Refinement of Statecharts with Run-to-Completion Semantics

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

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

Abstract. Statechart modelling notations, with so-called 'run to completion' semantics and simulation tools for validation, are popular with engineers for designing systems. 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 system 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. In this paper, we introduce a notion of refinement, similar to that of Event-B, into a 'run to completion' Statechart modelling notation, and leverage Event-B's tool support for proof. We describe the pitfalls in translating 'run to completion' models into Event-B refinements and suggest a solution. 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.

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 safety-critical and other high-consequence 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, the need for instantiation and state space explosion can make testing of safety, security and reliability properties intractable. Therefore, such properties 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. The binding is facilitated by translating to iUML-B [18,19,20], a diagrammatic modelling notation for Event-B. Hierarchical encapsulation maps well onto Statecharts in a similar way to nested state-machines in iUML-B. Binding UML Statecharts [17] to iUML-B 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) [22] 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 [12]), here, as previously [18], we focus 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 [16] our early attempts to relate Statecharts to Event-B. At that stage (and similarly in [20]) we suggested the necessary extensions and basic mechanism of translation but 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 safety-critical systems with the formal guarantees needed to ensure safety and reliability. The motivation of the work is entirely driven by the industrial partner, who feels that the current sematics for Statecharts is insufficient for formal verification.

The Event-B modelling method provides the logic and refinement theory required to formally analyse a system model. The open-source Rodin provides support for Event-B including automatic theorem provers and model checking capabilities. iUML-B augments the Event-B language with a graphical interface including state-machines.

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. In Section 5, we illustrate our translation of SCXML models into Event-B using the example introduced in Section 3. Section 6 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 7.

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 [22]. Here IQ and EQ are the triggers present in the internal and external queues respectively.

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

Listing 1: Pseudocode for 'run to completion'

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

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*.

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 form: **any** t **where** $\mathsf{G}(\mathsf{t}, \mathsf{v})$ **then** $\mathsf{S}(\mathsf{t}, \mathsf{v})$ **end** 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.

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 relate to 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. 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.

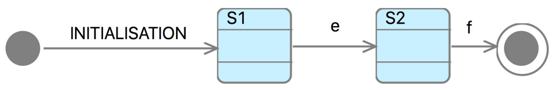


Fig. 1: An example iUML-B state-machine

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. iUML-B also provides the option of an alternative translation with a single state variable ranging over an enumerated type of states, however, the boolean representation of each state is more natural for a user to reference in SCXML guards and actions.

While the iUML-B translation deals with the basic data formalisation of state-machines it differs significantly from the aims of the work presented here. iUML-B adopts Event-B's simple guarded action semantics and does not have a concept of triggers and run-to-completion. Here we make use of iUML-B's state-machine translation but provide a completely different semantic by generating a behaviour into the underlying Event-B events that are linked to the generated iUML-B transitions.

```
 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
```

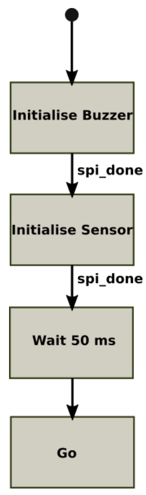
Listing 2: Translation of the state-machine in Fig. 1

3 Intrusion Detection System

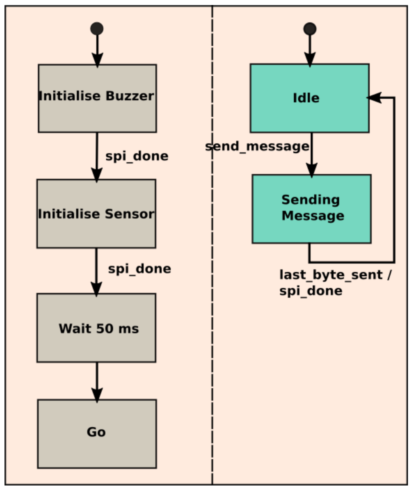
An *Intrusion Detection System* (IDS) is used to illustrate the use of refinement in Statecharts and how it is supported by Event-B verification tools. The IDS is designed using an *Application-Specific Integrated Circuit* (ASIC) which connects to a buzzer and a sensor over a *Serial Peripheral Interface* (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**

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



(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.

trigger, which indicates that the SPI system has finished, is an internal trigger that can be fired at any time.

In a subsequent level of refinement, shown in Fig. 2b, the designer uses superposition refinement to add a parallel state representing the SPI subsystem. The SPI subsystem is usually in 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 the 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 non-deterministic at this point.

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

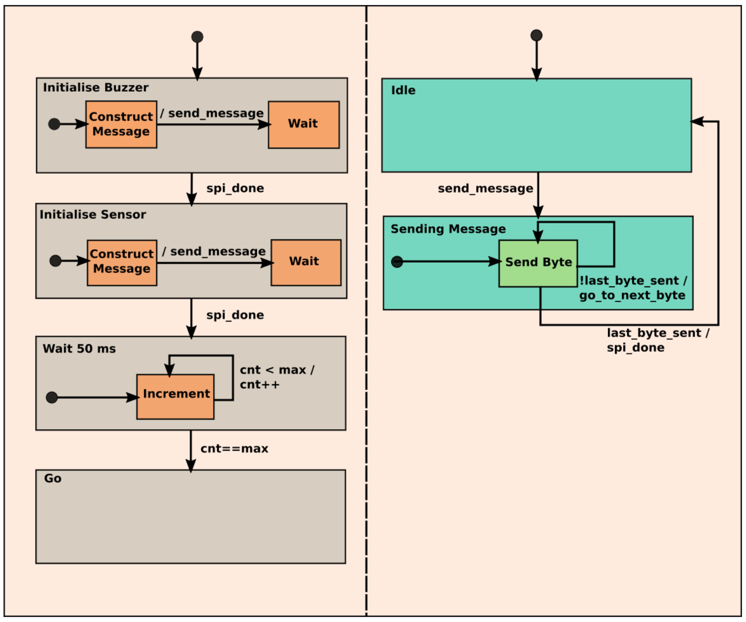


Fig. 3: Statechart diagram for IDS including implementation details for the messages sent between the system components.

state machine goes into the **Wait 50 ms** state, where it increments a counter until it reaches some maximum, then exits.

