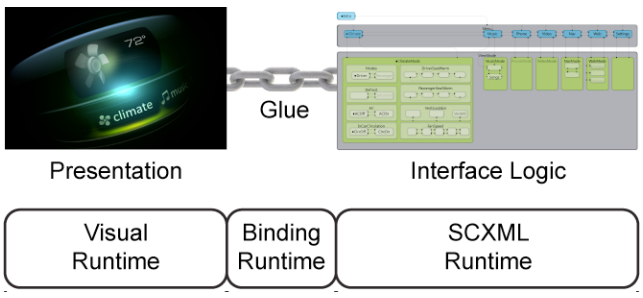

Figure 4: Gluing the Presentation to the Logic

Driving States from Presentation

Communication from the UI Composer-based interface to the state machine is performed via Studio's "actions". Instead of multiple actions on each of multiple slides tracking the `onPressureDown` event on a button and causing multiple interface changes (Figure 1) the artist instead creates a single master action that fires a semantic event into the state machine. The button always tells the state machine when it is pressed, for example, and it is up to the state machine to decide what—if anything—should occur as a result.

By processing all user interaction in the state machine, we enable the creation of multi-modal interfaces that can use touch, hardware input (keyboard or buttons), focusable interface elements, voice input, gaze tracking, camera-based gesture recognition, and more.

Synchronizing States and Presentation

Many of the applications we have developed have transitions in the presentation that correspond to a change in interaction. One such example is a 'welcome' animation that displays during application and vehicle start. During this animation no user input is accepted. When the animation completes interaction is enabled.

We could use the animation completing in the interface to trigger the logical state change. This provides a good experience to the end user, as the visual change is guaranteed to correspond to the interaction change. However, this also leaves our application at the mercy of the interface artist. If the artist modifies the duration of the animation to be 30 seconds, the user will not be able to interact with the interface during that time.

If, alternatively, a development team has an Interaction Designer ("ID") who is in charge of user experience and interaction flow independent of the artists, the ID may instead choose to use SCXML-based timeouts with fixed durations to trigger the interaction change. In this case the presentation is at the mercy of the logical interactions, possibly being pushed to a visual state before the artist's animation is complete.

We support the invocation of timeouts by using standard SCXML features. Upon entering a state we queue an event to `<send>` after a specified period, but an early exit of the state `<cancel>`s the queued event to avoid other effects. Figure 5 shows authoring such a situation in Architect.

RUNTIME IMPLEMENTATION

After editing the interface and logic, and gluing them together, the final piece needed for application support is a runtime for the SCXML logic. This runtime interprets the SCXML instead of compiling the state machine to code. This enables simpler introspection of the state machine during runtime. It also makes it easier to make changes to the logic without requiring any recompilation. Both of these result in shorter development cycles.

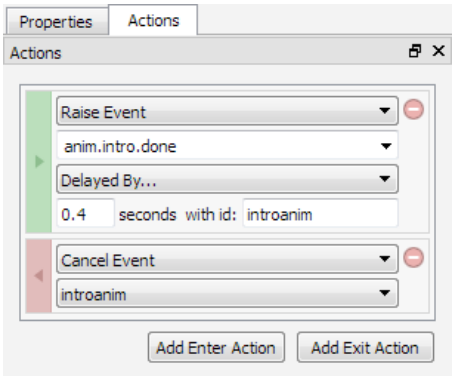

Figure 5: Firing an event after a timeout.

During evaluation of SCXML as a candidate language we first implemented prototype SCXML interpreters in the Ruby[9] and Lua[10] scripting languages. The official SCXML interpreter algorithm was still in flux at this time and we found it easy to test changes to the algorithm in these languages. In fact, the initial release of the NVIDIA® SHIELD™ portable game console [11] used the Lua-based interpreter for its game-browsing interface.

After we decided to commit to SCXML we wrote our official runtime in C++ with an SCXML scripting model that uses Lua for all conditional transitions, data model access, and executable `<script>` evaluation. The final implementation included in our product weighs in at around 4,000 lines of code, including the Lua script bindings but disregarding supporting libraries and header files.

The decision to use C++ was not due to performance issues; the Lua interpreter ran fast enough for our purposes on embedded hardware (though the initial SCXML parsing did delay startup slightly). The decision was based on four criteria:

1. The code base for all of UI Composer is C++, as it offers far superior debugging to Lua. Despite having our own Lua debugger (UI Composer Studio also exposes Lua in the interface layer) we find it easier to debug C++ than Lua.
2. Customers wishing to license our state machine may not want to use Lua at all; the scripting system is abstracted from the state machine and C++ is, in general, accepted by our customers in more varied contexts than Lua.
3. C++ allows a more compact representation of the problem and more optimization possibilities in terms of size/speed in the long term should such needs arise.
4. Our entire core development team is more experienced in C++ than Lua. However, the subset of SCXML required for our use case is a simple enough that it takes less than one developer to support the entire implementation, including all Lua bindings and maintaining our test suite.

SCXML Specification Features Not Supported

Our SCXML implementation is not fully compliant. There are features required by the standard that we have not found to be useful in our product, and have not implemented.

We do not implement invocation or communication with external services. This means that we do not support the `<invoke>` element, any subset features of `<send>` or `<cancel>` related to external services, or the `<content>` data container.

We do not support the `<param>` element for passing annotated data along with an event. While this might be useful in some scenarios, it has not yet been required. Also, there exist other mechanisms to accomplish the same goal in many cases, such as pushing event-related information into the data model instead of onto the event.

We do not support `<donedata>` for describing the resulting state machine information when it reaches a `<final>` state. Our applications using the state machine do not generally exist as services that need to communicate results to a separate system.

We do not support a SCXML I/O Processor (section D in the specification). Our engine only runs a single SCXML session at a time.

We are using this subset of SCXML in high-end production applications to great effect. While the missing features are certainly not useless, this shows that they are not necessary for certain domains. We hope that in the future the SCXML standard will be simplified to a core set of features—as occurred with the SVG Tiny standard [12]—with additional modules describing useful add-on functionality.

Unique Implementation Features

Our engine further deviates from the SCXML standard in various ways designed to improve the reliability of our applications.

State Machine Unit Testing

To help verify that modifications made to a complex state machine during editing did not break existing functionality we have developed an XML-based unit testing system for our SCXML engine. A unit test initializes a state machine with custom data model values and then specifies a series of events to inject into the system. Each event is followed by assertions about the currently active states or data model values. By stubbing out functions that make simple data model changes we can create tests that simulate a working application and fully test the machine in a standalone environment.

Because we have integrated unit testing into Architect an ID working on a state machine can periodically and very easily run all unit tests against the machine. If any unit test assertions fail the ID can investigate what recent changes may have broken the logic, or revise the unit test to reflect a desired change in the interaction and flow.

Dynamic Initial State

Applications deployed on the Android operating system may be killed and restarted by the OS. When this occurs it is the responsibility of the application to resume to match what the user was last doing. To support this, we support a custom `uic:initialexpr="…"` attribute on any state where an `initial="…"` attribute is valid.

The value of the attribute is evaluated as a Lua expression, and the result interpreted as a space-delimited set of state identifiers to target. This code-based state change *feels* like it makes the state machine less trustable, less precise. However, it is equivalent to an initial state with transitions leading to every possible combination of states, each guarded by a Lua condition determining if it is to be run. This feature does not change the functionality that is possible by the state machine; it simply makes the functionality possible in a more convenient manner.

We have similarly discussed adding support for a custom `targetexpr="…"` attribute to dynamically determine the state(s) targeted by a transition. As with `initialexpr`, this attribute should have no impact on the functionality possible with the machine, affecting only ease-of-use.

Remote Debugging

Our engine permits runtime debugging and introspection. The SCXML interpreter is able to communicate the active state(s) and current data model values over the network to Architect for live display during execution and debugging. Adding debugger support required only an additional 300 lines of C++ (not including transport protocol code).

Guarded Microstep Iteration

The official SCXML interpretation algorithm has an unbounded `while` loop that processes internal "microstep" transitions. Coupling this with a poorly designed state machine produces an infinite loop. Such a machine design is more likely than seems probable. We have repeatedly experienced a problem where an ID beginning work on an interface will create a pair of transitions between two states without taking the time to enter a triggering event for either. Consequently, as soon as one of those states is entered the state machine will unendingly switch between the two states as fast as possible.

To prevent this problem, and other more complex unstable configurations, our engine will only process a (large) fixed number of microsteps before moving on. While this value is currently fixed at 10,000 iterations we hope to make this configurable per state machine, in case the ID desires either a lower or higher limit.

Update-based Event Processing

The SCXML interpretation algorithm describes a main event loop that runs asynchronously from other systems, with a blocking call where it waits for events to process. Our engine instead runs synchronously, processing a queue of events until stable and then returning. This provides us

with a very predictable system, where we know that all events queued during one update frame will be processed before the next update renders to screen.

We hope to spend more time in the future researching real-time possibilities with algorithmic upper-bound guarantees on processing time.

Verified State Targets

All transitions that target a state are verified once before being taken to ensure that the referenced state id exists. Despite Architect preventing such a scenario, a user could hand-edit a SCXML file and enter an invalid state id. Further, this also guards against the possible case where the dynamic `initialexpr` Lua code returns invalid data.

CONCLUSION

Separating our interface development from interaction logic has simplified the development of complex applications. On a near-daily basis our in-house artists praise how much easier it is to control the interface from the state machine, and how much easier it is to find and fix user interaction bugs.

Our customers see using SCXML as the representation of the state machine as a benefit. As a text-based file format, it is amenable to storage and manipulation by source control systems. As an XML-based format, it can be understood and edited by humans and computers alike.

Using graphical statechart editing helps engage spatial reasoning, making interaction logic editing more accessible to visual artists. At the same time it prevents them from making many mistakes or typos that would produce an invalid state machine.

The graphical depiction of interaction logic provides an effective way to communicate with managers and other stakeholders about the high-level flow of an application. Because edits to this logic are immediately available in the application—instead of transcribing logic from a diagram into code—we have substantially reduced the time needed to test proposed changes and fix bugs.

We found that certain aspects of the SCXML format are harder to represent graphically, but these are rarely necessary in our experience.

We found that large portions of the SCXML standard are not necessary for it to be useful to our customers and us. At the same time, we have found the standard lacking certain features that we believe are either necessary or extremely beneficial to add.

Implementing SCXML support in C++ with a frame-based update engine enabled us to create a small, maintainable codebase that integrates well with our existing update-based interface system.

Using dynamic SCXML interpretation during application evaluation—instead of compiling the state machine to executable code and running that—enables us to provide debugging introspection about the current state(s) during development. This also reduces development time, enabling more, faster iterations on the application.

ACKNOWLEDGMENTS

We thank the entire UI Composer development team at NVIDIA. Without their hard work and attention to detail our endeavors would not have been possible.

REFERENCES

1. Harel, D. Statecharts: A Visual Formalism for Complex Systems. Science of Computer Programming, 8(3):231–274, 1987.
2. Samek, M. Introduction to Hierarchical State Machines. http://www.barrgroup.com/Embedded-Systems/How-To/Introduction-Hierarchical-State-Machines
3. State Chart XML (SCXML): State Machine Notation for Control Abstraction. http://www.w3.org/TR/scxml/
4. Xie, S., Kraemer, E., Stirewalt, R. E. K., Fleming, S. D., Huang, Y., and Dillon, L. K. On the benefits of UML 2.0 state diagrams on student comprehension of multi-threaded programs. http://cobweb.cs.uga.edu/~eileen/SE_Concurrency/state2/icse09Draft.pdf
5. Baker, P., Loh, S., and Weil, F. Model-Driven Engineering in a Large Industrial Context — Motorola Case Study. Model Driven Engineering Languages and Systems, 8th International Conference, MoDELS 2005, 2005.
6. Torchiano, M., Ricca, F., and Tonella, P. A comparative study on the re-documentation of existing software: Code annotations vs. drawing editors. International Symposium on Empirical Software Engineering, 2005.
7. Cooper, A. The Inmates Are Running the Asylum. Sams (1999), 168-170.
8. UML State Machine Diagrams. http://www.uml-diagrams.org/state-machine-diagrams.html#fork-pseudostate
9. Kistner, G. The Ruby XML StateChart Machine. https://github.com/Phrogz/RXSC
10. Kistner, G. The Lua XML StateChart Interpreter. https://github.com/Phrogz/LXSC
11. NVIDIA SHIELD. http://shield.nvidia.com
12. World Wide Web Consortium. Mobile SVG Profiles: SVG Tiny and SVG Basic. http://www.w3.org/TR/2003/REC-SVGMobile-20030114/

Multimodal Multi-Device Application Supported by an SCXML State Chart Machine

Nuno Almeida
DETI / IEETA
University of Aveiro
Portugal
nunoalmeida@ua.pt

Samuel Silva
IEETA
University of Aveiro
Portugal
sss@ua.pt

António Teixeira
DETI / IEETA
University of Aveiro
Portugal
ajst@ua.pt

ABSTRACT
The number of mobile and desktop devices available in the home environment is rapidly increasing with a notable emphasis on applications to serve AAL contexts. The coexistence of multiple devices and applications provides (and demands) new interaction possibilities, posing challenges regarding how different devices can be used simultaneously to access a specific application, taking the most out of each device features (e.g. screen size) and sharing input and output modalities.

To tackle these challenges, in a multi-device scenario, we propose a solution adopting the W3C MMI Architecture in which each device running the application runs an instance of the Interaction Manager (IM). Each instance can then act as an additional modality to the other IMs, allowing input and output events sharing. The proposed solution relies on an SCXML state-machine to define the application logic and communication: although sharing the same state-machine, each IM can be in its own state depending on the different combinations.

An application example is provided based on work carried out in the scope of Project Paelife, to illustrate the proposed solution applied to a multimodal multi-device enabled news reader.

Author Keywords
Multi device application;Multimodal interaction; SCXML

ACM Classification Keywords
H.5.m. Information Interfaces and Presentation (e.g. HCI): Miscellaneous

General Terms
Design.

INTRODUCTION
The use of mobile devices such as smartphones and tablets is widespread and has generated a strong demand for applications. The interaction of these devices with other common devices, present in our homes (e.g., television, Xbox), can enhance the way users interact with the different devices and their surroundings. This is particularly relevant in Ambient Assisted Living (AAL) scenarios, aiming to provide solutions that help people take the most out of these technologies, serving multiple usage scenarios and adaptability to different ages and disabilities.

The multitude of devices available in the home environment, for example, brings new possibilities into the way applications can be used, exploring each device characteristics (e.g., mobility, input/output modalities available) to provide a more versatile way of interaction. To this end, applications should allow users to interact with them not only in the common scenario of one device, but using the different devices available to the user. If users combine two or more devices, they should be able to use the more suitable modality or various modalities to interact with the application. The output modalities should be able to provide feedback in multiple ways, with the information presented in one device used to complement the other. For instance, a tablet can be used to interact with the application while the television provides a detailed view of some of the contents.

Developing an application to run in multiple devices presents a number of challenges concerning where the application logic will be instantiated and how to control and take advantage of the modalities available in the different devices.

In our work regarding multimodality [11, 10] we have adopted the W3C Multimodal Architecture. It is a loosely coupled and extensible architecture that supports multiple modalities and the distribution of modalities across multiple devices, such as PCs, tablets and smartphones. The architecture provides flexibility, allowing to change or add modalities in the system without the other components being aware.

To address the challenges of a multimodal multi-device application we propose that each device running the application runs an instance of the Interaction Manager (IM). Each of the IM instances can be in a different state depending on how the application is being used and what is the main purpose of each device at each time. In our approach the different IMs work as an additional modality to the remaining IM instances allowing input and output events sharing with the SCXML state-machine managing the parallel use of the different devices.

EICS 2014 Workshop: Engineering Interactive Systems with SCXML, June 17, 2014, Rome, Italy Copyright is held by the author/owner(s)

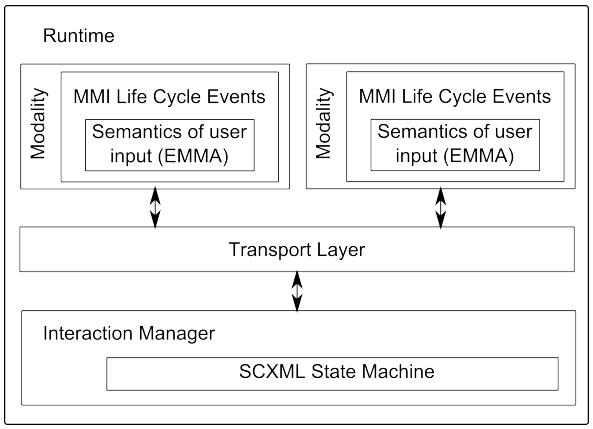

Figure 1. Multimodal Architecture

This approach to multimodal multi-device applications has been put to test in the scope of Project Paelife [10] from which we extract an application example to illustrate its main aspects.

