Refinement of SCXML state-charts via translation to Event-B

C. Snook[1], K.Morris[2], M.Butler[1], and R.Armstrong[2]

[1] University of Southampton, Southampton, United Kingdom
cfs@ecs.soton.ac.uk
[2] Sandia National Laboratories, Livermore, California, U.S.A.
knmorri@sandia.gov

Abstract. State-chart modelling notations, such as State Chart eXtensible Markup Language (SCXML), with so-called 'run to completion' semantics and simulation tools for validation, are popular with engineers for designing machines. However, they do not support refinement and they lack formal verification methods and tools. Properties concerning the synchronisation between different parts of a machine may be difficult to verify for all scenarios. Event-B, on the other hand, is based on refinement from an initial abstraction and is designed to make formal verification by automatic theorem provers feasible, obviating the need for instantiation and testing. We would like to combine the best of both approaches by incorporating a notion of refinement, similar to that of Event-B, into SCXML and leveraging Event-B's tool support for proof. We describe some the pitfalls in translating 'run to completion' models to Event-B refinements, and suggest a solution and propose extensions to the SCXML syntax to describe refinements. We illustrate the approach using our prototype translation tools and show by example, how a synchronisation property between parallel state-charts can be automatically proven at an incomplete refinement level by translation into Event-B.

Keywords: SCXML, State-charts, Event-B, iUML-B, refinement

1 Introduction

Formal verification of high consequence systems requires the analysis of formal models that capture the properties and functionality of the system of interest. Proof obligations for systems' properties or requirements can be made more tractable using refinement, where properties are expressed in terms of variables that are introduced at different abstraction levels.

A hierarchical development of a system model uses refinement concepts to link the different levels of abstraction. Each subsequent level increases model complexity by adding details in the form of functionality and implementation method. As the model complexity increases in each refinement level, tractability of the detailed model can be improved by the use of a graphical representation, with rich semantics that can support an infrastructure for formal verification.

The Event-B language [1] provides the logic and refinement theory required to formally analyze a system model. The open-source Rodin tool [2] provides support for Event-B including automatic theorem provers. iUML-B [4] augments the Event-B language with a graphical interface including state-machines.

2 Background

2.1 SCXML

State Chart eXtensible Markup Language (SCXML) is a modelling language based on Harel state-charts with facilities for adding data elements that are manipulated by transition actions and used in conditions for their firing. SCXML follows the usual 'run to completion' semantics of such state-chart languages, where trigger events[3] may be needed to enable transitions. Trigger events are queued when they are raised and then one is de-queued and consumed by firing all the transitions that it enables, followed by any (un-triggered) transitions that then become enabled due to the change of state caused by the initial transition firing. This is repeated until no transitions are enabled and then the next trigger is de-queued and consumed. There are two kinds of triggers: internal triggers are raised by transitions and external triggers are raised by the environment (spontaneously as far as our model is concerned). An external trigger may only be consumed when the internal trigger queue has been emptied. Listing 1 shows a pseudocode representaion of the run to completion sematics as defined within the latest WC3 Recommendation document **??**. Here IQ and EQ are the internal and external trigger present in the queue respectively.

```
1  while running:
2    while run2completion = false
3      if untriggered_enabled
4        execute(untriggered())
5      elseif IQ /= {}
6        execute(internal(IQ.dequeue))
7      else
8        run2completion = true
9      endif
10   endwhile
11   if EQ /= {}
12     execute(EQ.dequeue)
13     run2completion = false
14   endif
15 endwhile
```

Listing 1: Pseudocode for 'run to completion'

We adopt the commonly used terminology where a single transition is called a *micro-step* and a complete run (between de-queueing external triggers) is referred to as a *macro-step*.

