Refinement of SCXML Statecharts via translation to Event-B

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

[1] University of Southampton, Southampton, United Kingdom
{cfs,t.s.hoang,mjb}@soton.ac.uk
[2] Sandia National Laboratories, Livermore, California, U.S.A.
{knmorri,rob}@sandia.gov

Abstract. Statechart modelling notations with so-called 'run to completion' semantics and simulation tools for validation, are popular with engineers for designing machines. However, they do not support formal refinement and they lack formal static verification methods and tools. For example, properties concerning the synchronisation between different parts of a machine may be difficult to verify for all scenarios, and impossible to verify at an abstract level before the full details of sub-states have been added. 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, restricting instantiation and testing to a validation role. We would like to incorporate a notion of refinement, similar to that of Event-B, into a Statechart modelling notation, *State Chart eXtensible Markup Language* (SCXML) and leverage Event-B's tool support for proof. We describe the pitfalls in translating 'run to completion' models into Event-B refinements, 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 Statecharts can be automatically proven at an intermediate refinement level by translation into Event-B.

Keywords: SCXML, Statecharts, 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. Although high-consequence controls and systems are designed to limit complexity, the requirements and consequent proof obligations tend to increase the complexity of the formal verification. Proof obligations for such requirements can be made more tractable using abstraction/refinement, providing a natural divide and conquer strategy for controlling complexity.

Statecharts [7] are often used for high-consequence controls and other critical systems to provide an unambiguous, executable way of specifying functional as well as safety, security, and reliability properties. While functional properties (usually) can be tested, safety, security and reliability properties (usually) must be proved formally. Here we give a binding from Statecharts to Event-B [1] so that this type of reasoning can be carried out. Moreover, hierarchical encapsulation maps well onto Statecharts in a way that is not very different from previous work in iUML-B [13,14,15], a diagrammatic modelling notation for Event-B. Binding iUML-B to a UML [12] version of Statecharts is natural and the addition of run-to-completion semantics, expected by Statechart designers, is much of the contribution of this work. Another contribution is the augmentation of the textual and parse-able format for Statecharts, *State Chart eXtensible Markup Language* (SCXML) to accommodate elements necessary to support formal analysis.

There are many formal semantics that can be bound to the Statechart graphical language [5]. While Statecharts and various semantic interpretations of Statecharts admit refinement reified as both hierarchical or parallel composition (e.g. see Argos [9]), here, as previously [13], we focus only on hierarchical refinement, the form that Event-B natively admits. Here we define hierarchical composition to mean nesting new transition systems inside previously pure states, and parallel composition to be the combination in one machine of formerly separate transition systems. 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 semantics adopted here adheres closely to UML Statecharts [3] and is implemented in iUML-B. Models described in Statecharts are expressed in SCXML and translated into Event-B logic which uses the *Rodin platform* (Rodin) [2] for machine proofs. With suitable restrictions, Statecharts already provide a sound, intuitive, visual metaphor for refinement. Outfitted with a formal semantics, this work borrows from well-used Statechart practices in digital design. We previously reported [11] our early attempts to relate Statecharts to Event-B. At that stage (and similarly in [15]) we suggested the necessary extensions and basic mechanism of translation but, through restrictions and abstraction, avoided the more challenging problem of refinement with run to completion semantics. The goal of the present work is to provide usable, well-founded tools that are familiar to designers of high-consequence systems and yet provide the currently lacking formal guarantees needed to ensure safety, security, and reliability.

The rest of the paper is structured as follows. Section 2 provides background information on SCXML, Event-B, and iUML-B. Section 3 presents our running example. Section 4 discusses the various challenges for introducing a refinement notion into SCXML and demonstrates our approach. Section 5 shows our extensions to SCXML which are necessary for reasoning about properties of SCXML models. In Section 6, we illustrate our translation of SCXML models into Event-

```
while running:
  while run2completion = false
    if untriggered_enabled
      execute(untriggered())
    elseif IQ /= {}
      execute(internal(IQ.dequeue))
    else
      run2completion = true
    endif
  endwhile
  if EQ /= {}
    execute(EQ.dequeue)
    run2completion = false
  endif
endwhile
```

Listing 1: Pseudocode for 'run to completion'

B using the example introduced in Section 3. Section 7 shows how properties of the SCXML models can be specified as invariants and verified in Event-B. We summarise our contribution and conclude in Section 8.