This article is organized as follows: Section 2 briefly presents background and related work. Section 3 describes our application scenario and the cases of use for a multi device scenario application. Section 4 describes the method used to accomplished the utilization of application in a multi device scenario. Section 5 presents our application working in two different scenarios. Section 6 presents the conclusions and ideas for further work.

BACKGROUND AND RELATED WORK

The W3Cs Multimodal Architecture and Interfaces recommendation [3] is based on a Model-Viewer-Controller (MVC) design pattern. It can be divided into four major components (illustrated in Figure 1):

- Modality Components – provide user interaction capabilities, input or output: the Viewer in an MVC paradigm;
- Runtime Framework - offers communications capabilities for the modules of the system and acts as a container for the other components;
- Data component - represents the data model stored in the system: the Model in an MVC paradigm;
- Interaction Manager - acts as a state machine that manages the different modalities: the Controller in an MVC paradigm-

Considering the component most relevant to the presented work, the Interaction Manager (IM) of the multimodal framework is configured by a state machine described in the W3C SCXML.

The State Chart XML (SCXML) [1] is a general-purpose state machine notation based on events. It merges concepts from CCXML and Harel State Tables. The general characteristics of the SCXML and its extensibility enable that it can be used for different areas and purposes. A considerable amount of literature has been published on the W3C SCXML recommendation regarding its use in a number of contexts.

The SCXML not only defines the state-machine itself but also a data model which, in addition to storing active states, also provides a model to store other useful information. It features powerful state flows by allowing parallel states and substates. It adds a number of extensions to a basic state machine and the capability to execute content such as conditions, executable scripts, send messages to external entities and modify the data model. This is accomplished through two elements that execute content upon entering or leaving a state.

Researchers commonly refer to SCXML state charts in the context of multimodal interaction [3, 4]. Its use extends to mobile devices, smart homes, robots and AAL contexts. Although SCXML has a vast list of applications, the multimodal interaction architecture contributes to increase its use by proposing the SCXML state charts for the core of the Interaction Manager.

SCXML state charts are also used in dialog management [7, 9, 5], to describe the flow of conversation, and in ubiquitous computing [8, 6], where it is being explored to manage smart space environments, enabling the control of interaction across sensors and actuators.

Several implementations of the SCXML can be found for different environments. One of the most frequently used implementation is the Apache Commons SCXML[1] developed for Java, but other implementations are available including scxmlcc[2] and uscxml[3] for C++, PySCXML for Python[4] or SCION[5] for JavaScript. In addition to the SCXML implementation, there are tools available to create or edit state charts. The scxmlgui[6] is a graphical tool for creating and editing state charts offering interaction with the Apache Commons SCXML providing visual feedback of what is happening inside the state machine.

APPLICATION SCENARIO

Although the method proposed to create a multimodal multi-device application can cover several contexts, our scenario for developing a multimodal multi-device application focuses on aspects of the PaeLife project. Paelife is an European collaborative project between industry and academia and its goal is to contribute to an active ageing of the elderly by creating or adapting technologies facilitating its use by these persons.

In general, Paelife Personas are older adults with more than 60 years, which have some degree of experience with computers, although they may not be proficient using them. The Personas spend most of the time at home and for this reason a big screen would be an appropriate choice for the user to interact with an application. However, a portable device such

[1] `http://commons.apache.org/proper/commons-scxml/`
[2] `http://scxmlcc.org`
[3] `https://github.com/tklab-tud/uscxml`
[4] `https://github.com/jroxendal/PySCXML`
[5] `https://github.com/jbeard4/SCION`
[6] `https://code.google.com/p/scxmlgui/`

as a tablet would give the user the possibility to use the application in other situations.

To design our application we have adopted the proposal of [2] for creating a model of use, combining system goal expression, application type, information on users, tasks, devices, modalities, environment and interaction. Regarding interaction we consider multimodal interaction, having graphical and speech outputs and speech, gestures and touch inputs. This modalities are based in technologies available from previous research and development by partners of the project Paelife

One important goal of our work, described in this article, is the use of applications in a multi-device scenario with the most typical case comprising a static main unit connected to a television and a mobile unit (tablet), each of which independent but simultaneously interoperable. The interface should be as similar as possible in both the units, thus making it easier to use since only one interface needs to be learned. In this scenario interaction can be performed in three different ways: a) through the main unit; b) through the mobile unit; and c) through both the main and mobile units.

Interaction through the main unit
In this scenario, the user only uses the main unit to interact with the system. This way there is, for now, two interaction modalities: voice and gestures.

The interface of the main unit must be simple enough to ensure that the interaction through voice and gesture modalities is achieved in a simple and natural way. Given the various types of services that need to be provided, the main screen interface may be a dashboard that shows, in real time, personal information relevant to the user.

Interaction through the mobile unit
In this mode the user will interact only by using the input modalities available on the tablet, particularly touch and graphical output. If the user is close to the main unit the output modalities of the main unit can also be used, if needed, as detailed in the following case.

Interaction using integration between main unit and mobile unit
This mode will take advantage of the interaction between the main unit and the mobile unit to improve the usability of the system and implement new features.

For example, when detecting the user is within the range of the main unit, the application can allow using the main screen to visualize content while using the tablet as a controller.

When the user interacts simultaneously with both units, the number of possible ways to interact increases. In this case, we consider three content presentation modes, not mutually exclusive:

- The main unit displays content and the mobile unit is used as input;
- The main unit displays the same content as the mobile unit;

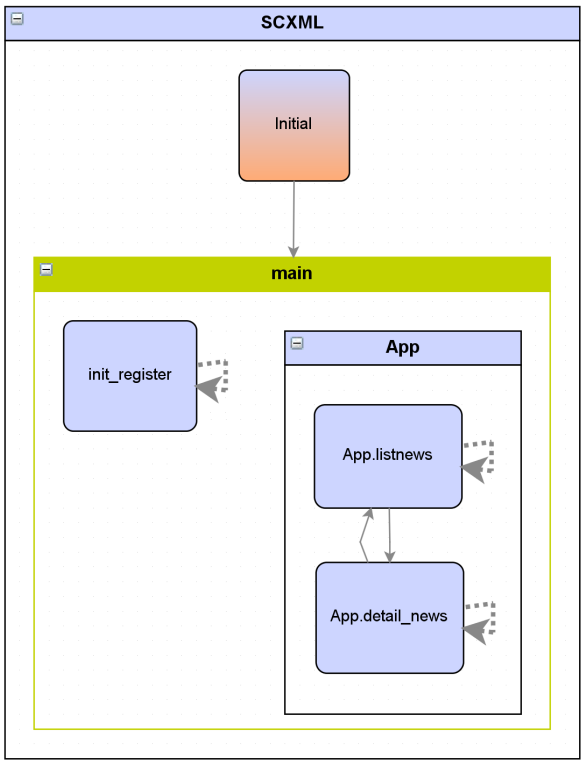

Figure 2. SCXML states chart for a news reader application

- The main unit displays the main content and the mobile unit displays secondary content.

The system should automatically select the most appropriate mode for each application/task the user is performing. However, the user should be able to freely switch between the various modes supported by the running application.

METHODS

To accomplish multimodal interaction in our application we have adopted the multimodal framework described by the W3C. The multimodal framework is a set of modules and one among the most important is the Interaction Manager (IM). Modalities are only allowed to communicate with the IM: the input modalities send MMI lifecycle events and the output modalities receive events from the IM. In the architecture the IM and data model are two distinct modules, but in practice they were implemented in the same module.

The implementation of the IM uses the Apache Common SCXML to manage the state machine defining the application logic. We extended the use of the SCXML to parse multimodal architecture lifecycle events and trigger them into the state machine. The extension also includes the generation of the lifecycle events to be transmitted to the modalities.

For the communication between the modules we implemented a simple HTTP server in the IM capable of receiving lifecycle events from the HTTP POST method. The method imlemented for the $< send >$ element of the SCXML, works in two ways. If the modality also has an HTTP server the IM creates a lifecycle event and sends the message to his

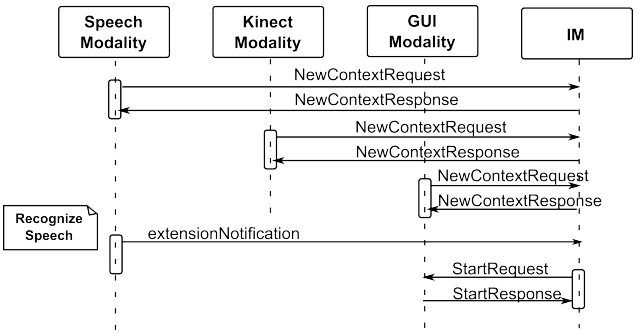

Figure 3. Sequence diagram of the communication between modalities and Interaction manager, showing the lifecycle events when the modalities start and after a generation of a speech event.

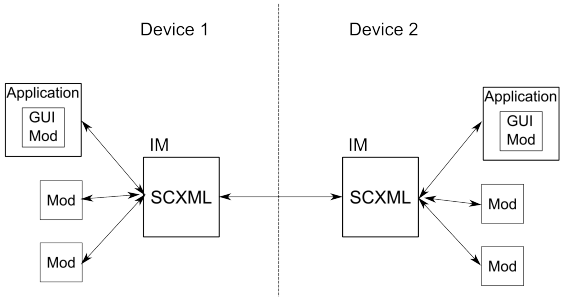

Figure 4. Collaborative use of two instantiations of the applications

server. In the cases for which the modality cannot implement an HTTP server, it makes HTTP GET requests and waits for a message from the IM.

For the design of the SCXML state machine, a data model is included that stores the identification of the available modalities, last performed action and other parameters regarding the last lifecycle event received. The state machine starts with a parallel state, owning two sub-states. The first sub-state is expecting for the registration of modalities, then the data model is updated to store the identification of a new modality. The second sub-state concerns the application logic and it contains other sub-states regarding the different contexts of the application. Application states should be able to change the data model, and create lifecycle events to send to modalities. Figure 2 depicts an overview of the states created for the news reader application presented in the following section.

The current framework includes body gestures, touch and speech input and output. Input modalities generate events that are coded into an EMMA message format and sent to the IM through lifecycle events.

APPLICATION

In what follows we briefly describe work carried out in the scope of project Paelife illustrating the methods proposed in this article and covering the single and multi-device scenarios.

One device scenario

The created application is a news reader, developed for Windows 8, which provides multimodal interaction aiming to offer a better user experience and usability compared to WIMP interfaces. The applications starts by loading the RSS news feeds according to the user's language. The main screen of the application shows a grid view with the loaded news. At the same time the feed is processed to configure a new grammar for the speech modality with the news headlines.

The graphical output modality is embedded in the application and is continuously listening for lifecycle events coming from the IM. The modality is responsible for changing the application view, opening the content of an article or going back to the previous screen.

Any input modality can register in the IM to send events and the generated data should provide semantic information. The semantic information follows a set of rules that are defined when the application is created. The semantics are interpreted to perform an action in the output modality.

The user interaction with the news reader, to read the entire content of the news, can be made by speech or touch. The user can utter the headline of the news or tap the corresponding square on the grid. They can choose one of the available modalities to interact, for example, in the main unit the user would use speech and in the portable (tablet) they would use touch. The same is applied for other actions such as go back to the grid view, by tapping a back button or saying go back.

Figure 3 shows a sequence diagram depicting an example of interaction with the application made using speech. The diagram shows the modalities registering in the beginning with the lifecycle *NewContextRequest* and the data model is updated, acknowledging the availability of the modalities. After the speech modality recognizes a command, it sends the event to the IM and then the IM sends a request to the modality to show the corresponding information. If the command is to visualize the content of a news item, the state machine changes its active state to *detail_news*, else it stays in the current state.

Multi-device scenario

The creation of the multi-device application is actually an update to the single device application. With this update the application now supports two different ways of presenting information, the first is the same as the one device application, the second shows images associated with the news in full screen. This way we can perform combinations of the visualization for the multi device application. Possible combinations regarding news content viewing are:

- Main unit showing the content / mobile unit also showing the content
- Main unit displaying full screen images / mobile unit showing the content
- Main unit showing the content / mobile unit doesnt change and presents the list of news.

Other combinations could be accomplished, but these options seemed the most appropriate for our goals.

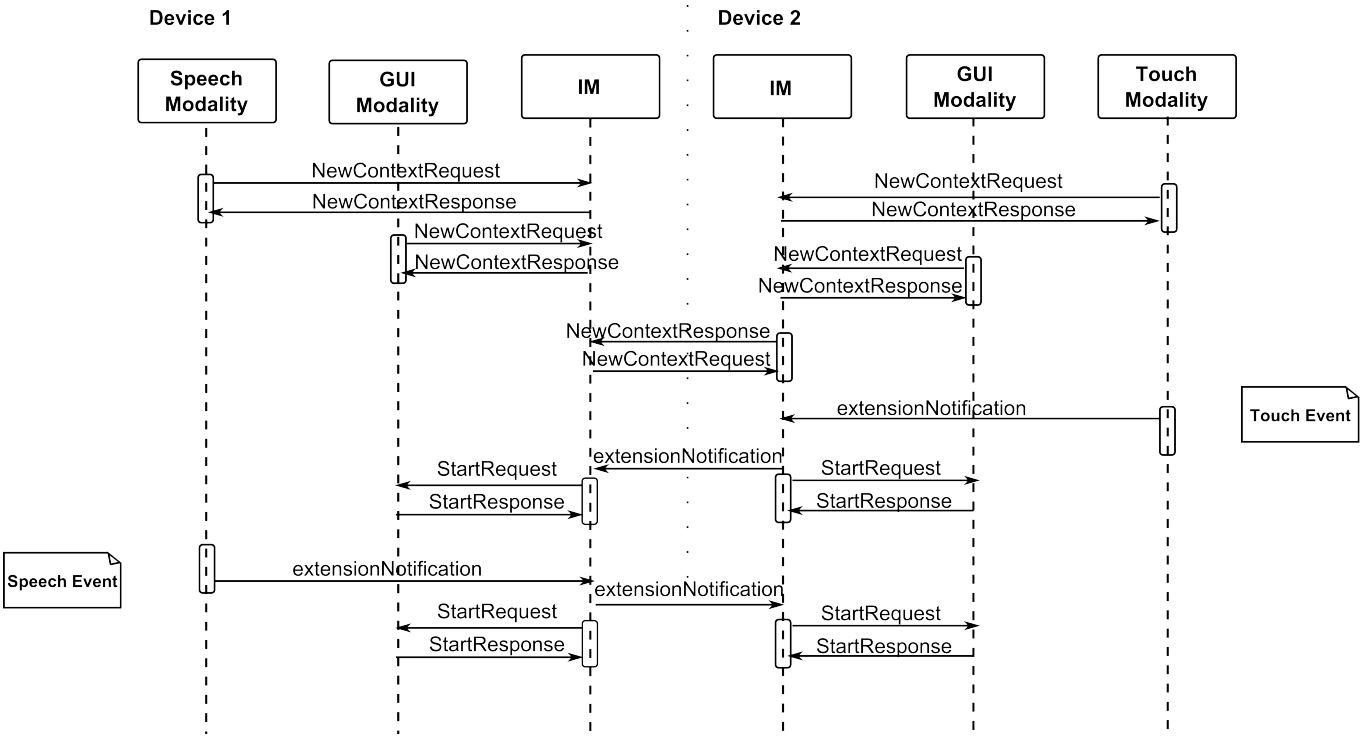

Figure 5. Sequence diagram of the communication between modalities and the interaction managers of each device, showing the lifecycle events when the modalities start and after a generation of a touch event.

Because we want to have the possibility of running the application on both devices, interacting with both at the same time, the application and IM are the same. The support for this connected mode of operation was attained by an evolution of the IM used for the single device scenario: the IM operating locally in the device was updated to send messages to other devices. Figure 4 shows how the system is connected, with the IMs in each device providing data to the other.

The data model of the SCXML stores the availability of other IMs and the mode in which that device is operating. The mode indicates if the application either shows the content of the news, the full screen image or if it does not change its state. In each state a condition was added to verify if a received event should be sent to the other IM.

The IM can now receive events of the other device and it sends lifecycle events for the output modalities connected to him. Figure 5 shows a sequence diagram of the application running in two devices and depicting an example in which the interaction with the application is performed through the second device using touch and through the first device using speech.

The IM knowing the current mode of the device that is running, the sent messages carries this information for the output modality. Figure 6 displays screen shots of the developed application, with the possible ways for presenting the information to users. On top, the figure shows the two devices displaying the information in the same way, in the middle the tablet shows the content with a small image and the TV only shows the image, in the bottom the TV shows the content and the tablet the grid of news. In this last possibility the user can read the current news on the TV and use the tablet only for selecting the news.

If, by any chance, one device is no longer available, the corresponding parameter of the data model in the state machine changes to signal this fact and the user can continue to interact with the application using the device that is still available.

