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).

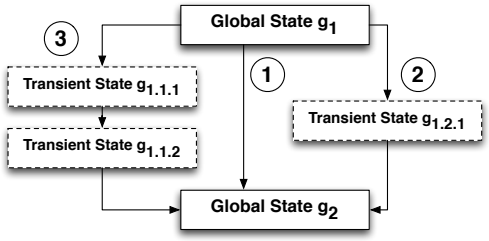

Figure 1. Three global transitions from global states g1 to g2 with 1) no, 2) a single and 3) multiple transient states.

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.

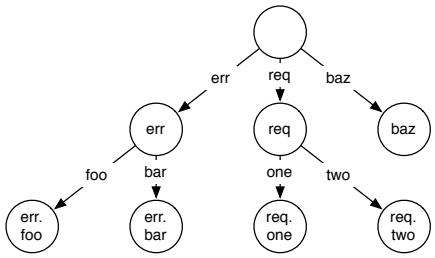

Figure 2. Event descriptor's prefix tree.

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.