2 Background

2.1 SCXML

SCXML is a modelling language based on Harel Statecharts 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 Statechart 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 representation of the run to completion semantics as defined within the latest W3C recommendation document [17]. Here IQ and EQ are the triggers present in the internal and external queues respectively.

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* $\mathsf{I}(\mathsf{v})$ 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, $\mathsf{G}(\mathsf{t}, \mathsf{v})$ is the guard of the event, and $\mathsf{S}(\mathsf{t}, \mathsf{v})$ is the action of the event.

$$\textbf{any}\ \mathsf{t}\ \textbf{where}\ \mathsf{G}(\mathsf{t}, \mathsf{v})\ \textbf{then}\ \mathsf{S}(\mathsf{t}, \mathsf{v})\ \textbf{end}$$

The action of an event comprises of one or more assignments, each of them has one of the following forms: (1) $\mathsf{v} := \mathsf{E}(\mathsf{t}, \mathsf{v})$, (2) $\mathsf{v} :\in \mathsf{E}(\mathsf{t}, \mathsf{v})$, and (3) $\mathsf{v} :| \mathsf{P}(\mathsf{t}, \mathsf{v})$. Assignments of form (1) are deterministic, assign the value of expression $\mathsf{E}(\mathsf{t}, \mathsf{v})$ to v. Assignments of forms (2) and (3) are non-deterministic. (2) assigns any value from the set $\mathsf{E}(\mathsf{t},\mathsf{v})$ to v, while (3) assigns any value satisfied predicate $\mathsf{P}(\mathsf{t},\mathsf{v})$ to v. A machine in Event-B corresponds to a transition system where *variables* represent the states and *events* specify the transitions. Note that invariants $\mathsf{I}(\mathsf{v})$ 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.

Machines can be refined by adding more details. Refinement can be done by extending the machine to include additional variables (*superposition refinement*) representing new features of the system, or to replace some (abstract) variables by new (concrete) variables (*data refinement*). More information about Event-B can be found in [8]. Event-B is supported by 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.

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. State-machines are typically refined by adding nested state-machines to states. Figure 1 shows an example of a simple state-machine with two states.

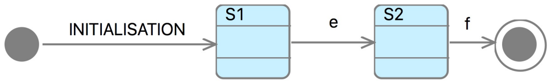


Fig. 1: An example iUML-B state-machine

```
variables S1 S2
invariants
  TRUE ∈ {S1, S2} ⇒ partition({TRUE}, {S1}∩{TRUE}, {S2}∩{TRUE})
events
  INITIALISATION: begin S1, S2 := TRUE, FALSE end
  e: when S1 = TRUE then S1, S2 := FALSE, TRUE end
  f: when S2 = TRUE then S2 := FALSE end
end
```

Listing 2: Translation of the state-machine in Fig. 1

Each state is encoded as a boolean variable and the current state is indicated by one of the boolean variables being set to TRUE. An invariant ensures that only one state is set to TRUE at a time. Events change the values of state variables to move the TRUE value according to the transitions in the state-machine. The Event-B translation[4] of the state-machine in Figure 1 can be seen in Listing 2.

3 Intrusion Detection System

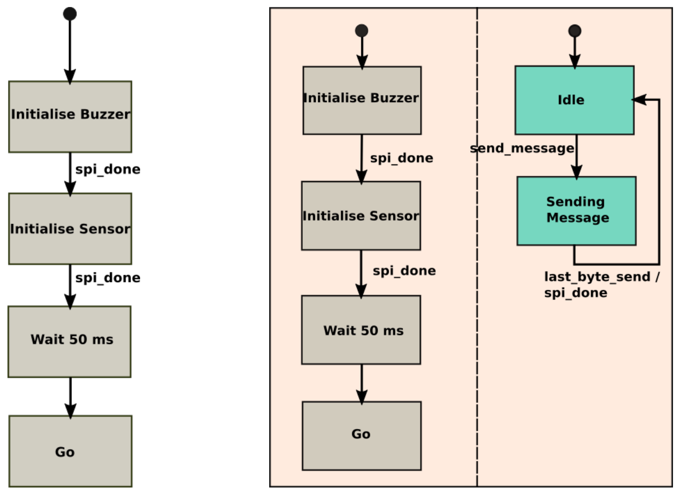
An *Intrusion Detection System* (IDS) is used to illustrate the use of refinement in Statecharts and how it is supported by Event-B verification tools. The IDS is designed using an *Application-Specific Integrated Circuit* (ASIC) which connects to a buzzer and a sensor over a *Serial Peripheral Interface* (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 initialise the sensor and the buzzer. After waiting for 50 milliseconds the ASIC enters its main routine, which makes the buzzer respond to the sensor. In the early design phase the Statechart model of this system may be limited to the ASIC that captures the initialisation of the peripherals and the 50 ms wait. In the interest of simplicity, we elide all details of the main routine.