CONCLUSION

In this paper we propose a method to develop multi device applications that can work on different devices. The interaction between applications is accomplished by using a multimodal framework which manages the communication between the modalities and the communication between the two interaction managers. A SCXML state machine controls the flow of the exchanged messages coming from the modalities and the other interaction managers to the output modalities.

An application example is provided, based on work carried out in the scope of Project Paelife, in which we show different combinations to visualize the information presented by a news reader application to the user, while interacting with it using two devices.

In the future our goal is to develop a method to determine if the two application are in condition of being used jointly: if the state machine is aware that the devices are in different places, each application should run separately.

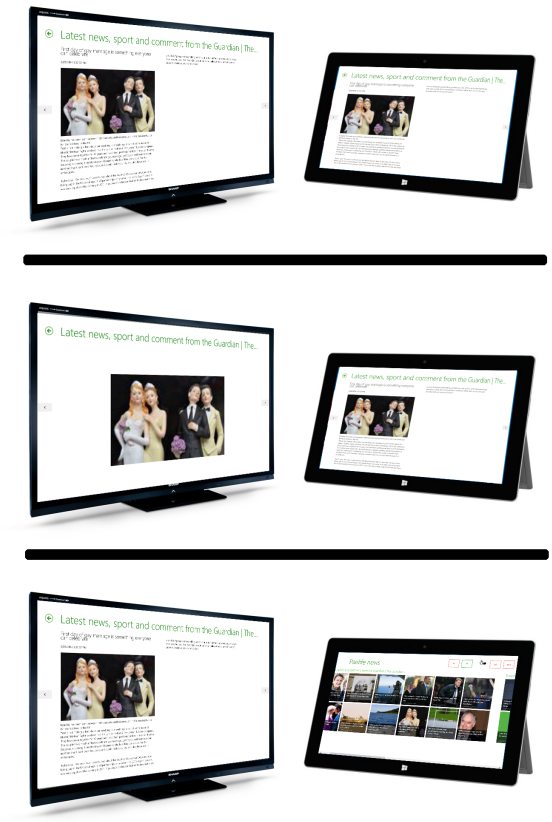

Figure 6. Screen shoots of the application with different view mode combinations: Top, both applications show the same content; Middle, the TV shows only the image and the tablet the news content; bottom, the TV shows the news content and the table presents the grid with all the news.

ACKNOWLEDGMENTS

The work presented is part of the COMPETE Programa Operacional Factores de Competitividade and the European Union (FEDER) under projects AAL4ALL (www.aal4all.org); Part of the work presented was funded by FEDER, COMPETE, FCT and AAL Joint Program in the context of PaeLife (AAL/0015/2009); IEETA Research Unit funding FCOMP-01-0124-FEDER-022682 (FCTPEstC/EEI/UI0127/2011) and project Cloud Thinking (funded by the QREN Mais Centro program, ref. CENTRO-07-ST24-FEDER-002031).

REFERENCES

1. Barnett, J., Akolkar, R., Auburn, R., Bodell, M., Burnett, D. C., Carter, J., McGlashan, S., Lager, T., Helbing, M., Hosn, R., et al. State chart XML (SCXML): State machine notation for control abstraction. *W3C Candidate Recommendation 13 March 2014* (2014).
2. Bernsen, N. O., and Dybkjær, L. *Multimodal usability*. Springer, 2009.
3. Dahl, D. A. The W3C multimodal architecture and interfaces standard. *Journal on Multimodal User Interfaces* (Apr. 2013).
4. Feuerstack, S., Colnago, J. H., de Souza, C. R., and Pizzolato, E. B. Designing and executing multimodal interfaces for the web based on state chart XML. In *Proceedings of 3rd Conferência Web W3C Brasil* (2011).
5. Gandhe, S., Taylor, A., Gerten, J., and Traum, D. Rapid development of advanced question-answering characters by non-experts. In *Proceedings of the SIGDIAL 2011 Conference*, ACL (2011), 347–349.
6. Harrington, A., and Cahill, V. Model-driven engineering of planning and optimisation algorithms for pervasive computing environments. *Pervasive and Mobile Computing 7*, 6 (2011), 705–726.
7. Morbini, F., DeVault, D., Sagae, K., Gerten, J., Nazarian, A., and Traum, D. FLoReS: A forward looking, reward seeking, dialogue manager. In *Natural Interaction with Robots, Knowbots and Smartphones*. Springer, 2014, 313–325.
8. Rouillard, J., and Tarby, J.-C. How to communicate smartly with your house? *International Journal of Ad Hoc and Ubiquitous Computing 7*, 3 (2011), 155–162.
9. Skantze, G., and Al Moubayed, S. IrisTK: a statechart-based toolkit for multi-party face-to-face interaction. In *Proceedings of the 14th ACM international conference on Multimodal interaction*, ACM (2012), 69–76.
10. Teixeira, A., Hämäläinen, A., Avelar, J., Almeida, N., Németh, G., Fegyó, T., Zainkó, C., Csapó, T., Tóth, B., Oliveira, A., et al. Speech-centric multimodal interaction for easy-to-access online services–a personal life assistant for the elderly. *Procedia Computer Science 27* (2014), 389–397.
11. Teixeira, A. J. S., Almeida, N., Pereira, C., and e Silva, M. O. W3C MMI architecture as a basis for enhanced interaction for ambient assisted living. In *Get Smart: Smart Homes, Cars, Devices and the Web, W3C Workshop on Rich Multimodal Application Development* (New York Metropolitan Area, US, July 2013).

Transforming a State Chart at Runtime

David Junger
University of Gothenburg
Göteborg, Sweden
tffy@free.fr

ABSTRACT
This paper proposes mechanisms allowing an SCXML interpreter to behave consistently and sensibly when its State Chart is being modified at runtime. A partial implementation, powering a graphical SCXML editor/ debugger, will serve as an illustration and proof of concept.

Author Keywords
State Chart; SCXML; DOM; dynamic; editor

ACM Classification Keywords
D.2.2 Design Tools and Techniques: State Diagrams
D.3.4 Processors: Runtime environments

General Terms
Design

INTRODUCTION

If SCXML is indeed to become the "HTML of multi-modal applications", what will be the HTML DOM of multimodal applications?

SCXML is a markup language for representing Harel State Charts (SCs), a powerful formalism to design and run application control and particularly User Interface (UI) control. It is possible to generate SCXML documents with XSL transformations or scripting with the generic XML Document Object Model (DOM)[2] before starting the interpretation.

But SCXML can become more flexible by exposing a specialized DOM API, like HTML does, and handling the complexity (and dangers) of modifying the SC while it is running.

USE CASES

Work is in progress on an SCXML editor and debugger [3] (which I hope will be interesting in itself) that relies on the proposed DOM behaviors, as a proof both that they work and that they are useful.

There are also cases directly related to UI applications of SCXML. If you're writing a multimodal application that wants to talk to an arbitrary number of users, that number being known only at runtime, then how do you create the manager states for each user? How about a modular UI that wants to spawn the same module many times, dynamically (think of tracks in audio editing software)?

How about artificial intelligence? Weighted transitions are already a wanted feature for SCXML, and the (quite safe) ability to modify their probabilities at runtime would enable learning. A much smarter system might want to evolve by actually changing transition targets and performing other structural operations:

– Hello sir, welcome back. Tea?
– Why don't you just ask me about my day?
– Oh… *(rewrites some transitions for event=user.enters)*
So, how was your day?

That is essentially the user editing the SC at runtime, but using natural language instead of a traditional GUI.

SCXML PROCESSING ALGORITHM

The complete algorithm is described in the SCXML candidate recommendation[1]. This paper cannot list every detail relevant to every DOM mutation.

The short version is that the SCXML interpreter will attempt to take as many transitions as it can, repeatedly until there is no more event to process and no more eventless transition it can take. It will then enter a stable configuration, waiting for more events.

The process of selecting transitions and taking them is called a *microstep*. It is divided in five phases:

- enabling transitions (finding transitions in active states that might be taken in the current context), possibly by consuming events
- preemption (eliminating conflicts between those enabled transitions)
- exit (removing states from the configuration and running their *onexit* handlers)
- running the transitions' executable content
- entry (adding states to the configuration, possibly some that were just exited, and running their *onentry* handlers)

Taking microsteps until a stable configuration is reached is called a *macrostep*. At the end of a macrostep, states that have been entered during the macrostep may invoke sub-processes which can live until the state is exited again.

EICS 2014 Workshop: Engineering Interactive Systems with SCXML, June 17, 2014, Rome, Italy

MUTATION TIMING

Interpreters supporting runtime mutations should prevent or defer those mutations while performing a microstep. This is important for pragmatic reasons (performance, complexity), but most importantly because accepting mutations mid-microstep leads to race conditions.

Imagine that multiple transitions are selected based on their compatible targets. Right after the preemption phase, a modification to their targets causes a conflict between them. The algorithm for the entry phase assumes any conflict has already been avoided, and so the interpreter enters an invalid configuration.

Pausing the Interpreter

In order to facilitate mutation timing, a supporting interpreter should allow its execution to be paused between microsteps. The pause should be triggered automatically when detecting a mutation that renders the SC invalid. In any case, the interpreter pauses only at the end of the current microstep, not necessarily when the pause is requested. Thus, it needs to somehow notify interested parties that it is paused.

When possible, applications should request a pause only after the current macrostep, to further reduce the impact on complexity and performance.

SCXML-controlled UIs that rely on runtime SC mutation should ensure that the pause is as short as possible. In particular by fetching resources in advance before pausing the interpreter.

Self-Mutation

Executable content runs during, not between, microsteps. Therefore, in order to change the SC's DOM through scripts or extensions within the SC itself, either the interpreter will accept mutations at any time but defer their application, or the script must define the mutations in a callback associated with a pause request. A simple extension would be a *script* element that automatically wraps its content in such a callback and requests a pause.

Delayed events and other timers

As a side-effect of pausing the execution of the SC, in order to preserve functionality, delayed *send*s[1.2] and other timers set by executable content (such as with the setTimeout method in ECMAScript) should also be paused.

PRINCIPLES

The proposed rules are based on these two guidelines:

Legal Configurations

The interpreter should ensure that the mutated SC is in a legal configuration[1.1] after the mutation. It may be impossible to reach that configuration by normal means, but the algorithm does not need to know that to keep running.

No Time Travel

The actions taken because of freshly deleted elements could be impossible to reverse, and it may be hard or impossible to simulate all that would have happened, had the SC been mutated from the start. This proposal does not address those issues (but see **Input Replay** near the end).

GENERAL RULES

An implementation that supports runtime mutations should detect when the SC becomes invalid, and pause execution until another mutation makes the SC valid again. It should let the mutation's originator know that interpretation cannot proceed, and may allow the mutation to be rolled back. The interpreter must not reject those mutations, because some changes to a valid SC will often go through a transitory invalid SC, as illustrated in Fig. 1-3:

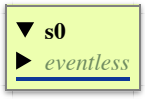

Fig 1. A "blank" SCXML document: a targetless transition in a state could be your editor's starting point.

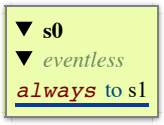

Fig 2. The transition is now targeting a state, but s1 has not been added yet, making the SC invalid.

Fig 3. The missing target is added, and the SC can resume.

In the example, it is possible to switch steps 2 and 3 to produce the same SC without going through an invalid step. In fact, it should be possible to find a way to build any SC by adding states and transitions one by one without ever creating an invalid document.

But in practice, the valid State Chart with an unreachable state is about as useful as the invalid one with a missing transition target, so we might as well show some tolerance for authors and scripts who like to start with transitions and create target states later, or clone a whole piece of a State Chart (thus duplicating state IDs) and work from there rather than rebuild it from scratch.

Dynamism and Compiling

Most existing implementations use compilation to improve speed, often relying on a static SC structure. Even the reference algorithm assumes that the configuration does not change between the end of a microstep and the exit phase of the next. Frequently modifying SC structure could severely degrade the performance of interpreters. There are several ways to mitigate that:

- allowing the SC to be mutated in batch, similar to a database's transaction mechanism, and deferring the calculations until all mutations have been performed (as a bonus, allow all the changes to be rolled back)
- change the algorithm to rely less on compilation (but too little would degrade performance in its own way)
- use lazy compilation / memoization
- compile in a way that optimizes mutation performance, e.g. by making it very fast to detect the impact of a mutation and decide what has to be recompiled

- begin to apply the mutations on a copy, in parallel to the execution of the SC

The JSSCxml interpreter uses both lazy compilation and mutation-optimizing compilation at this time.

SC MUTATIONS TO WATCH FOR

A State Chart is a graph whose nodes are states and arcs are transitions. As far as the SC's structure is concerned, the mutation of other elements, and secondary attributes of transitions, is perfectly safe between microsteps. These are all the mutations that matter (SC-breaking conditions are written in bold letters):

Transition Targets

An **invalid target** automatically means an invalid SC. Transitions inside a *history* or *initial* element can become invalid even if their target exists, when that **target lies outside of the element's parent state**.

Transitions' *internal* attribute

Switching the *internal* attribute of a transition may require recalculating its LCCA[1.3] for implementations that compile or memoize it.

Inserting and Removing Transitions

Removing a (targeted) transition probably involves some compilation but is otherwise safe. However, **creating a new transition to a non-existing target** would break the SC.

History and *initial* elements should be created with their mandatory transition built-in and targeting the default substate(s), similar to the way HTML *table* elements create a *tBody* when one is not explicitly written. Adding another or removing it should be forbidden; instead, it can only be modified or replaced.

Replacing Transitions

In the case of transitions, replacement can be safely reduced to removal + insertion.

Changing State IDs

State IDs are assumed to be unique and **renaming a state to an existing ID** would make the SC invalid. Otherwise, renaming a state will always render the SC invalid **if any transition was targeting the state's former ID**.

When detecting the mutation, a smart interpreter may offer the choice of refactoring any transition that previously targeted the renamed state so they keep targeting it with its new ID and the SC remains valid.

Inserting and Removing States

As with renaming, **deleting a state which was the target** (or contained a target) **of an external transition** will break the SC until the transitions are fixed.

Removing a state also means removing it from the configuration if it was active, canceling any invocation in it and its descendants.

Adding a state within an active parallel state, or an active atomic state, would cause that state to be entered immediately (including running its *onentry* handlers, and starting its invocations if the configuration is stable).

Replacing states

For states, there is something to be gained by treating replacement as more than removal + insertion: the replacement state can be targeted by all transitions targeting the former state, avoiding an intermediary invalid SC. However, replacing a state of one type with a state of another type is trickier than it may sound (see below).

State Type

The DOM generally does not allow a node's name to be modified. But implementations may provide a method to do so, while internally creating a new element to replace the existing one and putting the former's children in the new.

Changing a compound state into a parallel state, or vice-versa, would instantly make the SC's configuration illegal if they have more than one substate. Therefore, a newly parallel state should have all its inactive children entered. A parallel state becoming a regular compound state should cause all its children but the initial one to be exited. Whether their *onexit* handlers should run or not is debatable, but certainly their invocations need to be cancelled.

Changing **a non-atomic state into a final state** would make the SC invalid. Atomic states becoming final could also make the SC invalid, but in a relatively harmless way, and it can keep running as long as the final state's transitions are ignored and invocations cancelled.

If it was an active child of <scxml>, making it final should terminate the SC. If it was an active substate, its parent should raise a *done* event as soon as the SC resumes, and its parallel ancestor(s) may do so too if their other final descendants are active.

If a top-level final state is changed into a regular or parallel state, then the SC is already terminated and this is no longer a *runtime* mutation. A final state at another level becoming a regular state would stop ignoring any transitions and invocations it had.

Moving a State

Moving a state would entail some of the issues of adding and deleting states: entering or exiting the state as a result of its new position, and breaking the SC **if the state is moved to a final parent**. However, transitions targeting the state do not generally become invalid, although external transitions to it or its descendants must be recompiled, and **initial or default history transitions will likely become illegal**. Transitions from the state and its descendants targeting external states will require similar attention.

If an active state is moved to a position where it should still be active, such as under a parallel or atomic parent, then it should not be exited and re-entered. Instead, it and its active descendants simply remain active and their invocations keep running.

Invocations

Although they have no bearing on the SC structure, invocations need special attention because they are the only thing running between micro- and even macrosteps. Therefore, some mutations concerning *invoke* elements

should have immediate consequences. Obviously, removing an *invoke* in an active state entails canceling the invocation. A new *invoke* added to an active parent should be started immediately if the configuration is stable, or marked to be invoked once it becomes stable, as if its parent had just been entered.

The case of changing an active invocation's source is more delicate. Should the invocation be restarted immediately with its new source? the safer option is to let it run, and let the mutation's origin explicitly restart it if that is what it wants. Restarting the invocation without waiting could be further complicated if it relies on executable content in its parent to set initial parameters: that content will not be executed again until the state itself is re-entered.

REVISING HISTORY

Whenever a state mutation occurs, the ancestors of the mutated state (in the case of moving, both the former and, to a lesser extent, the new) will need to revise their recorded histories to make them legal (not to mention, default history transitions may break). Only immediate parents need to worry about shallow history, but every ancestor's deep history could be affected.