[3] In SCXML the triggers are called 'events', however, we refer to them as 'triggers' to avoid confusion with Event-B

2.2 Event-B

Event-B [1] is a formal method for system development. Main features of Event-B include the use of *refinement* to introduce system details gradually into the formal model. An Event-B model contains two parts: *contexts* and *machines*. Contexts contain *carrier sets*, *constants*, and *axioms* constraining the carrier sets and constants. Machines contain *variables* v, *invariants* $I(v)$ constraining the variables, and *events*. An event comprises a guard denoting its enabled-condition and an action describing how the variables are modified when the event is executed. In general, an event `e` has the following form, where t are the event parameters, $G(t, v)$ is the guard of the event, and $S(t, v)$ is the action of the event.

$$\texttt{e} \;\widehat{=}\; \textbf{any}\ t\ \textbf{where}\ G(t, v)\ \textbf{then}\ S(t, v)\ \textbf{end} \tag{1}$$

In the case where the event has no parameters, we use the following form

$$\texttt{e} \;\widehat{=}\; \textbf{when}\ G(v)\ \textbf{then}\ S(v)\ \textbf{end}\ , \tag{2}$$

and when the event has no parameters and guard, we use

$$\texttt{e} \;\widehat{=}\; \textbf{begin}\ S(v)\ \textbf{end}\ . \tag{3}$$

The action of an event comprises of one or more assignments, each of them has one of the following forms.

$$v := E(t, v) \tag{4}$$
$$v :\in E(t, v) \tag{5}$$
$$v :| P(t, v) \tag{6}$$

Assignments of the form (4) are deterministic, assign value of expression $E(t, v)$ to v. Assignments of the forms (5) and (6) are non-deterministic. (5) assigns any value from the set $E(t, v)$ to v, while (6) assigns any value satisified predicate $P(t, v)$ to v. Note that invariants $I(v)$ are inductive, i.e., they must be *maintained* by all events. This is more strict than general safety properties which hold for all reachable states of the Event-B machine. This is also the difference between verifying the consistency of Event-B machines using theorem proving and model checking (e.g., ProB) techniques: model checkers explore all reachable states of the system while interpreting the invariants as safety properties.

A machine in Event-B corresponds to a transition system where *variables* represent the states and *events* specify the transitions. More information about Event-B can be found in [**?**]. Event-B is supported by the *Rodin platform* (Rodin) [2], an extensible toolkit which includes facilities for modelling, verifying the consistency of models using theorem proving and model checking techniques, and validating models with simulation-based approaches.

In Event-B the run to completion pseudocode of Listing 1 could be represented (somewhat abstractly) as

FireUntriggered :
when
$UC = FALSE$
then
$execute(untriggered())$
end

FireInternallyTriggered :
when
$UC = TRUE$
$IQ \neq \varnothing$
then
$execute(IQ.dequeue)$
$UC := FALSE$
end

FireExternallyTriggered :
when
$UC = TRUE$
$IQ = \varnothing$
$EQ \neq \varnothing$
then
$execute(EQ.dequeue)$
$UC := FALSE$
end

Note that this is an abstract representation where each event would be specialised to select a particular set of transitions that can be fired in parallel and execute would be replaced by actions that encode the state changes made by those transitions. Representing the condition untriggered enabled is cumbersome since we would need to write a conjunction of all the possible untriggered guards. Instead we introduce a dummy untriggered event that is only fired when no other selection of untriggered transitions are available and sets a boolean flag, UC, to indicate that none of the real untriggered events was fired and a trigger needs to be consumed.

2.3 iUML-B State-machines

iUML-B provides a diagrammatic modelling notation for Event-B in the form of state-machines and class diagrams. The diagrammatic models are contained within an Event-B machine and generate or contribute to parts of it. For example a state-machine will automatically generate the Event-B data elements (sets, constants, axioms, variables, and invariants) to implement the states while Event-B events are expected to already exist to represent the transitions. Transitions contribute further guards and actions representing their state change, to the events that they elaborate. A choice of two alternative translation encodings are supported by the iUML-B tools. State-machines are typically refined by adding nested state-machines to states.

Figure 1 shows an example of a state-machine, named **SM**. Here we show a

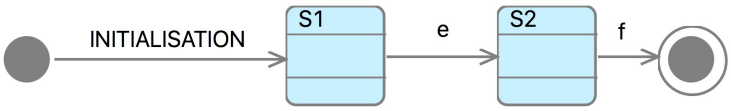
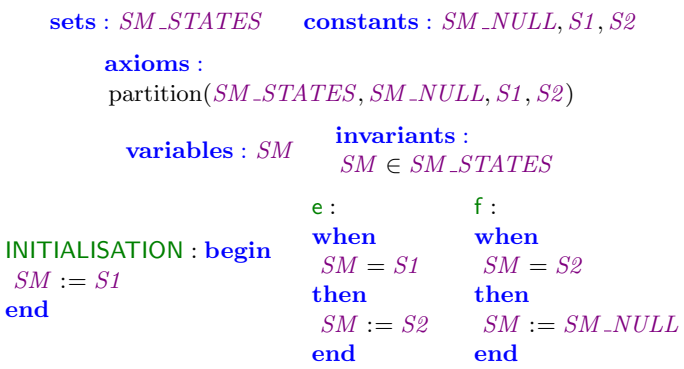