A Statechart model of this system is shown in Fig. 2a. The ASIC starts by initialising the buzzer, this involves sending a message over the SPI bus. These messages constitute an implementation detail that we elide at this abstraction level. Once the message is sent (which will be indicated by some event saying that the SPI system is done), the ASIC moves on to initialise the sensor. After that the ASIC moves into a waiting state for 50 ms, and finally moves into the state which represents normal operation. At this abstraction the **spi_done** triggered,

[4] Here, partition$(\mathrm{S}, \mathrm{T1}, \mathrm{T2}, \ldots)$ means the set S is partitioned into disjoint (sub-)sets $T1, T2, \ldots$. that cover S

which signals completion by the SPI system, is an internal trigger that can be fired at any time.

In a subsequent level of refinement, shown in Fig. 2b, the designer adds a parallel state representing the SPI subsystem. The SPI subsystem is usually on an **Idle** state until the **send_message** trigger is raised, 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** trigger, allowing the other parallel state to continue, while SPI subsystem returns to idle. In the current refined model we have incorporated the implementation details for raising **spi_done** and introduced a new internal trigger **send_message**, which is nondeterministic at this point.



(a) ASIC component high level abstraction

(b) First refinement introducing the abstract model of the SPI subsystem.

Fig. 2: Statechart diagram for IDS including the abstract representation of the ASIC and SPI components.

The model can be further refined by incorporating more details on how the initialisation states, the wait state, and the SPI subsystem operate, including how they interact with each other. The Statechart diagram for this refinement level is in Fig. 3. The **Initialise Buzzer** state constructs the SPI message to send, then raises the **send_message** trigger, and then 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 initialised in a very similar manner to the buzzer. After both peripherals are initialised, the state machine goes into the **Wait 50 ms** state, where it increments a counter until it reaches some maximum, then exits.

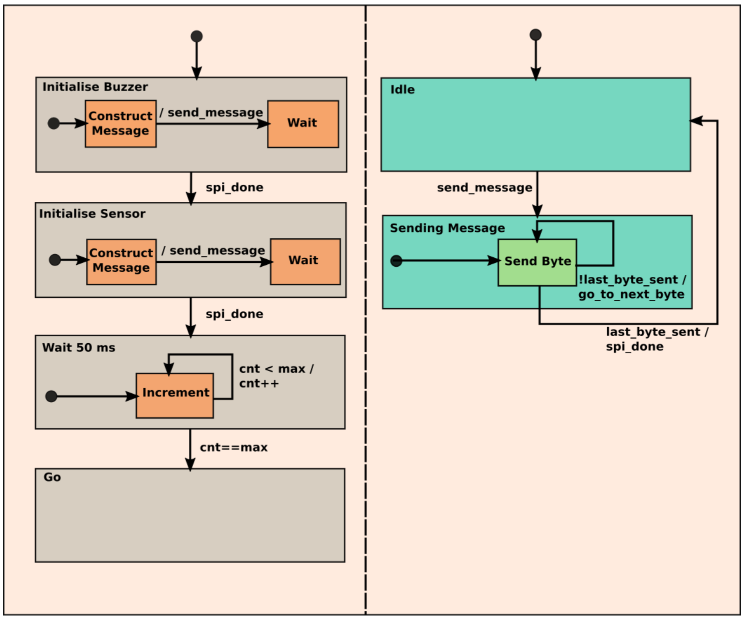


Fig. 3: Statechart diagram for IDS including implementation details for the messages send between the system components.