A history record that includes a removed state may need to replace it with one of its former siblings; and a recorded deep history that goes through a parallel parent of an added state will have to add the new substate's active or default atomic descendants.

RELATED AND FUTURE WORK

Relative Targeting

SCXML's global namespace for IDs can become a problem when trying to duplicate states without writing extra code to set unique IDs for them and refactor transitions to them.

A proposal that can be useful on its own, relative targeting means that instead of naming target IDs, a transition could indicate its hierarchical relation to its target state(s), e.g. <transition path="../state[2]"/> targets the third substate of the transition's parent state without knowing its ID.

Input Replay

A method used in live coding runtimes consists of recording all input to the system and, when a mutation occurs, starting the mutated system from the beginning and feeding it all the recorded input as fast as possible to see how the mutation affected the entire system.

Such a method can be particularly useful for UI, and certainly the no-time-travel principle in this paper does not mean that input replay is not a good idea. Only that it is not *always* a good or practical idea. When relevant (especially in a development environment), it would be a welcome addition to an interpreter's dynamic capabilities.

Dynamic SCXML and Static Analysis

One of the appeals of State Charts is their well-defined mathematical structure, which allows algorithms to *prove* certain properties of a SC. A dynamic SC is not, in general, automatically translatable to a static SC.

However, since the interpreter is paused whenever a mutation is being processed, and always resumes in a valid configuration, it can be said that the interpreter effectively runs a sequence of static SCs with hidden final states and top-level *initial* attribute.

While the most powerful uses of dynamic SCXML would result in nondeterministic sequences of SCs and initial configurations, it is also possible to use the DOM purely for its expressiveness over writing an equivalent, but larger and/or uglier, static SC. In the middle ground, the sequence of SCs is not easily predictable but has some predictable properties. For example, duplicating a substate may result in an infinite SC, but in a very regular way.

CONCLUSION

SCXML mutations may not be as easy to handle as HTML mutations, but a combination of reasonable timing constraints, helpful feedback for dangerous operations, and a minimal set of housekeeping rules can make it work.

The implementation of those proposals will be more or less demanding depending on each SCXML interpreter's implementation and the desired performance profile. Optimizing a dynamic SC and optimizing its mutation handling is a harder problem than compiling static SCs.

If the SCXML community finds DOM mutations useful, it is in everybody's interest to arrive at standard interface definitions and rules, then perhaps also a reference algorithm for implementing those rules. This paper, based on experimental work and strongly inspired by current Web standards, is written to initiate that effort.

ACKNOWLEDGMENTS

I couldn't present this paper without the financial support of the Centre for Language Technology in Gothenburg. The CLT also funded some of the development of JSSCxml.

Many thanks to the participants on the W3C's Voice Browser Working Group mailing list.

REFERENCES

1. *State Chart XML (SCXML): State Machine Notation for Control Abstraction*, W3C Candidate Recommendation http://www.w3.org/TR/scxml/

1.1. Legal State Configurations and Specifications http://www.w3.org/TR/scxml/#LegalStateConfigurations

1.2. <send> http://www.w3.org/TR/scxml/#send

1.3. 'Type' and Transitions http://www.w3.org/TR/scxml/#N101AA

2. *DOM*, WHATWG Living Standard http://dom.spec.whatwg.org

3. The JSSCxml interactive editor http://www.jsscxml.org/viewer

A Debugger for SCXML Documents

Stefan Radomski
TU Darmstadt
Telecooperation Group
radomski@tk.informatik.tu-darmstadt.de

Dirk Schnelle-Walka
TU Darmstadt
Telecooperation Group
dirk@tk.informatik.tu-darmstadt.de

Leif Singer
University of Victoria
Victoria, BC, Canada
lsinger@uvic.ca

ABSTRACT

The development of non-trivial applications as SCXML documents entails the requirement for application authors to verify and retrace their execution semantics and behavior. As of now, there are no tools available to debug SCXML documents as one would debug e.g. a Java or C program. In this paper we outline an approach to map established idioms for debugging onto the interpretation of SCXML documents, enabling document authors to break and step through their interpretation and to inspect the interpreter.

Author Keywords

SCXML; Harel State-Chart; Debugging; Developer Support

ACM Classification Keywords

D.2.5. Testing and Debugging: Distributed debugging

INTRODUCTION

With SCXML getting ready for recommendation status by the W3C, the need for accompanying infrastructure such as interpreters and debuggers is gaining relevance. While there is already a selection of interpreters to choose from, only few authoring environments and, to the best of our knowledge, no debuggers for SCXML documents are available.

In keeping with the spirit of open standards, this paper proposes a simple yet functional HTTP-based protocol to debug SCXML documents. By aligning the protocols concepts with the execution semantics as outlined in the SCXML draft, we hope that interpreter developers will have minimal effort to implement and support this approach.

We implemented the protocol as part of our interpreter and provide a HTML-based user interface running in a browser.

RELATED WORK

"Debugging sequential programs is a well understood task that draws on tools and and techniques developed over many years" [4]. Usually, the program is repeatedly stopped during execution where developers can then examine the current state of the program. Then, they can either continue the execution or restart to stop at an earlier point in the execution. This approach is also known as *cyclical debugging* [5]. This is mainly done to locate and fix code that is "responsible for a symptom violating a known specification" [2].

A study of Eisenstadt [1] showed that 50 percent of the problems associated with debugging have their roots in inadequate debugging tools. Hence, many vendors started to work on integration of debugging tools into IDEs and visualization of the underlying program construct. Hailpern [2] also mentions efforts in automatization of debugging through program slicing [3].

For SCXML there are a few tools, mostly developed as part of the various interpreter implementations, but none will allow to set breakpoints and to inspect the datamodel.

DEBUGGING TECHNIQUES

Without a dedicated debugger for SCXML documents an application author has to resort to (i) insert `<log>` elements as executable content in `<transition>`, `<onentry>` and `<onexit>` elements, or (ii) even step through the interpreters implementation as the SCXML document is evaluated.

The first approach amounts to *println-debugging* and requires careful preparation of the log statements by the developer to achieve a balance between traceability and intelligibility while coping with the potentially huge amount of messages. A major drawback of this technique is the time it takes to identify the places where log statements will help to reveal the problem and the fact that they are most likely deleted after the problem was resolved to improve readability of the SCXML document, causing every debugging session to start anew.

The second approach takes advantage of the fact that the interpreter itself is likely written in a language for which mature debugging tools already exists. These can be used to inspect the interpreter while it is evaluating an SCXML document but requires an in-depth understanding of an interpreter's implementation and diverts the focus from the actual SCXML document.

When we look at established debuggers such as GDB or the Java debugger and their various graphical frontends, the predominant concepts to debug programs is via breakpoints and variable inspection. An application developer sets a breakpoint at a line of source code and control flow halts

EICS'14 Workshop, Engineering Interactive Systems with SCXML, June 17, 2014, Rome, Italy

upon reaching it, allowing developers to inspect the variables in scope. Using this technique, a developer can then step through the execution one instruction at a time or resume control flow.

Extending the concept of breakpoints, we can imagine a similar approach when debugging SCXML documents, whereby the role of "single instruction" as a place to halt execution and to step towards needs to be defined.

BREAKPOINTS FOR SCXML

In our approach, an interpreter with a debugging session attached will raise a series of *qualified breakpoints* while interpreting an SCXML document. A qualified breakpoint references the current phase of execution and contains a set of additional attributes (e.g. the relevant DOM node or a state's id attribute) depending on the phase. User-supplied breakpoints are then matched against the current qualified breakpoint to determine whether to halt interpretation.

We identified an open set of phases of execution where we allow the interpreter to be suspended. Motivated by the "algorithm for SCXML interpretation" from the SCXML draft these are (approximately in order of execution from a stable configuration) given in table 1.

Phase Identifier	Description
`event-before`	After popping an event from the event queue (internal or external).
`microstep-before`	Before performing a microstep for the enabled transitions.
`state-before-exit`	Before the `<onexit>` elements for a state from the exit set are interpreted.
`executable-before`	Before an element of executable content is interpreted.
`executable-after`	After an element of executable content was successfully interpreted.
`state-after-exit`	After the `<onexit>` elements were interpreted.
`invoker-before-cancel`	Before cancelling the invocation of an invoker.
`invoker-after-cancel`	After cancelling the invocation of an invoker.
`transition-before`	Before a transitions executable content is interpreted.
`transition-after`	After a transitions executable content was interpreted.
`state-before-enter`	Before the `<onentry>` elements for a state from the exit set are interpreted.
`state-after-enter`	After the `<onentry>` elements were interpreted.
`microstep-after`	After a microstep was performed.
`invoker-before-invoke`	Before actually invoking the entry set's invokers.
`invoker-after-invoke`	After the entry set's invokers were invoked.
`stable-on`	When the interpreter reached a stable configuration.

Table 1. Phases of interpretation of an SCXML document.

The identifier of a phase can be decomposed into up to three components, (i) the *subject*, (ii) a *time* specifier (iii) and an optional *activity* if it is not implied already. Depending on the subject of the phase, there are additional attributes available in a qualified breakpoint listed in table 2. Formally, all qualified breakpoints featuring an element attribute would not need the other attributes as they can be be obtained via the element DOM node; including them is mere convenience for matching user-supplied breakpoints below.

Subject	Field	Description
`event`	`eventName`	The event's name.
`microstep`	`N/A`	
`state`	`stateId` `element`	The state's id attribute. The state's DOM element.
`executable`	`executable Name` `element`	The executable content's element name. The executable content's DOM element.

`invoker`	`invokeId`
	`invokeType`
	`element`
		The invoker's id
		The invoker's type
		The invoker's DOM element.

`transition`	`trans SourceId`
	`trans TargetId`
	`element`
		The id of the state containing a transition element.
		One of the transition's target states per id.
		The transition's DOM element.

`stable`	`N/A`	

Table 2. Attributes of qualified breakpoints per phase of interpretation.

Note, that we do not provide any facilities to debug datamodel specific source code contained in `<script>` elements or to step into e.g. the `cond` expression of a transition. While we do think that it would be most useful, the amount of supporting infrastructure for every new datamodel provided by an SCXML interpreter would be immense and is considered out of scope.

Also note that it would formally be sufficient to reduce the set of phases to `before-node` and `after-node` and to provide a single `xpath` expression per user-supplied breakpoint. And then to halt interpretation before or after a XML node was processed by the interpreter. We do, however, feel that the less general phase descriptors above help developers to identify the actual phase of interpretation where a breakpoint is needed.

Breakpoint Matching

Each qualified breakpoint is matched against the set of user-supplied breakpoints when it is raised. User-supplied breakpoints resemble qualified breakpoints with all their attributes optionally, except that the `element` DOM node attribute is replaced by an `xpath` expression. Additionally, user-supplied breakpoints may contain a `condition` attribute which is to be evaluated on the documents datamodel to conclude matching.

For a user-supplied breakpoint to match, it has to be less specific or identical in all its attributes while referencing the same phase of execution. The optional `condition` attribute per breakpoint is then evaluated to determine whether execution ought to be actually suspended. As the condition is evaluated on the datamodel, care has to be taken to ensure that it is free

of side-effects. Otherwise, evaluating the condition may alter the datamodel. The pseudo-code to determine a match is given in algorithm 1.

Input : BP_{user}, BP_{qual}
Output: Whether BP_{user} matches given BP_{qual}

```
1  if BP_user.phase ≠ BP_qual.phase then
       /* Not referencing the same phase                    */
2      return false;
3  end
   /* Iterate every field for the phase                     */
4  for field ∈ FieldsFor(BP_qual.phase) do
5      if BP_user.[field] = undef then
           /* No value for field specified                  */
6          continue;
7      end
8      if field = "eventName" then
           /* Event names are matched per SCXML draft
              as descriptors                                */
9          if not nameMatch
10             (BP_qual.eventName,
11              BP_user.eventName) then
12             return false;
13         end
14         continue;
15     end
16     if field = "xpath" then
           /* Field contains an XPath expression            */
17         if not NodeSet(BP_user.[field])
18                .contains(BP_qual.element)) then
19             return false;
20         end
21         continue;
22     end
       /* Rest is matched literally                         */
23     if BP_user.[field] ≠ BP_qual.[field] then
24         return false;
25     end
26 end
   /* Check the condition on the datamodel                  */
27 if BP_user.condition ≠ undef then
28     if not evalAsBool(BP_user.condition) then
29         return false;
30     end
31 end
32 return true;
```

Algorithm 1: Matching Breakpoints

The fields for a breakpoint pertaining to a phase depending on its subject are given in table 2 with `element` replaced by `xpath`.

Stepping Through

By introducing the concept of *qualified breakpoints*, we can assign meaning to "stepping through" a SCXML document. Whenever interpretation is halted, the debugger will simply allow the developer to step to the next qualified breakpoint, halting interpretation at each.

DEBUGGING PROTOCOL

In order to support the debugger, we devised a pragmatic HTTP-based protocol (table 3) passed between our SCXML runtime environment and a debugging client (see Fig. 1). We implemented the client protocol in a stand-alone ECMAScript / HTML document which can simply be started from the filesystem. The client follows an object-oriented approach with ECMAScript: basic functionality pertaining to the graphical presentation is in a base class with the HTTP protocol implemented in a derived class. This will allow other developers to reuse the graphical presentation while adapting communications to the specifics of their interpreters.

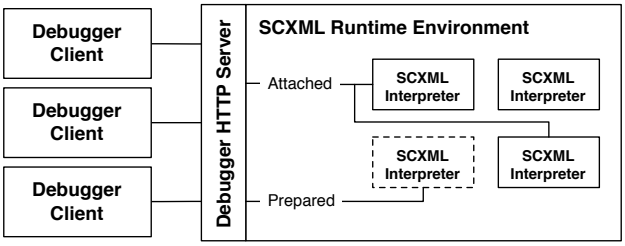

Figure 1. Debugging Architecture.

Path	Function
No session required	
`/connect`	Request a new session identifier
`/sessions`	List of running SCXML interpreters
Only when in session	
`/poll`	Request is long-polling for server push
`/disconnect`	Disband session and detach from interpreter, quit interpreter if prepared for this session
`/prepare`	Prepare a new SCXML interpreter with a given document
`/attach`	Attach to a running SCXML interpreter as returned by `/sessions`
`/start`	Only available with a debugger-prepared document, starts the interpreter
`/stop`	Only available with a debugger-prepared document, stops the interpreter
`/pause`	Pauses the interpreter
`/resume`	Resumes execution of an interpreter
`/step`	Causes every qualified breakpoint to match, thus allows stepping
`/bp/add`	Add a user-supplied breakpoint
`/bp/remove`	Remove a user-supplied breakpoint
`/bp/enable`	Enable a user-supplied breakpoint
`/bp/disable`	Disable a user-supplied breakpoint
`/bp/enable/all`	Enable breakpoint evaluation
`/bp/disable/all`	Disable breakpoint evaluation
`/bp/skipto`	Continue execution until given breakpoint is reached
`/eval`	Evaluate an expression on the datamodel

Table 3. HTTP Request path and their function.

All communication via HTTP takes place as POST request and respective replies with a `content-type` of `application/json`. If XML is to be transmitted as part of a request or reply, it is simply encoded as a JSON attribute. Every server reply contains a JSON attribute `status` which is either set to `success` or `failure` in which case `reason` contains a string detailing the cause of the failure.

There are two modes of operation for the debugger, it is either (i) attached to an already running interpreter, or (ii) prepares its own interpreter by passing an SCXML document. If attached to an already running interpreter, we do not allow to

stop execution altogether, only to pause execution and only as long as the debugger is connected.

A debugging session starts with the client connecting to the HTTP server and requesting a new session identifier at `/connect`. This identifier is subsequently used as an attribute in every request to identify the debugging session. To associate an interpreter with the session, the client either prepares a new interpreter by requesting `/prepare` with an XML document or a URL or attaches itself to a running session as returned by `/sessions`.

To support server push via HTTP, a connected client maintains a long-polling request to the server path `/poll` with its session identifier in the request. Whenever the server needs to asynchronously return information to the client (e.g. the match of a breakpoint or a log message), it is send as a reply and the client requests `/poll` anew. As a reply might be related to various events that occurred asynchronously, the `replyType` attribute identifies the type of the reply.

When a session is established, the server accepts user-supplied breakpoints encoded as a JSON structure with their respective attributes as discussed above. We do not reference breakpoints by an identifier but by value. This implies that there cannot be two identical breakpoints, which makes sense as they would match the exact same qualified breakpoints, causing the interpreter to halt twice.

Example Session

For the example, we will use `test152.scxml` from the SCXML Implementation Report Plan, XSLT transformed for the ECMAScript datamodel. This selection is arbitrary, except that the document features executable content and is rather compact.

The server in the following communication is a runtime environment for SCXML interpreters. Our implementation allows for the concurrent interpretation of an arbitrary number of SCXML documents and the runtime environment will coordinate new instances or attach client debuggers to running instances. The client is an HTML document running in a browser, issuing HTTP request via the `XMLHTTPRequest` object.