Fig. 1: An example iUML-B state-machine **SM**

translation of state-machine **SM** using the *enumeration* encoding, where each state is encoded as a constant from an enumerated set SM_STATES. Variable SM, which represents the current state of the state-machine, is initialised to $S1$. Events e and f change the value of SM according to the transitions in the state-machine. The Event-B translation can be seen below.



2.4 Intrusion Detection System

The simple intrusion detection system is designed using an Application-Specific Integrated Circuit (ASIC) which connects to a buzzer and a sensor over a Serial Peripheral Interface (SPI) bus. The system is controlled via the ASIC on the SPI bus. At power-up, the ASIC sends commands over the SPI bus to initialize the sensor and the buzzer. After waiting for 50 milliseconds the ASIC enters its main routine, which makes the buzzer respond to the sensor. The statechart model of this system is limited to the ASIC and captures the initialization of the peripherals and the 50 ms wait. In the interest of simplicity we elide the details of the main routine.

The ASIC starts by initializing the buzzer. This involves sending a message over the SPI bus, and at the highest level of abstract we can additional implementation details. Once the message is sent (which will be indicated by some event saying that the SPI system is done), the ASIC moves on to initializing the sensor. After the ASIC moves into a waiting state for 50 ms, and finally moves into the state which represents normal operation.

A subsequent level of refinement adds a parallel state representing the SPI subsystem. The SPI subsystem is usually on an **Idle** state until the **send_message** event occurs, at which point the SPI subsystem enters a state **Sending Message**, which represents sending the message, byte by byte. When the last byte of the message is sent, it raises the **spi_done** event, allowing the other parallel state to continue, while SPI subsystem returns to idle.

The model can be farther refined by incorporating more details on how the initialization states, the wait state, and the SPI subsystem operate, including how they interact with each other. The **Initialize Buzzer** state constructs the SPI message to send, then it raises the **send_message** event, and then it waits. After **send_message** is raised, the SPI subsystem reacts. It spins for a while in the **send_byte** state, looping as many times as it takes to get to the last byte in the message. When the last byte in the message is sent, it goes back to idle and raises an event which allows the state machine on the left to proceed. The sensor is then initialized in a very similar manner to the buzzer. After both peripherals are initialized, the state machine goes into the **Wait 50 ms** state, where it increments a counter until it reaches some maximum, then exits.

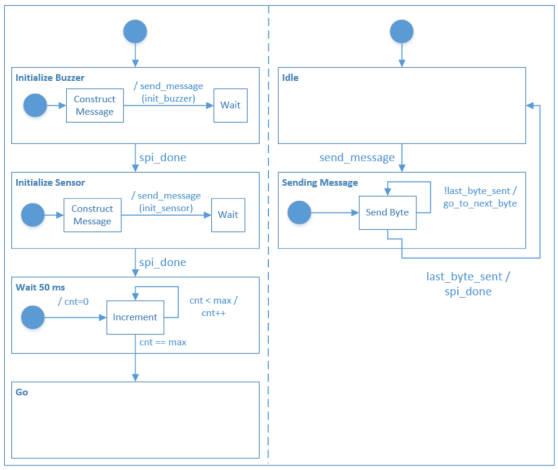


Fig. 2: Statechart diagram for SecBot intrusion detection system

Karla: Not sure if we should include these figures or the IUMLB generated ones. If we include this figure maybe we can simplify it by using color for each of the refinement levels

Colin: introduce the secbot example showing how we would like to develop it in refinements

3 Discussion

In order to introduce a notion of refinement into SCXML we need to consider the kinds of things we would like to do in refinements and what properties should be preserved. In practice we wish to leverage existing Event-B verification tools and hence adopt a notion of refinement that can be automatically translated into an equivalent Event-B model consisting of a chain of refinements. While it might be possible to utilise data refinement by replacing a state-chart with an alternative one, this would greatly complicate things and is impractical when the SCXML model is a single state-chart (rather than a chain of refined models). Hence we start from the following requirements which allow superposition refinements and guard strengthening in SCXML models:

- The firing conditions of a transition can be strengthened by adding further textual constraints about the state of other variables and statemachines in the system.
- The firing conditions of a transition can be strengthened by being more specific about the (nested) source state,
- Nested state-charts can be added in refinements.
- Ancilliary data can be added and corresponding actions to alter it added to transitions.
- Raise actions can be added to transitions to define how internal triggers are raised. These internal triggers may have already been introduced and used to trigger transitions in which case they are non-deterministically raised at the abstract levels. (Note that external triggers are always unguarded and cannot be refined).

– Invariants can be added to states to specify properties that hold while in that state.

Refinement should preserve the value of the abstract state after each micro-step and at the end of each macro-step. The abstract state should not be altered by any new micro-steps that are introduced into an abstract macro-step, nor by any new macro-steps that are introduced. (Note that these goals take the view that macro-steps should align through refinement. An alternative approach that we are considering for future work takes the view that the macro-steps need not align and a micro-step may shift from one macro-step to another in a refinement).

There seems to be an inherent difficulty with refining 'run to completion' semantics which require that every enabled micro step, is completed before the next macro step is started. The problem is that, in a refinement, we want to strengthen the conditions for a micro step. However, by making the micro steps more constrained we disable them and make their completion more easily achieved. This makes the guard for taking the next macro step weaker breaking the notion of refinement.

4 Tooling

A tool to automatically translate SCXML models into iUML-B has been produced. The tool is based on the Eclipse Modelling Framework (EMF) and uses a an SCXML metamodel provided by Sirius [3] which has good support for extensibility. The tooling for iUML-B and Event-B already contains EMF metamodels and provides a generic translator framework which has been specialised for the SCXML to iUML-B translation.

The following syntax extensions are added to SCXML models to support modelling features needed in iUML-B/Event-B. These extensions are prefixed with 'iumlb:' in order to distinguish them from the scxml XML parser. (So that they are ignored by SCXML simulation tools).

– **iumlb:refinement** - an integer attribute representing the refinement level at which the parent element should be introduced.
– **iumlb:invariant** - an element that generates an invariant in iUML-B. This provides a way to add invariants to states so that important properties concerning the synchronisation of state with ancilliary data and other statemachines can be expressed.
– **iumlb:guard** - an element that generates a transition guard in iUML-B. This provides a way to add new guard conditions to transitions over several refinement as well as providing an element with attributes such as derived (for Event-B theroems), name and comment.
– **iumlb:predicate** - a string attribute used for the predicate of a guard or invariant.
– ... other attributes useful for iUML-B elements: name, derived, type, comment.

Hierarchical nested state charts are translated into similarly structured iUML-B state-machines. The generated iUML-B model contains refinements that add nested state-machines as indicated in the SCXML state-chart by the **iumlb:refinement** attributes annotated on state elements. iUML-B transitions are generated for each SCXML transition and linked to Event-B events that represent each of the possible synchronisations that could involve that transition.

5 Example

The SecBot example : pics from slides

Note why it would not be a refinement

6 Related Work

This is the related work...

7 Conclusion

This is the conclusion...

Colin: maybe consider the note below (from our previous plans for a paper)
Note: Things to highlight with the choice of example 1. Look at a system that is better model with SCXML run to completion semantics than iUML-B semantics 2. Look at how you can check for violations of refinement in a SCXML model construction 3. Look at the sort of invariant properties you can verify about a SCXML model

All data supporting this study are openly available from the University of Southampton repository at http://doi.org/10.????/SOTON/D0???

References

1. J-R. Abrial. *Modeling in Event-B: System and Software Engineering.* Cambridge University Press, 2010.
2. J-R. Abrial, M. Butler, S. Hallerstede, T.S. Hoang, F. Mehta, and L. Voisin. Rodin: An open toolset for modelling and reasoning in Event-B. *Software Tools for Technology Transfer*, 12(6):447–466, November 2010.
3. Eclipse Foundation. Sirius project website. https://eclipse.org/sirius/overview.html, 2016.
4. C. Snook. iUML-B statemachines. In *Proceedings of the Rodin Workshop 2014*, Toulouse, France, 2014. http://eprints.soton.ac.uk/365301/.