The system described must send messages to complete the initialisation of the buzzer and sensor, but once the main routine is reached (**Go** state) no more messages should be sent through the SPI bus. As a result, when the ASIC is in the **Go** state the SPI subsystem must be in the **Idle** state. This system property must be satisfied by the system model first refinement and any subsequent refinement representations of the system.

4 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 would be possible to utilise Event-B's data refinement to perform complex refinements at the Statechart level (for example replacing an abstract Statechart with a different one in the refined model), this would lead to complex proof obligations and is impractical when the SCXML model is a single

Statechart (rather than a chain of refined models). We prefer to use particular refinement idioms at the Statechart level that correspond to Event-B's superposition refinement and thus have simpler proof obligations. These refinement idioms are very natural from an engineering perspective (as illustrated by the running example). 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 state machines in the system.
– The firing conditions of a transition can be strengthened by being more specific about the (nested) source state,
– Nested Statecharts can be added in refinements.
– Ancillary data can be added and corresponding actions to alter it can be 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 is 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 may disable them and hence make the completion of enabled ones more easily achieved. This makes the guard for taking the next macro-step weaker breaking the notion of refinement.

While it is always possible to abstract away sufficiently to reach a common semantics (see [15] for example), in this work we wish to explore verification that considers 'run to completion' behaviour as closely as possible. To simulate the 'run to completion' semantics in Event-B, we initially adopted a scheduler approach where 'engine' events decide which user transitions should be fired based on their guards. Boolean flags are then used to enable these transitions which must then fire before the next step of the engine. The engine implements the operational semantics of Listing 1 by deciding when to use internal or external triggers. To allow for transition guards to be strengthened in later refinements the scheduling engine may continue without actually firing the transitions. However, this introduces many additional behaviours making simulation difficult. We abstract away from the SCXML rules about executing parallel transitions in document order and adopt non-deterministic firing order of transitions. A consequence is that if transitions that can fire in parallel assign to the same variable as each other the resulting value is non-deterministic.

Due to difficulties with non-deterministic firing of user transitions we developed an alternative approach where a separate event is generated for each combination of transitions that could possibly be fired together in the same step. For example, if T1 and T2 are transitions that could both become enabled at the same scheduler step, four events are needed to cater for the possible combinations: neither, T1, T2 and both (where the combined event is constructed from the conjunction of guards and parallel firing of actions). To allow for strengthening of the guards in refinement we omit the negation of guards leaving the choice of lesser combinations, including the empty one, non-deterministically available in case of future refinement. For example, T1 could fire alone even if T2 is enabled since we cannot add the negation of T2's guard to T1 unless we know that it will never be strengthened. This non-determinism in the model accurately reflects the abstract run to completion where we do not, yet know whether T2 will be enabled or not in future refinements. For future work we may consider extending SCXML with an attribute to indicate that a guard is *finalised* so that the negation can be used.

With this approach, since there is only ever a single event to be fired, the scheduler can be integrated with the events that represent the transition combinations, greatly simplifying the Event-B model. Instead of explicit events to progress and implement the scheduling engine, an abstract machine is provided with events that can be refined by the translation of the user's SCXML model. This has benefits both for animation which is easier to follow having less translation artefacts and for proof where the obligations are directly associated with particular transition combinations. Another benefit is that any parallel assignments to the same variable are rejected by the Event-B static checker. The disadvantage, of course, is that there could be a combinatorial explosion in the number of events generated. In practice though, this is unlikely since the number of parallel state machines is usually quite small. So far our examples have required few or no parallel transitions.