Client → Server `/connect`

Client ← Server `/connect`

```
session:   "d8782c2d",
status:    "success"
```

We started by requesting a new session identifier. This will cause the server to instantiate an empty debugging session without an SCXML interpreter associated.

Client → Server `/sessions`

Client ← Server `/sessions`

```
sessions:  [],
status:    "success"
```

Here we requested a list of running interpreters from the server to potentially attach this session to. The list came back empty as no interpreters are running in this example. All subsequent client requests will contain the `session` attribute and all server replies a `status` attribute. We drop both in the following communication for brevity.

Client → Server `/poll`

We issued the long-polling HTTP request for server push. It will only return when the server needs to notify us asynchronously.

Client → Server `/bp/add`

```
phase:     "state-after-enter",
stateId:   "s2",
```

Client ← Server `/bp/add`

The client added a breakpoint, asking the interpreter to halt interpretation after entering state `s2`.

Client → Server `/prepare`

```
url:   "http://localhost/test152.scxml",
xml:   -- Escaped XML document --
```

Client ← Server `/prepare`

Client → Server `/step`

Client ← Server `/step`

We prepared an interpreter by uploading an SCXML document containing `test152.scxml` and started interpretation by stepping to the first qualified breakpoint.

Client ← Server `/poll`

```
qualified:
  phase:     "state-before-enter"
  stateId:   "s0"
  xpath:     "//state[@id="s0"]"
replyType:   "breakpoint"
```

Client → Server `/poll`

The request to `/poll` returned with the first qualified breakpoint triggered just before entering state `s0`. This breakpoint is not caused by a user-supplied breakpoint, so no `breakpoint` attribute is set. The `xpath` attribute allows us to update eventual visualizations in the client. Furthermore, no `active` nor `basic` states are returned as the interpreters configuration is still empty.

Client → Server `/resume`

Client ← Server `/resume`

As we started the interpretation with a single step to the next qualified breakpoint, the interpreter is still halted at the very first opportunity, just before entering state `s0`. Here, we ask the interpreter to resume normal interpretation as opposed to `step` to the next qualified breakpoint.

Client ← Server `/poll`

```
activeStates:   ["s2"]
basicStates:    ["s2"]
breakpoint:
  phase:        "state-after-enter"
  stateId:      "s2"
qualified:
  phase:        "state-after-enter"
  stateId:      "s2"
  xpath:        "//state[@id="s2"]"
```

Client → Server `/poll`

As the interpreter was resumed, it continued to raise qualified breakpoints. It was halted again after entering state `s2` as it matched the user-supplied breakpoint which is also referenced in the reply. Furthermore, we can see that `s2` is both a basic and an active state of the interpreters configuration.

Client → Server `/eval`

```
expression:   "_event"
```

Client ← Server `/eval`

```
eval:   -- _event as a JSON structure --
```

Here, we ask the interpreter to interpret the expression `"_event"` on the current datamodel, causing a reply containing a JSON structure of the result of the evaluation. Using this approach, we can inspect all variables in the interpreters datamodel – even invoke functions.

Client → Server `/disconnect`

Client ← Server `/disconnect`

We finally disconnect the client. As the associated interpreter in the runtime environment was prepared from and for this session, this will cause the interpreter to exit. If the debugging session were attached to an interpreter already running (via a request to `/attach`), this would not stop the intepreter, just detach the session.

DEBUGGER INTERFACE

We implemented a client for the protocol detailed above in HTML/ECMAScript and the server component as part of our SCXML interpreter. By choosing HTML for the client, we traded native widget-sets and operating system integration for platform independence – it will even run on mobile devices.

A screenshot of the debugger interface is given in Fig 2. The debugger's main window is a draggable `<div>` with its position fixed to the browser's viewport. This allows a developer to scroll through the SCXML document displayed in the main browser window while the debugger remains in place (see Fig 3).

The debugger itself is composed of a title bar with a drop-down button containing a menu to load SCXML documents, attach the debugger to a running instance and save breakpoints. Next to the drop-down are the well-known debugging controls of start/stop, pause/resume and stepping. Beneath the title bar is a input field to specify the interpreters base URL and a button to establish the connection to the interpreter.

Figure 2. Debugger's HTML interface.

The bulk of the debugger interface displays three collapsible panels for (i) managing breakpoints, (ii) inspecting the datamodel, (iii) and messages returned by the server or raised by the client itself.

At the bottom of the window are some status indicators and the name of the document the associated interpreter has loaded if any.

The breakpoint panel's header contains buttons to disable breakpoints altogether, add a new breakpoint and remove all breakpoints. When execution is halted, a subtitle is displayed referencing the current phase of interpretation. In the panel's content area, the user-supplied breakpoints are listed with along with buttons to (i) enable/disable, (ii) skip to, (iii) edit, (iv) and remove the breakpoint. When the interpretation was halted due to such a user-supplied breakpoint being matched, it is highlighted in the list.

The datamodel panel allows developers to issue expressions for the SCXML interpreter to evaluate on the datamodel and displays their results. It is possible to actually alter the datamodel if the expression is not free of side-effects.

The final panel contains a text-area where various messages from the server or client are displayed and serves primarily for debugging the debugger and the protocol.

EVALUATION

We did not perform a formal evaluation in the form of a user study. Yet as a preliminary evaluation, we showed the debugger to two SCXML experts to explore. We did get some reports about usability defects, mostly related to peculiarities with the HTML interface or the placement and visual representations of commands. The overall approach was deemed to be intuitive as the experts were familiar with debuggers

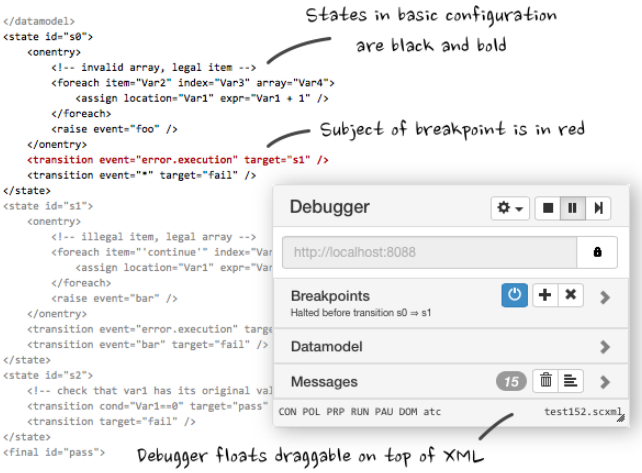

Figure 3. SCXML Document is displayed beneath Debugger.

from traditional development environments. We plan to address these issues and conduct a more in-depth evaluation in future work.

CONCLUSION

We proposed an approach to map the established debugging techniques of breakpoints, stepping and variable inspection onto the interpretation of SCXML documents. By introducing the concept of *qualified breakpoints* we were able to assign familiar semantics to these techniques.

There are several areas where the debugger could benefit from future work. Foremost (i) the integration of an authoring environment and subsequently (ii) a more formal user-study. With regard to the first point, we currently only display the SCXML document being debugged in the browser's main window with some highlighting for the interpreter's configuration and the element related to the last qualified breakpoint. Allowing SCXML developers to actually edit the SCXML document, maybe in a navigable state-chart representation would be most useful. This would entail a more pressing need for the second point as the resulting integrated development environment for SCXML implies a larger design-space for the user interface to validate as part of a user-study.

ACKNOWLEDGMENTS

This work has been partially supported by the FP7 EU large-scale integrating project SMART VORTEX (Scalable Semantic Product Data Stream Management for Collaboration and Decision Making in Engineering) co-financed by the European Union. For more details, visit `http://www.smartvortex.eu/`.

REFERENCES

1. Eisenstadt, M. My hairiest bug war stories (1997). *Communications of the ACM 40*, 4 (1997), 30–37.
2. Hailpern, B., and Santhanam, P. Software debugging, testing, and verification. *IBM Systems Journal 41*, 1 (2002), 4–12.
3. Horwitz, S., Reps, T., and Binkley, D. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS) 12*, 1 (1990), 26–60.
4. LeBlanc, T. J., and Mellor-Crummey, J. M. Debugging parallel programs with instant replay. *Computers, IEEE Transactions on 100*, 4 (1987), 471–482.
5. McDowell, C. E., and Helmbold, D. P. Debugging concurrent programs. *ACM Computing Surveys 21*, 4 (1989).

Semantics of States and Transitions in statecharts-based markup languages: a comparative study between SWC and SCXML

Marco Winckler, Charly Carrére, Eric Barboni
ICS-Team, Institute of Research in Informatics of Toulouse (IRIT), Univerity Paul Sabatier (UPS)
118 route de Narbonne, 31062 Toulouse Cedex, France
{winckler, carrere, barboni}@irit.fr

ABSTRACT
Statecharts has been demonstrated as a suitable solution for specifying the navigational behavior of hypermedia systems. However, in order to cope with the idiosyncrasies of the Web development (such as representation of client and server stages) we have been proposed an extension to the original Harel's statecharts called StateWebCharts (SWC). In this paper we discuss the rationale for extending statecharts notations for specific application domains such as the Web. Moreover, we illustrate how the domain-specific constructs provided by SWC might help to solve problems that would require specific semantics for states and transitions. Then, we compare the constructs proposed by the SWC notation with Harel's statecharts and SCXML. We argue that it would be possible to convert SWC specification into SCXML by losing some semantic on transitions and states. Conversely, extensions for adding domain-specific semantics on SCXML would benefit not only its inner utility to specifying Web application but it could also useful in other application domains.

Author Keywords
Statecharts, Web navigation, Web applications, SCXML, StateWebCharts, SWC.

ACM Classification Keywords
H.5.m. Information interfaces and presentation (e.g., HCI).

INTRODUCTION
Research on navigation modelling has a long history in hypertext and hypermedia domain and it has strongly influenced the technology for the Web. State-based notations such as Petri nets [7] and StateCharts [2][4][6][7][9] have been explored to model navigation for hypertext systems. However, such proposals are not able to represent some aspects of Web applications such as dynamic content generation, support to link-types (toward external states, for instance), client and server-side execution. However, some of them [2, 7, 12] do not make explicit the separation between interaction and navigation aspects in the models while this is a critical aspect for the usability of Web application. Connallen [1] proposed an efficient solution for modelling Web applications using UML stereotypes. Such as an approach mainly target data-intensive applications and even propose prototyping environments to increase productivity. However, the limitation is that navigation is described at a very coarse grain (for instance navigation between classes of documents) and it is almost impossible to represent detailed navigation on instances of these classes or documents. The same problem appears in Kock [5] which may reduce creativity at design time as they impose the underlying technology and as they do not provide efficient abstraction views of the application under development. In order to cope with the idiosyncrasies of Web navigation, we have proposed in previous work [10] an extension to Harel's statecharts called StateWebCharts notation (SWC). Such as a notation dedicated constructs for modelling specificities of states and transitions in Web applications. Most elements included in SWC notation aim at providing explicit feedback about the interaction between users and the system.

In this paper we discuss the importance of representing domain-specific semantics that can be associated to states and transitions whilst using statechart-based markup languages such as SWC and SCXML [13]. We illustrate how domain-specific constructs can help to solve problems that would require specific semantics for states and transitions for the Web. We assume that the semantics of states and transitions of SWC might be specific to the Web domain and not easily generalizable. However, other application domains have their idiosyncrasies thus require different semantics for transitions and states. Nonetheless, we argue that by adding semantics to states and transitions of SCXML, it would be possible to employ SCXML as a markup language for copying with the same challenges addressed by SWC. In the next section we present the SWC notation and we illustrate its uses. Then we compare the syntax of construct present in the original Harel's statecharts, those proposed by SWC and SCXML markup languages.

EICS 2014 Workshop, Engineering Interactive Systems with SCXML, June 17, 2014, Rome, Italy

THE STATEWEBCHART NOTATION (SWC)

SWC is rooted on Harel's StateCharts [3] but it adds semantics to it to address Web domain issues. SWC' states are abstractions of containers for objects (graphic or executable objects). For Web applications such containers are usually HTML pages. States in SWC are represented according to their function in the modelling. In a similar way, a SWC transition explicitly represents the agent activating it. Each individual Web page is considered a container for objects and each container is associated to a state. Links and interactive objects causing transitions are triggered by events. The semantic for a SWC state is: current states and their containers are active while non-currents are hidden. Figure 1 show all SWC elements.

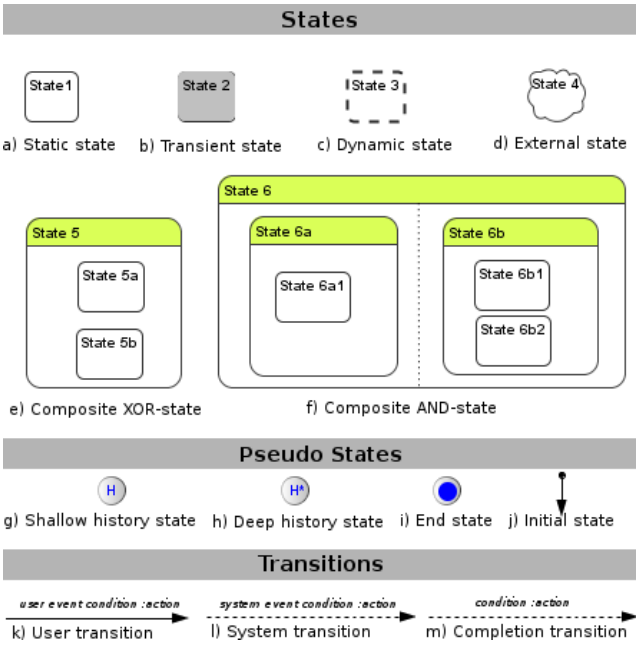

Figure 1. Graphical representation of StateWebCharts.

Static states (Figure 1.a) are the most basic structures to represent information in SWC. They refer to a container with a static set of objects; in a static state all objects are present in the browser. However, those objects are not necessarily static by themselves; they could have dynamic behaviour as we usually find, for example, in applets, JavaScript or animated images. Static is the default type.

Transient states (Figure 1.b) describe a non-deterministic behaviour in the state machine. Transient states are needed when a single transition cannot determine the next state for the state machine. Only completion or system events are accepted as outgoing transitions of transient states. Transient states only process instructions and they do not have a visual representation towards users. They refer to server-side parts of Web applications, such as PHP scripts.

Dynamic states (Figure 1.c) represent content that is dynamically generated at runtime. They are usually the result of a transient state processing. The associated container of a dynamic state is empty. The semantics for this state is that in the modelling phase designers are not able to determine which content (transitions and objects) will be made available at run time. However, designers can include static objects and transitions inside dynamic states; in such case transitions are represented, but the designer must keep in mind that missing transitions might appear at run time and change the navigation behaviour.

External states (Figure 1.d) represent information that is accessible through relationships (transitions) but are not part of the current design. For example, consider two states A and B. While creating a transition from A to B, the content of B is not accessible and cannot be modified. Thus, B is considered external to the current design, which is often the case of external sites. External states avoid representing transitions without a target state, however all activities (i.e. entry, do, and exit) in external states are null.

SWC's events indicate the agent triggering them: user (e.g. a mouse click), system (e.g. a method invocation that affects the activity in a state) or completion (e.g. execution of the next activity). A completion event is a fictional event that is associated to transitions, e.g. change the system state after a timestamp. This classification of event sources is propagated to the representation of transitions. Transitions whose event is triggered by a user are graphically drawn as continuous arrows (Figure 1.k.) while transitions triggered by system or completion events are drawn as dashed arrows (Figure 1.l and Figure 1.m, respectively).

