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 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

CHI'12, May 5–10, 2012, Austin, Texas, USA.

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}

if $BP_{user}.phase \neq BP_{qual}.phase$ **then**
  /* Not referencing the same phase */
  return false;
end
/* Iterate every field for the phase */
for $field \in FieldsFor(BP_{qual}.phase)$ **do**
  if $BP_{user}.[field] =$ **undef then**
    /* No value for field specified */
    continue;
  end
  if $field =$ "$eventName$" **then**
    /* Event names are matched per SCXML draft as descriptors */
    if not $nameMatch$ $(BP_{qual}.eventName, BP_{user}.eventName)$ **then**
      return false;
    end
    continue;
  end
  if $field =$ "$xpath$" **then**
    /* Field contains an XPath expression */
    if not $NodeSet(BP_{user}.[field])$ $.contains(BP_{qual}.element))$ **then**
      return false;
    end
    continue;
  end
  /* Rest is matched literally */
  if $BP_{user}.[field] \neq BP_{qual}.[field]$ **then**
    return false;
  end
end
/* Check the condition on the datamodel */
if $BP_{user}.condition \neq$ **undef then**
  if not $evalAsBool(BP_{user}.condition)$ **then**
    return false;
  end
end
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

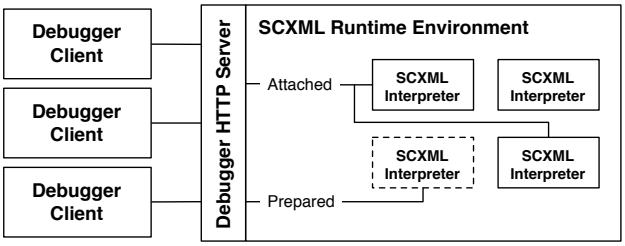

Figure 1. Debugging Architecture.

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.

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 the HTML-based 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

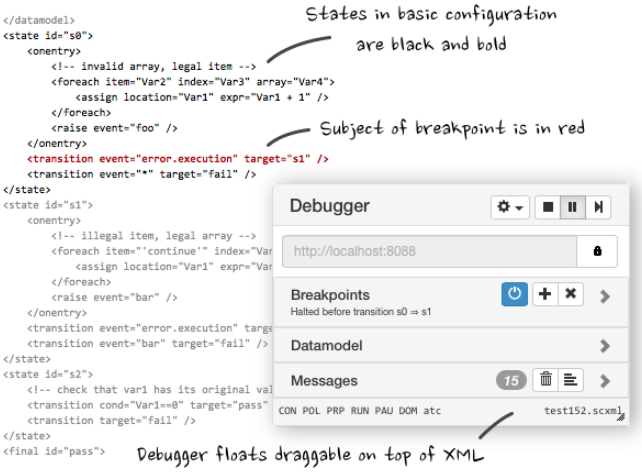

Figure 3. SCXML Document is displayed beneath Debugger.

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).