5 Extensions to SCXML

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 for 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 (see Listing 3, line 3).

```
1 ...
2 <state id="Wait50ms" iumlb:refinement="2">
3   <initial iumlb:refinement="2">
4     <transition cond="cnt=0" target="Increment"/>
5   </initial>
6   <iumlb:invariant name="check_cnt" iumlb:predicate="cnt &lt; max" iumlb:refinement="2"/>
7   <transition cond="" target="Go" iumlb:refinement="0" />
8   <state id="Increment" iumlb:refinement="2">
9     <transition cond="" event="tick" target="Increment">
10      <assign attr="cnt" expr="cnt+1" location="cnt"/>
11      <iumlb:guard name="stillCounting" iumlb:predicate="cnt &lt; 5"/>
12    </transition>
13    <transition cond="" target="WAITDone">
14      <iumlb:guard name="doneCounting" iumlb:predicate="cnt = 5"/>
15      </transition>
16  </state>
17  <final id="WAITDone" iumlb:refinement="2"/>
18  <datamodel>
19    <data expr="0" id="cnt" src="" iumlb:type="NAT "/>
20  </datamodel>
21 </state>
22 ...
```

Listing 3: **Wait50ms** state snippet of SCXML model representation illustrating the use of different SCXML modeling features, as well as, added syntax extensions

- **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 ancillary data and other state machines can be expressed (see Listing 3, line 6).
- **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 (Listing 3, line 14) as well as providing an element with attributes such as derived (for Event-B theorems), name and comment.
- **iumlb:predicate** - a string attribute used for the predicate of a guard or invariant (Listing 3, line 11).

Other attributes added for iUML-B elements are: **iumlb:name**, **iumlb:derived**, **iumlb:type**, **iumlb: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 Statechart (see Listing 3) 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.

6 SCXML Translation

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 an

SCXML meta-model provided by Sirius [4] which has good support for extensibility. The tooling for iUML-B and Event-B already contains EMF meta-models and provides a generic translator framework which has been specialised for the SCXML to iUML-B translation.

Each SCXML transition is translated into a corresponding iUML-B transition. Fig. 4 shows the iUML-B representation, first refinement level design, of the IDS described in Fig. 2b. In the translation from iUML-B to Event-B the 'run to completion' semantics is supported by Event-B through the construction of a basis that constitutes an abstract execution model, which all designed systems refine. The basis includes Event-B *context* and *machine*. The context, shown in Listing 4, defines the basis triggers as a partition of internal and external triggers, declared as SCXML_FutureInternalTrigger and SCXML_FutureExternalTrigger respectively. The abstract model of the IDS extends the basis context and declares a new axiom that defines the partitions of the SCXML_FutureInternalTrigger set, which would include the **spi_done** trigger and a new partition SCXML_FutureInternalTrigger0 to enable the introduction of additional internal triggers by subsequent refinements.

```
context
 basis_c // (generated for SCXML)
sets
 SCXML_TRIGGER // all possible triggers
constants
 SCXML_FutureInternalTrigger // all possible internal triggers
 SCXML_FutureExternalTrigger // all possible external triggers
axioms
 partition(SCXML_TRIGGER, SCXML_FutureInternalTrigger, SCXML_FutureExternalTrigger)
end
```

Listing 4: Abstract basis context

The basis machine, partially shown in Listing 5, declares the variables that correspond to the triggers present in the queue at any given time, as well as, the SCXML_uc flag, which signals when a run to completion macro-step has been completed and no un-triggered transitions are enabled. At the point of initialisation, both queues are empty and SCXML_uc is set to FALSE. The parametric SCXML_futureExternalTrigger event updates the external trigger queue with any newly raised trigger. To raise an external trigger, each external trigger in the model must extend this event. The SCXML_futureInternalTransitionSet basis event must be refined by any event in the model conditioned on an internal trigger. The guards of this event check for the completion of the previous macro-step. A similar behaviour is followed by SCXML_futureExternalTransitionSet event, if no internal triggers are in the queue.