In order to represent behaviour such as those found in StateCharts, SWC provides the following pseudo-states (g) shallow history, (h) deep history, (i) end state and (j) initial state. These pseudo-states do not have any container associated to them. Pseudo-states and composite state in SWC are very close of the definition given by StateCharts (see [10] for details). Both states and transitions can have associated actions. When associated to transitions, actions represent what is executed by the system while traversing a transition. When associated to state, actions represent the activity performed by the state. All SWC constructs are stored in a XML format as illustrated at the Figure 2.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with SWCEditor -->
<swc>
<CompositeState     id="root"   label="root"   file="null"
initial="S1" concurrent="false">
<BasicState  id="S1" label="main intro" type="BasicState"
file="spider_intro.html" >
</BasicState>
<BasicState   id="S2"  label="schedule"  type="BasicState"
file="spider_schedule.html">
</BasicState>
...
</CompositeState>
<Transition  id="t1"  type="user"  label=""  source="S1"
target="S2" trigger="mouseClick" guard="true" action="">
</Transition>
...
</swc>
```

Figure 2. XML file describing a SWC model.

SWC IN PRACTICE

SWC models can be built using the tool SWCEditor [11] which supports the creation, edition, visualisation, simulation and analysis of SWC models. Hereafter we illustrate how the some elements of the SWC notation have been operationalized using the SWCEditor to solve problems associated to navigation modelling of Web applications.

Separation between client/server states

One of the main features of SWC is the possibility to associate specific semantics for states and transitions in the navigation diagrams. Figure 3 illustrates these semantics by a simple SWC statemachine diagram which models the navigation behaviour for client/dynamic/transient states and user/system driven transitions.

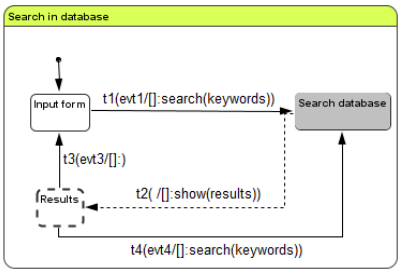

Figure 3. Navigation modelling client/dynamic/transient states and user/system transitions.

As we shall see at Figure 3, states are depicted accordingly to the semantic given to states. For example, the state "input form" is a Web page that contains a web form whilst the page "results" is automatically generated at the client-side (i.e. the browser) as a response to an execution of a state that can only be processed on the server side (i.e. "search database"). User driven transitions, depicted with continuous lines, are interpreted as users' clicks whilst transitions automated by the system (ex. "t2") are depicted as dashed lines. Such as inner semantic for states and transitions can properly mapped to the proper constructs used to build the Web sites.

Setting boundaries between local and external models

During early evaluation phases of development designers have to check if abstract modelling will behave as expected. Simulations of models can be useful for that purpose. Thanks to the special constructs of SWC it is possible to associate navigation model with advanced Web prototypes. Figure 4 presents how SWCEditor allows simulation and co-execution of SWC models. First of all, let us to focus on the left part of the figure 4. There are two windows: the simulator window (at top-left) and the visualization window (at bottom-left). The window simulator is composed of two panels showing: the set of active state (grey panel at left) and the set of enabled transitions at a time (white panel at right). The visualization window is the main graphic editor of SWC models (the SWCEditor module).

When an enabled transition is selected the system fires it immediately causing the changing of the system, which displays the next stable configuration. The current statemachine configuration is shown in red. If a container is associated to a state, it is possible to concurrently display the corresponding container (typically a Web page) in a browser during the simulation. The concurrent simulation of model and implementation is suitable during the prototyping activity. Thus, designers can follow the changes in the abstract specification at the SWCEditor as well as its concrete implementation at the Web browser. Figure 4 shows in a browser window (at right part) the corresponding Web page for the current state in that simulation. Notice that external states are used to represent external links attached to the current web site design.

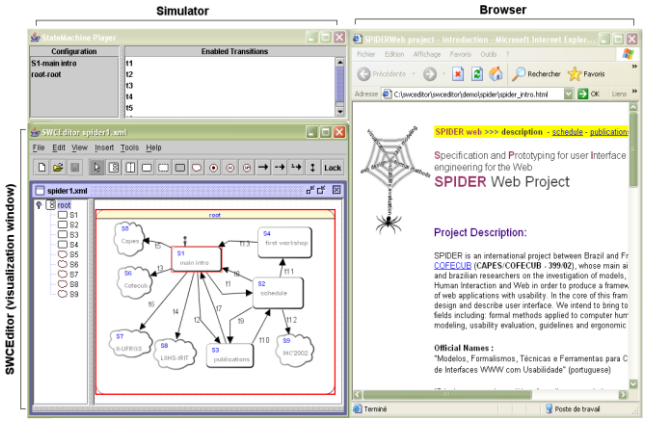

Figure 4. Co-execution of navigation models and Web prototypes.

Automated usability inspection of SWC models

One of the advantages of the semantic added to constructs is to support the reason about models in a certain way. In previous work [14] we have investigated how to use SWC models to support guidelines verification in early phases of development. The basic idea was to map concepts present in ergonomic guidelines (ex. "page") to SWC constructs as show in Figure 5. After that, we have implemented automated parsers for guidelines such as "*Each **page** must have a **link** to it*" that inspect SWC models as follows "*Each **state** must have a **transition** pointing to it*". Those tools thus exploit the semantic of models for automatically inspecting models in

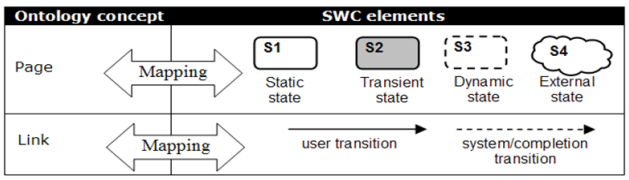

Figure 5. Mapping SWC constructs and Ontological concepts.

COMPARING NOTATIONS

In order to assess the expressiveness power of SWC, we compare in Table 1 its constructs with those defined by the original Harel's statecharts and duly supported by SCXML.

Table 1. Comparing constructs in Harel's statecharts, SCXML and SWC.

Harel's	SCXML	SWC
Statemachine	The language start by an <scxml> tag, example: **<scxml>** <state> … </state> **</scxml>**	The language start by an <swc> tag, ex : **<swc>** <state type="BasicState"> … </state> **</swc>**
States	Basic state reference, example: <state> … </state>	<**BasicState**> … </BasicState> Possible types: Basic/Static/TransientState/External
Composite State	Composition defined by inner hierarchy, example: <state id="S" **initial="s1"** > <state id="s1"> </state> </state> ---------- **AND states**: Classic state hierarchy, ex : <state id="S" initial="s1" > <state id="s1"> </state> </state> ---------- **OR states**: The <parallel> element encapsulates a set of child states which are simultaneously, ex : **<parallel id="Test5P">** <state id="Test5PSub1" initial="Test5PSub1Final"> <final id="Test5PSub1Final"/> </state> <state id="Test5PSub2" initial="Test5PSub2Final"> <final id="Test5PSub2Final"/> </state> <onexit> <log expr="'all parallel states done'"/> </onexit> **</parallel>**	Dedicate state type, ex : <**CompositeState** id="root" label="root" file="null" initial="S1" concurrent="false"> </**CompositeState**> ---------- **AND states**: <CompositeState id="root" label="root" file="null" initial="S1" **concurrent="false"** /> ---------- **OR states**: <CompositeState id="root" label="root" file="null" initial="S1" **concurrent="true"**/>
History	Determined by a pseudo-state, ex : <history type="deep" id="history-actions"> </history>	Determined by a pseudo-state, ex : <CompositeState id=”root” …> <**DeepHistory** id=”S1” /> <**ShallowHistory** id=”S2” /> </CompositeState>
Final states	Determined by a pseudo-element, ex : **<final id="Test5PSub1Final"/>**	Determined by a pseudo-element, ex : <**EndState** id=”S1” />
Variables	<datamodel> is a wrapper element which encapsulates any number of <data> elements, ex : **<datamodel>** <**data** id="door_closed" expr="true"/> **</datamodel>** ---------- **<script>** time.setHours(_event.data.currentHour + (_event.isAm ? 0 : 12) - 1); **</script>**	The name and value are in the parameter declaration, ex : **<parameters>** <parameter name="param1" value="0" /> **</parameters>**
Conditions	The conditions are defined using multiple tags, ex : **<if cond="true">** **<foreach array="cart.books" item="book">** <log expr="'Cart contains book with ISBN ' + book.isbn"/> **</foreach>** **<elseif cond="false"/>** <log expr="You can't use it"/> **</else>** <log expr="Error boolean"/> **</if>**	Condition in transition definition, ex : <Transition id="t1" type="user" label="" source="S1" target="S2" **trigger="mouseClick"** guard="true" action="" />
Action	<state id="s1" initial="s11"> **<onexit>** <log expr="'leaving s1'"/> **</onexit>** **<onentry>** <log expr="'entering S'"/> **<onentry>** </state>	Action defined in the Transition definition, ex : <Transition id="t1" type="user" label="" **source="S1" target="S2"** trigger="mouseClick" guard="true" **action="methodCall()"** />
Transition	<**transition** event="ping" target="takeOrder"/>	**<Transition** id="t2" type="user" label="" source="S1" target="S3" trigger="mouseClick" guard="true" action=""> **</Transition>**
External communication	**<invoke** id="timer" type="x-clock" src="clock.pl"> **<finalize>** <script> time.setHours(_event.data.currentHour + (_event.isAm ? 0 : 12) - 1); </script> **</finalize>** **</invoke>** ---------- **<send target="csta://csta-server.example.com/" type="x-csta">** <content> <csta:MakeCall> <csta:callingDevice>22343</csta:callingDevice> <csta:calledDirectoryNumber>18005551212</csta:calledDirectoryNumber> </csta:MakeCall> </content> **</send>**	

As we shall see in Table 1, SWC and SCXML cover most of the original elements proposed by Harel's statecharts. Nonetheless, a few elements differ with respect to the inner Document Type Definition (DTD) they implement. Indeed, whilst SWC features a specific tag, SCXML implicitly represent for composite states by adding sub-states inside the tags. In addition, actions in SCXLM are represented by dedicated tags whilst SWC embedded them as expressions associated to attributes elements in the tag transition. Some tags have different names for addressing the same element, ex. *final* and *endstate* for indicating end pseudostates. Most of these differences are syntactic and can be easily overcome by a few transformation rules ensuring the compatibility between notations.

However, SWC does not take into account complex external communication mechanisms. Further investigate would be required to determine in which extension *communication mechanisms* could correspond to *dynamic* states in SWC. In all cases, all these difference worth to be carefully discussed, and would require extension in both notations if compatibility should be assured.

Lastly but not less important, a significant difference between SWC and SXCML is that the latter one provides a generic representation of states and transitions without any domain-specific semantics, whilst the former clearly features a semantics for navigation of Web application. Indeed, SWC offers four alternative types which specific semantics for basic states, whilst SCXML only provides one type of state. This is observable by the attribute *type* that can be associated to *states* and transitions. Moreover, SWC also provides another attribute to states that allows the mapping to contents, namely *file*.

CONCLUSION

SWC and SCXML are both based on Harel's statecharts and therefore share many similarities. In some extensions, models built in one notation could be translated to another, however, the compatibility is not 100% accurate and we would lose semantic and functionality in this operation. Further studies are required to determine the compatibility level and the side-effect implications of converting models. But still, we estimate that some level of compatibility ensured by model-transformation is possible.

However, if we consider the Web as a suitable application domain for SCXML we might argue that this notation lacks of some attributes to express the rich semantic of navigation. This lack of semantics of states and transitions would prevent the reasoning about the application and the development of dedicated tools as illustrated by the research around SWC. Moreover, we assume that this lack of semantics might not be specific to Web navigation models and other researchers would be interested in proposing other elements.

The proliferation of DLS might not be a definite solution for similar problems in different application domains. For that purpose, as standard language such as SCXML would be ideal as a lingua franca between statechart-based DSL like SWC. We argue that the level of semantic expected for state and transitions in SCXML could be easily solved by a couple of attributes that could be added to markup language. If so, we could pursue the research about navigation modeling using SCXML as a replacement to SWC and still achieve similar results as those previous illustrated in this paper.

REFERENCES

1. Connallen, J. Building Web Applications with UML. Addison-Wesley, 1999.
2. Dimuro, G. P.; Costa, A. C. R. Towards an automata-based navigation model for the specification of Web sites. In: 5th Workshop on Formal Methods, Gramado, 2002. Electronic Notes in Theoretical Computer Science.
3. Harel, D. StateCharts: a visual formalism for computer system. Science of Computer Programming, 8, N. 3:231-271 p., 1987.
4. Horrocks, I. Constructing the User Interface with Statecharts. Addison-Wesley, 1999.
5. Koch, N.; Kraus, A. The expressive Power of UML-based Web Engineering. In 2nd Int. Workshop on Web-oriented Software Technology (IWWOST02). June 2002.
6. Leung, K., Hui, L., Yiu, S., Tang, R. Modelling Web Navigation by StateCharts. In proc. 24th Inter. C.S.A., 2000, Electronic Edition (IEEE Computer Society).
7. Oliveira, M.C.F. de; Turine, M. A. S.; Masiero, P.C. A Statechart-Based Model for Modeling Hypermedia Applications. ACM TOIS. April 2001.
8. Stotts, P. D.; Furuta, R. Petri-net-based hypertext: document structure with browsing semantics. ACM Trans. on Inf. Syst. 7, 1 (Jan. 1989), Pages 3 - 29.
9. Turine, M. A. S.; Oliveira, M. C. F.; Masieiro, P. C. A navigation-oriented hypertext model based on statecharts. In Proc. 8th ACM Hypertext. 1997, Southampton, UK.
10. Winckler, M.; Palanque, P. StateWebCharts: a Formal Description Technique Dedicated to Navigation Modelling of Web Applications. DSVIS'2003, Portugal, June 2003.
11. Winckler, M.; Barboni, E.; Farenc, C.; Palanque, P. SWCEditor: a Model-Based Tool for Interactive Modelling of Web Navigation. International Conference on Computer-Aided Design of User Interface - CADUI'2004, Funchal, Portugal, 13-16 January 2004.
12. Zheng, Y.; Pong, M. C. 1992. Using statecharts to model hypertext. In Proc.of the ACM Conference ECHT'92, Milan, Italy. ACM Press, New York, NY, 242-250.
13. State Chart XML (SCXML): State Machine Notation for Control Abstraction. W3C Candidate Recommendation 13 March 2014. At: http://www.w3.org/TR/scxml/
14. Xiong, J., Farenc, C., Winckler, M. Towards an Ontology-based Approach for Dealing with Web Guidelines. In Proc. Int. Workshop on Web Usability and Accessibility. Auckland, New Zealand, September 1-4, 2008. Springer LNCS 5176, pages 132-141.

From Harel To Kripke: A Provable Datamodel for SCXML

Stefan Radomski
TU Darmstadt
Telecooperation Group
radomski@tk.informatik.tu-darmstadt.de

Tim Neubacher
TU Darmstadt
neubacher@cs.tu-darmstadt.de

Dirk Schnelle-Walka
TU Darmstadt
Telecooperation Group
dirk@tk.informatik.tu-darmstadt.de

ABSTRACT

When writing critical applications, developers need a way to formally prove that the resulting system complies to a set of constraints and exposes a specified behavior. With SCXML being a markup language for Harel state-charts, there is an untapped possibility to reduce the expressiveness of its embedded datamodel to enable model-checking techniques. In this paper we introduce a *Promela* datamodel for SCXML documents, enabling to transform these documents onto input files for the SPIN model-checker. By retaining most of the semantics, developers can prove various properties of systems expressed via SCXML documents employing this datamodel.

Author Keywords

SCXML; Harel State-Chart; Formal Verification; Languages

ACM Classification Keywords

D.2.4. Software/Program Verification: Model checking

INTRODUCTION

The use of model-checking tools is most pronounced with controller software for embedded systems: A formal proof that a controller for an elevator will always allow the system to reach a state where the passenger cabin is on the ground floor with the doors open would guarantee this very essential property of elevators. Enabling model-checking approaches for SCXML [1] documents would, consequentially, allow us to formalize and guarantee similar properties of the systems described.

One popular implementation for model-checking is the SPIN model checker: A system described in the PRocess MEta LAngauge (Promela) is taken as input and SPIN allows to analyze this program with respect to different questions, e.g.:

1. Is there an execution sequence that invalidates an assertion?
2. Can the system reach an invalid end-state?
3. Is the system always making progress?
4. Does a claim given in Linear Temporal Logic (LTL) hold?

Especially the LTL claims enable very elaborate techniques to prove various properties of a system. There are several operators to create simple and compound claims dealing with properties in linear temporal logic.

Always ($[\,]\mathtt{c}_1$): A given claim will always be true.

Eventually ($<>\ \mathtt{c}_1$): Some claim will be true in the future, with the future starting now.

Not ($!\mathtt{c}_1$): Negates a claim.

Next ($\mathtt{X}\mathtt{c}_1$): In the next state a given claim will be the case.

Strong Until ($\mathtt{c}_1\ \mathtt{U}\ \mathtt{c}_2$): A claim is replaced by another claim.

Weak Until ($\mathtt{c}_1\ \mathtt{W}\ \mathtt{c}_2$): When a claim holds, another one will be the case later.

And ($\mathtt{c}_1\ \&\&\ \mathtt{c}_2$), **Or** ($\mathtt{c}_1\ \|\ \mathtt{c}_2$), **Implies** ($\mathtt{c}_1 \rightarrow \mathtt{c}_2$), **Equivalence** ($\mathtt{c}_1 \leftrightarrow \mathtt{c}_2$): Additional boolean operators for logical composition.

Claims can be simple atomic properties, expressions of integer arithmetic, or again claims. By compounding these with the operators above, complex claims about the temporal relationship between properties of a system can be established and proven.

In this paper we will show that an SCXML document with a suitable datamodel can be transformed onto a Promela program, enabling developers to utilize all of SPIN's model-checking techniques.

RELATED WORK

After the introduction of statecharts by David Harel in 1987 [6] as a visual formalism for complex systems, statecharts have gained widespread usage, e.g. through STATEMATE [7] or as part of the Unified Modeling Language (UML) and SCXML.

In this section some of the existing approaches to enable model-checking tools for statecharts, mostly for the operational semantics of STATEMATE and UML, are briefly described. For a more detailed discussion of these approaches and a few others, we refer to the paper of Bhaduri and Ramesh [2].

We have distinguished the approaches into two groups, depending on the model-checking tool they aim for. The two

EICS'14 Workshop, Engineering Interactive Systems with SCXML, June 17, 2014, Rome, Italy

most addressed model-checking tools are PROMELA/SPIN, already described above, and the SMV system [11].

The SMV system is a model-checking tool for checking finite state machines (FSMs) against specifications in the temporal logic CTL. The tool uses BDDs[3] for the representation of state sets and transitions relations and a symbolic model-checking technique for verification.

State-Charts to SMV

One of the first approaches for translating state-charts into SMV code can be found in Chan et al.[4]. The authors are using the Requirements State Machine Language (RSML) [10], a variation of Harel state-charts, as basis for the translation to SMV code. While this translation scheme is working for deterministic state-charts, the translations do not preserve the semantics of non-deterministic ones. Furthermore, RSML has no priority scheme for resolving certain conflicting transitions, history connectors, synchronizations through activities and optional trigger events.

