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.

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 the interactivity 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.

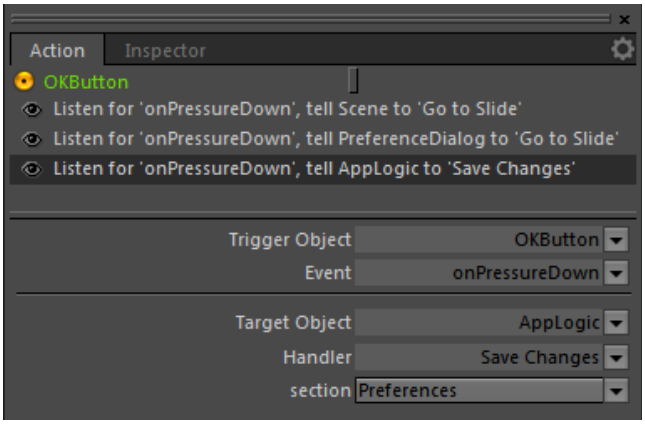

Figure 1: Scriptless Actions in UI Composer Studio

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.
2. Artists would be prevented 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 expression 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 problem of the combinatorial explosion 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 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 visual specifications 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 now 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 understanding the SCXML specification and typing raw, syntactically valid XML was too cumbersome and error-prone. Once we decided to use SCXML we needed to write an editor to easily create valid markup, and 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 shows 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

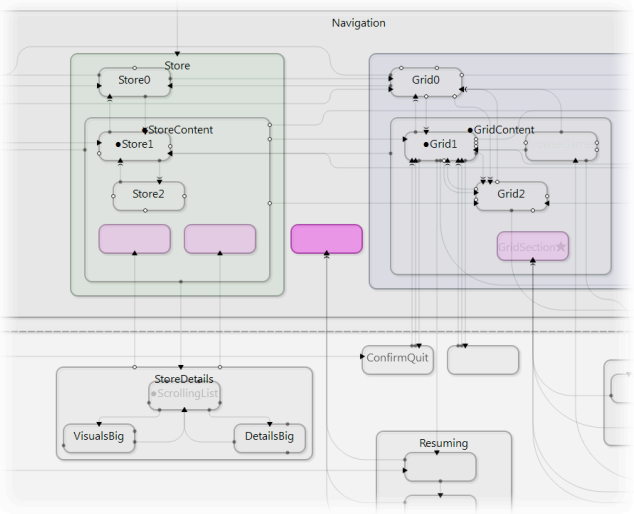

Figure 2: Portion of a production statechart created with Architect, with transition event labels hidden

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 representation 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 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. The circled **(H)** and **(H)*** notations used by Harel do not allow sufficient distinction between history states within the same parent state. We instead display the full name of history states. We also append a star glyph to the name to visually differentiate them from normal states, using different glyphs depending on whether the history is shallow or deep.

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. However, if we prevent a child from moving outside the boundaries of the parent, a "claustrophobic" feel is introduced, forcing users to fight the system. 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. An 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 creating an '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, there is an alternative mode 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 visually differentiate it.

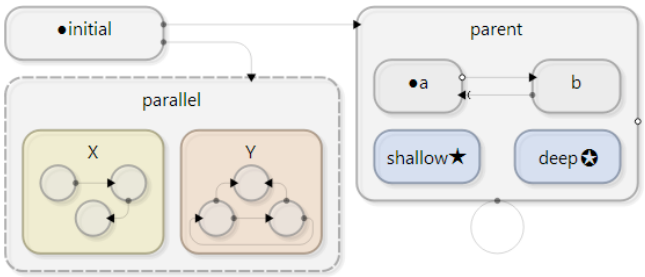

Figure 3: Examples of states and transitions in Architect

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. An example of this is shown 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 occur.

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 turn.

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 others, results 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. However, SCXML alternatively allows 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 node (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 is too likely that an intended visual-only edit to transition routing could 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 might specify the interface changes as executable content in a custom namespace. However, our automotive customers would like to be able to re-use a single state machine with multiple 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 must separate presentation completely 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 possible: 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.

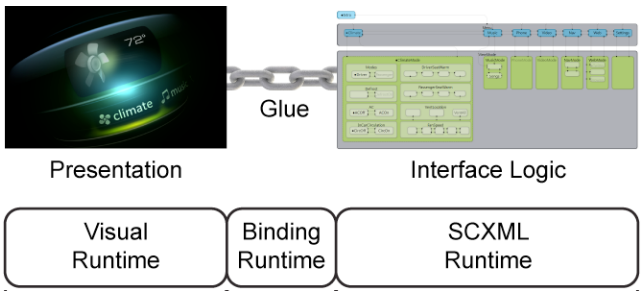

Figure 4: Gluing the Presentation to the Logic

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, 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 early exit of the state cancels the queued event. Figure 5 shows authoring such a situation in Architect.

RUNTIME IMPLEMENTATION

Beyond editing the interface and logic, and gluing them together, the final piece needed for application support is an 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.

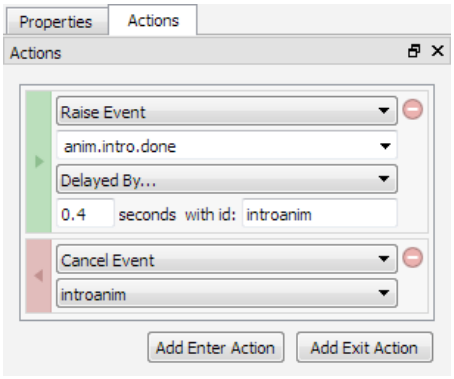

Figure 5: Firing an event after a timeout.

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 use SCXML we wrote our official engine 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, 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. There exist other mechanisms to accomplish the same goal in many cases, for example 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 these 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 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.

By integrating 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.

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.

Using SCXML as the representation of the state machine is seen as a benefit to our customers. 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. 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. 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," in 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. Mobile SVG Profiles: SVG Tiny and SVG Basic. http://www.w3.org/TR/2003/REC-SVGMobile-20030114/