To completely control the execution flow of the model, all un-triggered transitions must refine the SCXML_futureUntriggeredTransitionSet event. Any of the previously discussed events can raise a set of internal triggers, {i1,i2...}, by introducing a guard that defines {i1,i2...} $\subseteq$ SCXML_raisedTrigger parameter (not shown in listings). As shown in Listing 6 an event that corresponds to a trig-

```
1  machine
2   basis_m  // (generated for SCXML)
3  sees
4   basis_c
5  variables
6   SCXML_iq   // internal trigger queue
7   SCXML_eq   // external trigger queue
8   SCXML_uc   // run to completion flag
9  invariants
10  SCXML_iq ⊆ SCXML_FutureInternalTrigger  // internal trigger queue
11  SCXML_eq ⊆ SCXML_FutureExternalTrigger  // external trigger queue
12  SCXML_iq ∩ SCXML_eq= ∅        // queues are disjoint
13  SCXML_uc ∈ BOOL         // completion flag
14 events
15
16  INITIALISATION:
17  begin
18   SCXML_iq := ∅   //internal Q is initially empty
19   SCXML_eq := ∅   //external Q is initially empty
20   SCXML_uc := FALSE  //completion is initially FALSE
21  end
22
23  SCXML_futureExternalTrigger:
24  any SCXML_raisedTriggers where
25   SCXML_raisedTriggers ⊆ SCXML_FutureExternalTrigger
26  then
27   SCXML_eq := SCXML_eq ∪ SCXML_raisedTriggers
28  end
29
30  SCXML_futureInternalTransitionSet:
31  any SCXML_it SCXML_raisedTriggers where
32   SCXML_it ∈ SCXML_iq
33   SCXML_uc = TRUE
34   SCXML_raisedTriggers ⊆ SCXML_FutureInternalTrigger
35  then
36   SCXML_uc := FALSE
37   SCXML_iq := (SCXML_iq ∪ SCXML_raisedTriggers) \ {SCXML_it}
38  end
39
40  SCXML_futureUntriggeredTransitionSet:
41  any SCXML_raisedTriggers where
42   SCXML_uc = FALSE
43   SCXML_raisedTriggers ⊆ SCXML_FutureInternalTrigger
44  then
45   SCXML_uc := FALSE
46   SCXML_iq := SCXML_iq ∪ SCXML_raisedTriggers
47  end
48
49 end
```

Listing 5: Snippet of abstract basis machine

gered transition uses the provided parameter SCXML_it (SCXML_et for externally triggered transitions) to define which trigger enables the event (see line 8).

```
spi_done__InitialiseSensor_Wait50ms:
refines SCXML_futureInternalTransitionSet
any SCXML_it SCXML_raisedTriggers where
  SCXML_it ∈ SCXML_iq
  SCXML_uc = TRUE
  SCXML_raisedTriggers ⊆ SCXML_FutureInternalTrigger
  InitialiseSensor = TRUE
  SCXML_it = spi_done  //trigger for this transition
then
  SCXML_uc := FALSE
  SCXML_iq := (SCXML_iq ∪ SCXML_raisedTriggers) \ {SCXML_it}
  InitialiseSensor := FALSE
  Wait50ms := TRUE
end
```

Listing 6: Event-B event corresponding to internal triggered transition to **Wait50ms** state in refinement level 1 shown in Fig. 2a

7 Verification of Intrusion Detection System

One of our main goals is to express properties in SCXML intermediate refinements and prove them via translation to Event-B. In this section we illustrate how this can be done in the IDS example.

Properties about the synchronisation of parallel state-machines (such as $\mathsf{ASIC} = \mathsf{Go} \Rightarrow \mathsf{SPI} = \mathsf{IDLE}$) can be difficult to verify for all scenarios via simulation in SCXML. Proof of such properties is a major benefit of translating into Event-B. Furthermore, in order to benefit from the abstraction provided by Event-B, we would like to prove such things at abstract levels before the complication of further details are introduced. Typically these further details concern the raising of internal triggers that contribute to the synchronisation we wish to verify. Therefore additional constraints, that are an abstraction of the missing details, are needed about triggers in order to perform the proof.

Fig. 4 is the generated iUML-B showing state invariants (textual properties with a star icon inside states) to be verified. Note that the invariants are added to the SCXML model but are easier to visualise in the iUML-B with the current tooling. The main aim is to show the property Idle=TRUE holds in state Go. This is true because after sending the message while in InitialiseSensor, no other messages are triggered by the ASIC, so the SPI subsystem stays in the Idle state indefinitely. To enable the provers to discharge the proof obligation we work back along the ASIC's sequence of states. That is, $\mathsf{Idle} = \mathsf{TRUE}$ is maintained in state Go if it holds in state Wait50ms and no send_message triggers are raised by the entry transition Wait50ms_Go nor once the ASIC subsystem is in state Go. To ensure this we add a guard $\mathsf{send\_message} \notin \mathsf{SCXML\_raisedTriggers}$ to Wait50ms_Go to prevent any future refinement from raising the trigger send_message. (Currently, this is added verbatim but we envision a 'doesn't raise' notation to avoid the user having to reference the translation artefact, SCXML_raisedTriggers). We also need to prevent any future transitions from raising this trigger in the state Go. To automate this for all abstract 'future' events, they could be automati-