Another approach can be found in [5]. The authors are translating STATEMATE state-charts to SVM using the temporal language ETL. The approach attempts to reflect the hierarchical structure of state-charts as close as possible in SVM in order to obtain a *fully abstract* or *modular* translation. As there is no subroutine style hierarchical composition of modules in SVM, only AND-hierarchy of state-charts can be modeled by this translation. Furthermore, with the modular translation interlevel-transitions and the PROMELA priority scheme for conflicting transitions can not be handled.

State-Charts to PROMELA/SPIN

In [13] the authors are using *extended hierarchical automata* (EHA[12]) as an intermediate format for their translation to the PROMELA language. The translation is based on the operational semantics of STATEMATE and the semantics of an EHA is given in terms of a Kripke structure. In the paper two translations frameworks are presented resulting in sequential or parallel PROMELA code. For different reasons, the authors have restricted themselves to a subset of state-charts. Data transformations, history and timing issues are not considered. Additionally the transition labels are restricted as follows:

1. Only boolean combinations of predicates *in(st)* are allowed in expression *Cond*

2. The only effect of taking a transition is the generation of events.

Another very similar approach using EHAs as an intermediate format for the translation can be found in the paper from Latella et al.[8]. This translation is based on the operational semantics of UML [9] instead of PROMELA's and therefor slightly modified. Like the previous approach, the considered subset of state-charts does not include history, activity states or actions. Furthermore, time and change events, object creation and destruction events and deferred events and branch transitions are not considered. As data and variables are not considered, actions can only generate events.

APPROACH

Our approach to transform SCXML documents onto Promela programs is divided in two steps:

1. *Flatten* the SCXML document into an equivalent document without any parallel, nested or history states. This, essentially, transforms the state-chart into a state-machine as only a single state can only ever be active.

2. Transform the state-machine onto a Promela program where we can use the model-checking techniques of SPIN.

The first step is completely agnostic of the datamodel and just syntactically transforms the SCXML document. We do loose some expressiveness but can account for most by extending the interpreter slightly, this is discussed in detail later. In fact, all tests from the SCXML IRP suite for the ECMAScript datamodel still pass after being transformed and interpreted with slight modification of the interpreter (with the exception of a few unrelated tests already failing with the original document).

In the second step, SCXML documents employing the `promela` datamodel can be transformed onto Promela programs for the SPIN model-checker. This datamodel is rather restrictive as the Promela language only has very limited expressiveness.

SCXML STATE-CHARTS TO STATE-MACHINES

To transform the Harel state-charts into state-machines, we use a power set construction similar to the one employed when creating deterministic from non-deterministic finite automatons. As there are no formal operational semantics for SCXML, we took a pragmatic approach wherein we use the interpreter itself to create an equivalent flattened document. This shifts the problem of operational semantics onto the existence of a compliant interpreter as the resulting documents will exhibit the same behavior as the SCXML interpreter we used for the transformation.

For the following formalization, we will use definitions from the SCXML standard for readability. The reader is encouraged to refer to the standard itself.

Global States

First we need to define what constitutes a *global state* in SCXML. We can ignore the state of the embedded datamodel: as long as we process the same set of statements in the same order, the datamodel's internal state will be the same as with the state-chart representation. We encode an SCXML interpreters global state S_g at a given time t as follows:

$$S_a(t) := \{s \mid s \in \text{current configuration}\} \quad (1)$$

$$S_v(t) := \{s \mid s \in S_a(t_2), t_2 < t\} \quad (2)$$

$$S_h(t,i) := \{\text{history of } s_i \text{ at time } t\} \quad (3)$$

$$S_h(t) := (S_h(t,1), .., S_h(t,H)) \quad (4)$$

$$S_g(t) := (S_a(t), S_v(t), S_h(t)) \quad (5)$$

Where $S_a(t)$ is the set of active states at a given time, $S_v(t)$ is the set of states we already visited at least once before and

$S_h(t)$ is the set of states to be reentered per history state in the SCXML document.

This construction leads us to:

LEMMA 1. *The machine's configuration will only ever contain a single active state per step. This is obvious as we will have neither nested nor parallel states per construction.*

Global Transitions

For every global state, we need to establish all optimally enabled sets of transitions that can occur in this configuration. As we cannot assume anything about the state of the data-model, we will need a construction where the first enabled transition with a matching condition represents the correct set of transitions from the original state-chart. To ease the formulae we will assume that each transition has at most a single event descriptor, without loss of generality, as we can easily bring a transition into this form by duplicating it for each descriptor.

We start by gathering all transitions from the active configuration's states, combine, filter, and sort them. Let g be any global state from $S_g(t)$:

$$T_g := \{t \mid t.source \in S_a(g), t \text{ a transition}\} \quad (6)$$

$$\mathcal{P}(T_g) := \{(z_1, .., z_K) \mid z_i \in T_g, 1 \leq K \leq |T_g|\} \quad (7)$$

Initially, $\mathcal{P}(T_g)$ will contain the power set of every combination of transitions in the current configuration in document order for a total of $2^{|T_g|}$ sets. Some of these transition sets are invalid as they could never form an optimally enabled set for a given event name.

Looking at the selection of transitions from the SCXML standard and its execution semantics, there are several criteria to reduce $\mathcal{P}(T_g)$:

$$Inv_1 := \{T \in \mathcal{P}(T_g) \mid \forall t_i, t_j \in T, i \neq j, \nexists e, e \neq \epsilon : e \in t_i.event \wedge e \in t_j.event\} \quad (8)$$

$$Inv_2 := \{T \in \mathcal{P}(T_g) \mid \exists t_i, t_j \in T : t_i.source \supset t_j.source, t_i.event \subseteq t_j.event\} \quad (9)$$

$$Inv_3 := \{T \in \mathcal{P}(T_g) \mid \exists t_i, t_j \in T : t_i \text{ preempts } t_j\} \quad (10)$$

$$Inv_4 := \{T \in \mathcal{P}(T_g) \mid \exists t_i, t_j \in T : t_i.event = \epsilon \wedge t_j.event \neq \epsilon\} \quad (11)$$

$$\mathcal{R}(g) := \mathcal{P}(T_g) \setminus \{Inv_1 \cup Inv_2 \cup Inv_3 \cup Inv_4\} \quad (12)$$

Equation 8 invalidates a set if there is no event name that would enable all of its constituting transitions as such a set can never be enabled. The next equation (9) identifies sets that contain two nested transitions where the inner will always be enabled whenever the outer is enabled. Such a set cannot exist as SCXML would only trigger the deepest enabled transition per basic state in a configuration. Equation (10) filters sets that contain two transitions with a non-empty intersection of their respective exit sets. This is known as transition preemption in the SCXML standard - we can just drop those as we operate on the power set of all potentially enabled transitions. The last equation (11) drops those sets that mix eventful and eventless transitions. They can never occur together as they are taken in different processing steps of the SCXML interpretation algorithm (macro- vs microstep).

At this point we have all potential optimally enabled transition sets and their subsets for the global configuration g in $\mathcal{R}(g)$. We can now aggregate each individual transition set into a new global transition for the current global state as follows: 1. The global transition's `event` attribute is the longest event descriptor from the set, 2. its `cond` attribute is the conjunction of all its individual `cond` attributes, 3. its `target` is detailed in the next section with the actual construction.

We do know that the every event matched by the longest (most specific) event descriptor from a set will be matched by each shorter (less specific) event descriptor as per equation 8. Therefore, event names matching the longest event descriptor will enable all transitions from the original set.

LEMMA 2. *Every global transition from $\mathcal{R}(g)$ is enabled by a given event name if each of its constituting transition's are enabled.*

LEMMA 3. *Only a single global transition will ever be taken per step. All transitions per state conflict pairwise as they all have the active global state in its exit set and there is only one active state per step.*

As we also have the subsets of all potential optimally enabled sets in $\mathcal{R}(g)$, we can construct a global transition's `cond` attribute by syntactically conjuncting the `cond` attributes and sort them by the number of contained transitions. Which will cause the largest set to be selected when interpreting the transformed document later.

LEMMA 4. *Every global transition from $\mathcal{R}(g)$ has its guard evaluate to true if each of its constituting transition's guards are true.*

We conclude by sorting the transition sets in $\mathcal{R}(g)$ as follows:

1. **Most specific event descriptors first**: A transition set enabled by a more specific event descriptor will contain more transitions than those with a less specific descriptor.

2. **Supersets precede subsets**: For those sets enabled by the same event descriptor, supersets need to precede subsets. The subsets are still eligible to be chosen when a transition from the superset contains a `cond` attribute that will evaluate to `false` at runtime.

$$\mathcal{T}(g) := (T_1, .., T_N \mid T_i \in \mathcal{R}(g), N = |\mathcal{R}(g))| \quad (13)$$
$$\forall k, l(1 \leq k < l \leq N) :$$
$$T_k \supset T_l,$$
$$T_k.event \subseteq T_l.event)$$

This results in $\mathcal{T}(g)$ as the sorted set of potential optimally enabled transition sets for the global configuration g and their

subsets with the essential property:

LEMMA 5. *Every global transition in $\mathcal{T}(g)$ is optimally enabled iff its constituting transitions are optimally enabled.*

Construction

Now that we defined an interpreters *global state* and its respective transitions, we can construct the state machine. To illustrate the approach, we will start by assuming that the interpreter is already in the initial stable configuration. As mentioned earlier, we actually use a standards compliant interpreter to help with the transformation by intercepting various calls: (i) Whenever the interpreter were to interpret executable SCXML content, (ii) whenever an external component were to be invoked or cancelled, (iii) every entry or exit of a state and before a transition were to be taken, (iv) as well as all processing of `<donedata>` when a final state (also from compound states) was reached.

When the interpreter is in a stable configuration g, we construct the sorted set of potentially optimal enabled transitions $\mathcal{T}(g)$ as explained above. For each global transition in this set, we perform a microstep for its constituting transitions, causing the interpreter to process the implied event by (i) exiting the states from the transition's exit sets and interpreting their `<onexit>` handlers, (ii) interpreting the transition's executable content, (iii) processing any `<datamodel>` elements when the data binding is `late`, (iv) interpreting the transition's entry sets `<onentry>` handlers and (v) invoking any external component activated by the new configuration.

If this leads to a new global state, we will repeat the process until all global states were visited. In essence, we perform a depth-first search for all reachable global states. After each microstep for a transition set from $\mathcal{T}(g)$ per global state, we reset the interpreters configuration, visited states and history to their original values to take all transitions as if we started in the original current global state. While the interpreter processes the micostep, we gather the various *actions* we intercepted and associate them with the current global transition.

For the initial transition, we just introduce a new global state with its constituting sets empty and have a single transition from this one to the first actual global state.

When we exhaustively spanned the global state space we can construct the flattened SCXML document: For every global state, we introduce a new state in the flattened SCXML document with only its global transitions from $\mathcal{T}(g)$ as child elements. If there were no actions performed by the interpreter and associated with a global transition, its `target` will just be the global state we reached after performing its microstep earlier. When there were any actions, its `target` will be the start state of a *transient state chain* connected via guardless transitions and ending in the global destination state as before.

Within a transient state chain, we organize all the actions we gathered when we took the global transition's microstep with the original document. For the sake of construction we will just argue to create one transient state in the chain per action encountered during the microstep, when in fact several actions can be aggregated into one transient state (e.g. multiple consecutive `<onexit>` handlers).

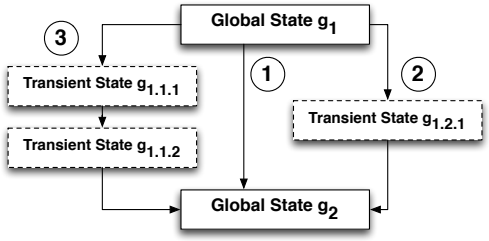

Figure 1. Three global transitions from global states g1 to g2 with 1) no, 2) a single and 3) multiple transient states.

The actions are to be handled in the order they were observed as follows:

1. For each `<onentry>` and `<onexit>` handler encountered, copy it into a transient state.
2. Every executable content from a transition is copied into a new state, either into an `<onentry>` or `<onexit>` handler, it does not matter.
3. For every `<invoke>`, add an attribute `persist="true"` and copy it. We will discuss the extensions required below.
4. For every canceling of an invoker, we add a new `<uninvoke>` with the invokers `id` in a transient state. Again, see below for discussion.
5. Whenever a state was entered and the data binding is set to `late`, copy its eventual `<datamodel>` elements into a transient state iff $S_v(t-1)$ did not already contain the state. Also copy its `<script>` elements.
6. Whenever `<donedata>` was about to be send, add a transient state with a `<raise>` element with the `<donedata>`'s eventual content. Again, see below for discussion.

This construction ensures (i) that the error semantics for executable content remains as with the original document (next `<onentry>`, `<onexit>` or transition block is processed when an error is encountered), (ii) data with late binding is initialized at the correct time, (iii) all statements for the embedded datamodel is executed in the same order and (iv) invokers are started and stopped correctly.

Finally we copy any other global elements from the original `<scxml>` element such as `<script>` or the global `<datamodel>` and write an SCXML file.

Now, there are a few things we implied in the construction that a standards-compliant interpreter cannot do without modifications and some language features that cannot be transformed at all; they are discussed in the following subsections.

The invoke Element

It is not possible to support the `<invoke>` element with a flat SCXML state-machine as the invoked element will only

ever be active when the invoking state is in the current configuration. As traversing the nested compound or parallel states from the original state-chart will cause the interpreter to assume different global configurations, each of those will trigger a state transition in the flattened document, causing the invoked component to be canceled. Having `<invoke>` in every global state that contains the respective original state still causes the invoked component to be continuously invoked and canceled.

As implied earlier, we extended the interpreter to support an additional attribute `persist` with `<invoke>` elements, causing them to remain invoked until an `<uninvoke>` with the invoker's id is encountered in a state entered.

The donedata Element

Whenever a compliant interpreter enters a final state, i.e. of a compound state, it will raise an internal `done.<stateid>` event. This is realized by raising a respective event in a global transition's transient state chain, but if the final state contained a `<donedata>` element, its contents are to be sent along with the internal event. The `<raise>` element in SCXML does not support to specify content like the `<send>` element does. We just extended our interpreter for `<raise>` to be a `<send>` to the internal event queue.

The in() Predicate

With all states from the original SCXML state-chart document being aggregated into global states, their names changed, causing the `In()` predicate to fail. As we still encode all states from $S_a(t)$ in the global state's identifiers and the transient states respectively, we can easily support this predicate by having it parse the set of active states from the current identifier.

SCXML STATE-MACHINES TO PROMELA

The sections above described a construction to transform a large subset of SCXML documents into equivalent documents without nested, parallel or history states, regardless of the employed datamodel. In this section we will introduce the `promela` datamodel, which enables the transformation of such a state-machine onto a Promela program as input for the SPIN model-checker.

While the intermediate step of constructing a state-machine as decribed above might not be strictly necessary to express an equivalent system in Promela (cf. extended hierarchical automatons), it helps to trivialize the transformation which is important in the absence of formal operational semantics.

The Promela Language

In order to decide the set of language features for a `promela` datamodel to support, we first need to have a brief discussion about the Promela language itself and the workings of model-checking with SPIN.

The Promela language itself is already rather restricted as it will implicitly be transformed onto a Kripke structure as a transition system with a label function $\mathcal{K} := (S, I, T, L)$. Where S is a set of states, $I \in S$ is the initial state, $T \in S \times S$ the set of transitions and $L : S -> 2^P$ a label function to associate properties with a given state.

A system in Promela is modeled as a set of concurrent processes, passing events via channels. In an exhaustive search, every possible interleaving of statements of these processes is, as an execution sequence, validated for one of the criteria given in the introduction. Statements can be grouped into an `atomic` block to prevent them from being interleaved by statements from another process in an execution sequence.

The only datatypes in Promela are booleans and integer values of varying sizes, there is no notion of strings. There are the usual constructs for control flow, such as loops and conditional execution. For the analysis, labels on statements play a prominent role, e.g. it is possible to label a statement as `progress`, causing the model-checker to consider the execution sequence to make progress if it will always pass such a statement sometime in the future.

A Promela program, per convention, starts by running the process called `init`, which can spawn other concurrent processes. To synchronize processes, channels of varying length as simple FIFO queues are available where values can be pushed into and popped from.

A Finite State Machine in Promela

In preparation of the construction below, we exemplified an implementation for a state-machine modeled in Promela in listing 1.

```
/* event descriptors and their prefixes */
#define e1   0
#define e11  1
#define e2   2
#define e3   3

/* global states */
#define s1   0
#define s2   1
#define s3   2

int e;                    /* current event */
int s;                    /* current state */
chan iQ = [100] of {int} /* internal queue */
chan eQ = [100] of {int} /* external queue */
bit doneEventSource1;     /* stop event source */
...

proctype step() { /* state machine process */
  /* initial transition's statements */
  atomic { ...
    s=s1; /* set initial state */
  }
  goto: nextStep;

  /* statements per global transition */
  t1ExecContent:
  atomic { ...
    s=s2; /* Update current state */
    iQ!e3 /* push events as part of <raise> */
  }
  eQ!e3 /* push events as part of <send> */
  goto: nextStep;

  t2ExecContent: ...
  goto: done;

  nextStep: /* pop an event */
  if
  :: empty(iQ) -> eQ ? e /* from external queue */
  :: else -> iQ ? e      /* from internal queue */
  fi

  /* event dispatching per state */
  if
```

```
    :: (s==s1 & e==e1) -> goto t1ExecContent;
    :: (s==s2 & e==e2) -> goto t2ExecContent;
    :: else -> goto nextStep;
  fi;
  /* stop event sources and return */
  done: doneEventSource1 = 1; ...
}

/* an external event source */
proctype eventSource1() {
  doneEventSource1 = 0;
  newEvent:
  if
    :: doneEventSource1 -> skip;
    /* push random event sequence */
    :: eQ!e1;       goto newEvent;
    :: eQ!e11;      goto newEvent;
    :: eQ!e2; Q!e3; goto newEvent;
  fi;
}
... /* other event sources */

init() {
  run step();
  run eventSource1();
}
```

Listing 1. A SCXML state-machine in Promela

Using this state-machine as a template, one can already see how a given SCXML state-machine with a suitable datamodel can be transformed into a Promela state-machine. The set of event descriptors for the SCXML state-machine is encoded as an equivalent class of events, a global state is similarly represented as an integer value. The event queues are simple FIFO message channels of sufficient length (long enough to contain all event sequence permutations). The procedure `init` will start `step` and the external event sources (here `eventSource1`), wherein we raise events for the external queue. After the processing of statements for the initial set of transitions from the SCXML state-machine we try to pop an event. If the internal queue is empty, we try to (blockingly) read an event from the external queue.

Then we dispatch the event with respect to the current state and the event's name by executing its global transition's statements from its transient state chain introduced earlier in an `atomic` block. We continue to do so until one transition leads to a document-final state.

The event dispatching as implied above has one fatal flaw: In an exhaustive search *every* condition that is true will lead to a new execution sequence. That is, at analysis time we will get false reports from conditions deeper down in the list that were also true but not meant to be taken (i.e. subsets of the optimally enabled transition set). The solution is to use nested `if / else` blocks for every condition per state.

In SCXML, there is no notion of *event envelopes*: An external event sequence raised by some component can be interleaved by events raised by other components. The processing of an event, however, is performed exclusively, which is reflected by the `atomic` blocks. Events *raised* for the internal queue can be embedded in the atomic block as they cannot be interleaved by other events. Whereas events *send* to our own external queue, have to be enqueued outside of the `atomic` block. Event delays are not modeled as every possible sequence of events will be created in an exhaustive search by SPIN.

The `eventSource1` will enqueue any sequences of events as they can be delivered by external systems (e.g. a parent SCXML document or via `basichttp`). It is, again, important to think about the possible sequences of events and whether interleaving can occur. The easiest solution is to have one concurrent process per external event source, enqueue event sequences as they can occur and let SPIN handle the interleaving.

Promela Datamodel

Now that we have an idea how a state-machine can be expressed in Promela, we can argue about the language features which we can introduce via a datamodel into the SCXML runtime while still being able to transform it. Such a datamodel will enable developers to write SCXML documents with a behavior that can be proven via SPIN and interpreted by an SCXML interpreter.

We seperate the datamodel's features into the various SCXML language features where it is relevant and introduce a subset of the Promela language for each. The Promela language as such is given as a YACC grammar with a hand-written lexer in the SPIN distribution. By isolating the various production rules and a subset of their children, we allow an application developer to use subsets of the actual Promela syntax.

The Promela runtime as implemented in the SPIN model-checker is, unfortunately, unsuited to be embedded as such as a scripting language into an SCXML interpreter: (i) The parser is not reentrant, only allowing a single expression to be parsed at a time, (ii) global variables are used in the parser as well as the actual runtime, (iii) generic function names pollute the global namespace unacceptably. All of which are perfectly fine, for a stand-alone program but unsuited for an SCXML interpreter when potentially invoking multiple nested SCXML interpreters with the `promela` datamodel. This necessitates our datamodel to reimplement the semantics for the language features we will support when interpreted as part of an SCXML document.

Furthermore, some Promela statements will cause the SPIN model-checker to *branch out* into every possible execution sequence, a feature without an equivalent counterpart for a SCXML interpreter and something we have to consider when providing language features in the datamodel: We only want the Promela program to consider every possible sequence of events, not indeterminism introduced by e.g. ambiguous control flow statements.

Data Element

The `<data>` element can occur as a child of `<datamodel>` in SCXML states and allows to declare variables. With a late data `binding`, these are only introduced when their respective parent state is entered for the first time. There is only limited support for variable scopes in Promela, variables can be *local* to a procedure, *hidden* with regard to the program's state or global. As neither of these supports the semantics of the SCXML `<data>` element with a late binding, we will only support early data bindings as global Promela variables.

Within a <data> element, we allow developers to write any sequence of statements that can be reduced to Promela declaration lists (decl_lst rule from the Promela grammar) with the exception of user defined types and channel declarations.

What remains is the declaration and initial assignment of all Promela native types as atoms and arrays with a fixed size. These expressions will just be copied into the head of the resulting Promela source file right after the declaration of states and events.

Assign Element

Within the <assign> element, a developer can provide assignments that can be reduced via the Promela grammar's assign rule. If the <assign> element has an id attribute its content is supposed to be an expression, if not an actual assignment is expected.

Script Element

While it is desirable to allow any sequence of Promela statements in a <script> element, only a subset can be supported. In fact, of all possible statements allowed by the Promela grammar (stmnt rule), we only support assignments and iterations for now, as most others will cause the SPIN model-checker to potentially branch out. Remember that all these statements will end up in the respective atomic block of a global transition.

It is conceivable to support some other statements as well, but at the end, they all are ultimately used to assign values to variables and we prefer to e.g. handle control flow via the state-machine.

Attribute cond

For the cond attribute, we will allow a subset of expressions (expr rule) that can be evaluated as a boolean value. We do not support operations on message queues or those that refer to Promela's execution process (pid) in these attributes.

Evaluate as String

There are several situation with an SCXML datamodel, where an expression is supposed to be evaluated as a string. As there are no strings in Promela, it is not possible to support these in any meaningful way. At the moment, evaluating a Promela expression as a string will return a string representation of its integer or boolean value. Evaluating an array will yield a JSON structure with an array.

This has severe consequences as we cannot, in any meaningful way, represent an event's data, the interpreter's name or session identifier, nor e.g. the location of the basichttp I/O processor.

It is conceivable to introduce *string literals* and encode them as integer values: Whenever the datamodel encounters a string as part of an expression, it would introduce a new literal and assign an integer value. This might improve expressiveness but we did not research this any further.

Foreach Element

As variable arrays and ranges are available in the Promela syntax, there is a straight-forward semantics for the <foreach> element. The array attribute is either a variable reference or two arithmetic expressions separated by two dots. The item attribute is just a variable reference.

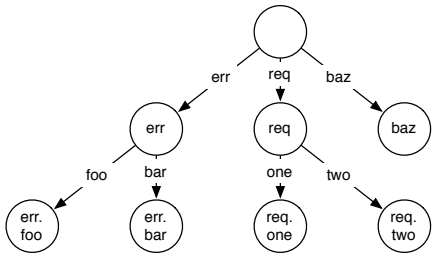

Figure 2. Event descriptor's prefix tree.

Construction

Using the template in listing 1, we already have an idea how we could express a state-machine in Promela. In the following section we will detail how the remaining SCXML language features can be transformed into a Promela program.

Event Names and Dispatching

We do not need to know the name of every possible event that is eventually passed into the interpreter, only the set of event descriptors. We start by building a prefix tree from all event descriptors at the global transition's event attribute at transformation time (see figure 2). Here, a symbol does not correspond to a single character, but to each sequence of characters separated by a dot. That is, error and errFoo are prefix free, whereas err.or and err.Foo are not. This corresponds to the event name matching for descriptors from the SCXML standard.

Every event that is to be represented in the Promela program is expressed or transformed to its deepest matching node in the prefix tree, where any remaining suffix is dropped. When dispatching events, every global transition that is enabled by an event encoding a given node in the prefix tree is also enabled by all its children.

This will cause global transitions enabled by e.g. error to be selected in the Promela program for every event name that start with its event descriptor, mimicking the behavior from the SCXML standard.

As we cannot, in any meaningful way, support an event's data, two events with the same name in the original SCXML document but handled differently with respect to their data will have to be separated by an application developer when writing for the promela datamodel. It is conceivable to provide tool support for this differentiation, but for now we will have to assume that an event's name is sufficient to imply the set of statements its processing will entail.

External Event Sources

A state-machine in itself might already be required to be proven for correctness but the more interesting and general approach is enabled by allowing external components to pass event sequences into the machine.

External components will send sequences of events to an interpreter's external queue and we can model them in Promela as concurrent processes, enqueuing events or sequences of events to the external queue. By having SPIN handle the interleaving of the statements in the concurrent processes, we will validate for all possible event sequences.

Now, we cannot know the external components when transforming the SCXML document into a Promela program, it might be a HTTP client passing events via the `basichttp` I/O processor, so we need for an application developer to specify them. This is done by introducing special XML comments:

```
<!-- promela-event-source:
  e1, e2, e2
  e1, e3
-->
```

These are valid children of the `<scxml>` and `<invoke>` elements and will cause the resulting Promela program to contain a procedure enqueuing the event sequences separated by newlines onto the external queue. When supplied within the `<invoke>` element, their respective processes will be started and stopped as the invoked component would.

The if / else / elseif Elements

There is an obvious but wrong approach to express conditional control flow for SCXML `<if>` / `<elseif>` / `<else>` blocks in Promela. As with the selection of transitions during event dispatching, every true condition in a Promela `if` statement will cause the SPIN interpreter to branch out. Therefore, we need to, again, nest `<elseif>` elements as `if` statements. The Promela `else` statement will only be considered if none of the other conditions are eligible.

The raise / send Elements

We already had a brief discussion about `<send>` and `<raise>` as part of executable content in a global transition's transient state chain. The important point is that events *raised* cannot be interleaved by other events so we can enqueue their encoded prefix-tree node within a global transitions `atomic` block onto our internal queue, whereas events *send* to ourself need to be enqueued to our external queue after we left the atomic block.

There is one problem with this approach though, the set of events to be send or raised eventually depends on the conditional interpretation of `<if>` / `<elseif>` / `<else>` blocks. To nevertheless keep the transition's `atomic` block intact, we enqueue events to be send to our external queue in a temporary queue and move them into the external queue, when we left the block.

```
...
/* statements per global transition */
tExecContent:
atomic {
  if
    :: expr1 -> { tmpQ!e3; } /* send */
    :: expr2 -> { iQ!e3; }   /* raise */
    ...
  fi
  ...
}
/* push send events to external queue
   here to allow interleaving */
for (tmpQItem in tmpQ) {
  eQ!tmpQItem;
}
goto: nextStep;
```

The foreach Element

We choose expressiveness for the `<foreach>` element with our `promela` datamodel to have a simple equivalent in a Promela program. The `array` attribute is taken as a range or a Promela array and assigned to the global variable introduced in `item` for each iteration.

ANALYSIS WITH SPIN

In order to analyze a Promela program, SPIN relies upon labeled statements: When an execution sequence will always eventually pass a statement preceded by a *progress label*, the sequence is considered to make progress, similarly with *acceptance-* and *end labels*. In order to introduce such labels into the Promela program, we allow developers to write special XML comments within executable SCXML content that are copied verbatim into the atomic block of transitions that caused them to be interpreted:

```
<!-- promela-inline:
  progress: skip;
-->
```

The `skip` statement here is side-effect free and always executable. When a `promela-inline` comment is a child of the `<scxml>` element, its contents are copied verbatim into the programs body after the variable declarations. This feature can be used to i.e. introduce LTL claims. In fact, a developer can introduce any Promela code into a global transition's `atomic` block or the program's body. This allows to customize the Promela program considerably and it is the responsibility of the developer to ensure that the resulting program still reflects the behavior of the original SCXML document.

CONCLUSION

We described a new `promela` datamodel and a two step construction to transform a large subset of SCXML documents employing this datamodel into equivalent Promela programs. This allows for a formal verification of such SCXML documents with respect to their properties along all possible event sequences.

While the expressiveness of the datamodel is rather limited (e.g. no strings and as such no data attached to an event) it is very suited to be employed as a formally proven subsystem when used in a nested, invoked SCXML interpreter from a parent interpreter with a more expressive datamodel.

We extended the approaches described in related work with support for `<history>` states, external events and an actual implementation as part of our SCXML interpreter.

ACKNOWLEDGMENTS

This work has been partially supported by the FP7 EU large-scale integrating project SMART VORTEX (Scalable Semantic Product Data Stream Management for Collaboration and Decision Making in Engineering) co-financed by

the European Union. For more details, visit `http://www.smartvortex.eu/`.

REFERENCES

1. Barnett, J., Akolkar, R., Auburn, R., Bodell, M., Burnett, D. C., Carter, J., McGlashan, S., Lager, T., Helbing, M., Hosn, R., Raman, T., Reifenrath, K., and Rosenthal, N. State chart XML (SCXML): State machine notation for control abstraction. W3C working draft, W3C, Mar. 2014. http://www.w3.org/TR/2014/CR-scxml-20140313/.
2. Bhaduri, P., and Ramesh, S. Model checking of statechart models: Survey and research directions. *ArXiv Computer Science e-prints* (July 2004).
3. Bryant, R. E. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput. 35*, 8 (Aug. 1986), 677–691.
4. Chan, W., Anderson, R. J., Beame, P., Burns, S., Modugno, F., Notkin, D., and Reese, J. D. Model checking large software specifications. *IEEE Trans. Softw. Eng. 24*, 7 (July 1998), 498–520.
5. Clarke, E. M., and Heinle, W. Modular translation of statecharts to smv. Tech. rep., 2000.
6. Harel, D. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program. 8*, 3 (June 1987), 231–274.
7. Harel, D., Lachover, H., Naamad, A., Pnueli, A., Politi, M., Sherman, R., and Trauring, S. A. Statemate: a working environment for the development of complex reactive systems (1988). 1–3.
8. Latella, D., Majzik, I., and Massink, M. Automatic verification of a behavioural subset of uml statechart diagrams using the spin model-checker. *Formal Asp. Comput. 11*, 6 (1999), 637–664.
9. Latella, D., Majzik, I., and Massink, M. Towards a formal operational semantics of uml statechart diagrams. In *FMOODS*, P. Ciancarini, A. Fantechi, R. Gorrieri, P. Ciancarini, A. Fantechi, and R. Gorrieri, Eds., vol. 139 of *IFIP Conference Proceedings*, Kluwer (1999).
10. Leveson, N. G., Heimdahl, M. P. E., Hildreth, H., and Reese, J. D. Requirements specification for process-control systems. *IEEE Trans. Software Eng. 20*, 9 (1994), 684–707.
11. McMillan, K. L. *Symbolic Model Checking: An Approach to the State Explosion Problem*. PhD thesis, Pittsburgh, PA, USA, 1992. UMI Order No. GAX92-24209.
12. Mikk, E., Lakhnech, Y., and Siegel, M. Hierarchical automata as model for statecharts. In *ASIAN*, R. K. Shyamasundar and K. Ueda, Eds., vol. 1345 of *Lecture Notes in Computer Science*, Springer (1997), 181–196.
13. Mikk, E., Lakhnech, Y., Siegel, M., and Holzmann, G. J. Implementing statecharts in promela/spin (1998). 90–101.