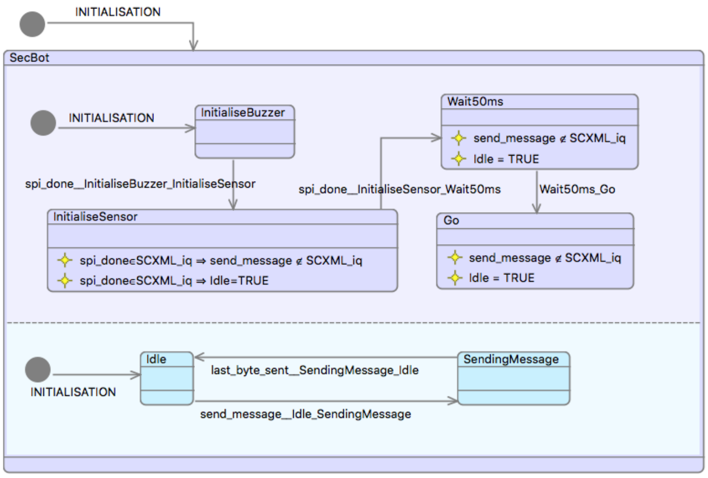


Fig. 4: State invariants to be verified at refinement level 1.

cally generated and added to satisfy all user invariants concerning the raising of internal triggers regardless of whether they are violated in future levels. For example, the guard $\mathsf{Go} = \mathsf{TRUE} \Rightarrow \mathsf{send\_message} \notin \mathsf{SCXML\_raisedTriggers}$ needs to be automatically added to the three 'basis' events SCXML_future..TransitionSet to prove they do not break the property being verified. If it is not obeyed by future transitions, guard strengthening proof obligations will fail, making it obvious where the problems lie. As indicated above, we now need to prove by similar means that Idle=TRUE holds in state Wait50ms. In this case, however, we can only say that Idle=TRUE in state InitialiseSensor after the SPI-system finishes sending the message and raises the trigger, spi_done. Hence the state invariant for InitialiseSensor becomes $\mathsf{spi\_done} \in \mathsf{SCXML\_iq} \Rightarrow \mathsf{Idle}{=}\mathsf{TRUE}$. In order to prove this we again need a corresponding state invariant about send_message and need to make sure that the SPI system will never raise send_message. We also ensure it does not raise spi_done until it is finished. With these invariants and additional guards the Rodin automatic provers are able to prove all proof obligations and hence verify that the SPI system remains in Idle after servicing the 'Initialise Sensor' message.

In order to prove these properties are true at an abstract level the prover forces us to add constraints on the behaviour to be added in later refinements. For example to complete the proof we needed to add a guard to specify that a transition would not raise a particular trigger in any future refinement. If these constraints are broken by later refinements, proof obligations about guard strengthening will be unprovable. The abstract guards should be removed at later refinements when the details have been specified. To do this we could introduce ranges in our refinement attributes.

8 Conclusion

We have shown how a slightly extended and annotated Statechart, with a typical 'run to completion' semantic, can be translated into the Event-B notation for verification of synchronisation properties using the Event-B theorem proving tools. Furthermore, borrowing from the refinement concepts of Event-B, we introduce a notion of refinement to Statecharts and demonstrate how the proof of a property at an abstract level, helps formulate constraints that must apply (and will be verified to do so) in further refinements.

Refinement of UML Statecharts has been studied previously in [6,10,16]. In [6], the authors propose a "purely additive" refinement process where no elements (e.g. events, guards, etc.) of the original model can be removed and the "external" behaviour of the model is therefore preserved. This refinement process is similar to Event-B "superposition" refinement which we use in our translation. In [10], the authors consider a coalgebraic description of UML Statecharts, and define an equivalence relationship and a behavioural refinement notion between Statecharts. In [16], the authors define a structured operational semantics of Statecharts based on label transition systems. Behaviour refinements are then constructed based on this semantics. The authors prove that a "safe-extension" of UML Statecharts is a correct behavioural refinement. In our paper, we focus on the run-to-completion semantics of Statecharts, whereas none of the above work deals with it explicitly. Furthermore, the refinement process supported in [6,10] is based on refinement patterns (called refinement rules/laws), whereas we rely on the more general theory of refinement, given by the proof obligations of Event-B, for proving the refinement relationship between Statecharts.

In future work we will continue to experiment with different examples to explore the alternative translation strategies in more detail. In particular, further work on refinement of the micro/macro-step and whether correspondence of macro-steps can be relaxed; whether more complex refinement techniques could be supported (for example, using ranges in refinement annotations) would be useful; supporting/comparing alternative variations of semantics (by generating a different basis/scheduler for the translation).

For our interpretation of Statecharts in iUML-B, we used the 'run-to-completion' semantics of Statecharts. In particular, we have carefully designed our translated model such that the semantics is captured as a generic abstract model, which is subsequently refined by the translation of the SCXML model. An advantage of this approach is that we can easily adapt the basis model with other alternative semantics [5] without changing the translation of the SCXML model.

While Statecharts interpreted in iUML-B provide a way to incorporate refinement in an intuitive way, reversing this to *discover* refinements holds promise. Checking a particular Statechart model for hierarchical structures that happen to follow the refinement proof obligations suggests an automatic way to accomplish abstract interpretation on an existing model. Such discovered abstraction/refinement relationships might improve the scalability of more complex Statechart models "for free".

All data supporting this study are openly available from the University of Southampton repository at https://tinyurl.com/SCXML-REF **(DOI FOR FINAL)**

Acknowledgment: The authors would like to thank Jason Michnovicz for developing the IDS example used throughout the manuscript.

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. Alexandre David, M. Oliver Möller, and Wang Yi. Formal verification of UML statecharts with real-time extensions. In Ralf-Detlef Kutsche and Herbert Weber, editors, *Fundamental Approaches to Software Engineering*, pages 218–232, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.
4. Eclipse Foundation. Sirius project website. https://eclipse.org/sirius/overview.html, March 2016.
5. Rik Eshuis. Reconciling statechart semantics. *Science of Computer Programming*, 74(3):65–99, January 2009.
6. Conner Hansen, Eugene Syriani, and Levi Lucio. Towards controlling refinements of statecharts. *CoRR*, abs/1503.07266, 2015.
7. David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, June 1987.
8. Thai Son Hoang. An introduction to the Event-B modelling method. In *Industrial Deployment of System Engineering Methods*, pages 211–236. Springer-Verlag, 2013.
9. F. Maraninchi. The Argos language: Graphical representation of automata and description of reactive systems. In *In IEEE Workshop on Visual Languages*, 1991.
10. Sun Meng, Zhang Naixiao, and L. S. Barbosa. On semantics and refinement of uml statecharts: a coalgebraic view. In *Proceedings of the Second International Conference on Software Engineering and Formal Methods, 2004. SEFM 2004.*, pages 164–173, Sept 2004.
11. Karla Morris and Colin Snook. Reconciling SCXML statechart representations and Event-B lower level semantics. In *HCCV - Workshop on High-Consequence Control Verification*, 2016.
12. James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The (2nd Edition).* Pearson Higher Education, 2004.
13. C. Snook. iUML-B statemachines. In *Proceedings of the Rodin Workshop 2014*, Toulouse, France, 2014. `http://eprints.soton.ac.uk/365301/`.
14. Colin Snook and Michael Butler. UML-B: Formal modeling and design aided by uml. *ACM Trans. Softw. Eng. Methodol.*, 15(1):92–122, January 2006.
15. Colin Snook, Vitaly Savicks, and Michael Butler. Verification of UML models by translation to UML-B. In Bernhard K. Aichernig, Frank S. de Boer, and Marcello M. Bonsangue, editors, *Formal Methods for Components and Objects*, pages 251–266, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
16. Nora Szasz and Pedro Vilanova. Behavioral refinements of UML-Statecharts. Technical Report RT 10-13, Universidad de la Rep'ublica, Montevideo, Uruguay, 2010.
17. W3C. State chart XML SCXML: State machine notation for control abstraction. http://www.w3.org/TR/scxml/, September 2015.