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, a desirable safety property is that *when the ASIC is in the* ***Go*** *state the SPI subsystem must be in the* ***Idle*** *state.* This safety property should hold from the first refinement and be preserved in all future refinements.

4 Design Rationale

We consider the kinds of things we would like to do in SCXML 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. We use particular refinement idioms at the Statechart level that correspond to Event-B's superposition refinement and thus have simple 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.
– Actions that modify ancillary data 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.
– External triggers represent inputs to the model. If no restrictions are imposed on the inputs then the events that raise external trigger are always unguarded and cannot be refined.
– Invariants can be added to states to specify properties that hold while in that state.

While it would be possible to utilise Event-B's data refinement to perform more substantial Statechart refinements (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).

Adherence to Event-B refinement means that refined transitions (hence micro- and macro-steps) should preserve the abstract state and new ones should not alter the abstract state. With this approach, there is an inherent difficulty in refining 'run to completion' semantics where every enabled micro-step must be 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 [20] 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 were then used to enable these transitions which may fire before the next step of the engine. The engine implemented 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 (hence achieving completion earlier) the scheduling engine was allowed to continue without actually firing the transitions. However, this non-deterministic completion introduced many additional behaviours making simulation difficult.

Due to these difficulties with non-deterministic completion 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. The non-determinism is useful to allow abstractions which facilitate verification proofs but must be removed in refinements to reach a design suitable for implementation. In future work we intend to add an attribute *finalised* to indicate that no further guard strengthening refinements will be made to a transition, removing non-determinism throughout the refinement chain.

Since there is only ever a single event to be fired in a particular micro-step, 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 into events that represent combinations of transitions that can fire in the same micro-step. Each refinement produces a new set of events representing the (possibly extended) transition combinations that may occur at that level of refinement. This has benefits both for simulation (i.e. execution of the Statechart for validation) 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, to be fired in parallel, transitions must have the same trigger and be located in parallel Statecharts. A high number of events is also not necessarily a bad thing since they are automatically generated and the main purpose of the Event-B model is for proof which could be simplified by replacing some of the unnecessary sequential steps of the model by a choice. If the number of combinations is excessive it may indicate poor modelling style which can be reduced by introducing more internal triggers. So far our examples have required few or no parallel transitions.

The following syntax extensions are added to SCXML models to support refinement and invariant verification.

- **refinement** - an integer attribute representing the refinement level at which the parent element should be introduced,
- **invariant** - an invariant property (such as synchronisation of state with ancillary data and other state machines) that holds while in the parent state,
- **guard** - a guard condition of the parent transition (allowing transition conditions to be added at particular refinement levels).

```
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 3: Abstract basis context

5 SCXML Translation

The translation from SCXML to Event-B is based on an abstract 'basis' that models the 'run to completion' semantics. This basis consists of an Event-B *context* and *machine* that are the same for all input models and are refined by the specific output of the translation. The basis context, Listing 3, introduces a given set of all possible triggers that is partitioned into internal and external ones, some of which will be introduced in future refinements. Refinements partition these trigger sets further to introduce concrete triggers, leaving a new abstract set to represent the remaining triggers yet to be introduced. For example, the IDS model introduces a specific internal trigger, **spi_done**, by partitioning SCXML_FutureInternalTrigger into the singleton **{spi_done}** and a new set, SCXML_FutureInternalTrigger0, representing the remainder.

The basis machine, part of which is shown in Listing 4, declares variables that correspond to the triggers present in the queue at any given time, and a flag, SCXML_uc, that signals when a run to completion macro-step has been completed (no un-triggered transitions are enabled). After initialisation, both trigger queues are empty and SCXML_uc is set to FALSE so that un-triggered transitions are dealt with. The basis machine provides events that describe the generic behaviour of models that follow the run to completion semantics in terms of altering the trigger queues and completion flag. Since new events introduced in a refinement cannot modify existing variables, all future events generated by translation of the specific SCXML model, will refine these abstract events. The abstract event, SCXML_futureExternalTrigger represents the raising of an external trigger. The abstract event, SCXML_futureInternalTransitionSet represents a combination of transitions that are triggered by an internal trigger. The guards of this event ensure prior completion of the previous macro-step. A similar event, SCXML_futureExternalTransitionSet (not shown) represents a combination of transitions that are triggered by an external trigger and has the additional guard that the internal trigger queue is empty. These two triggered transition events reset the completion flag to ensure that any un-triggered transitions that may have become enabled have a chance to fire next. The abstract event SCXML_futureUntriggeredTransitionSet represents a combination of transitions that are un-triggered and may only be fired when the completion flag is unset (FALSE). It leaves the completion flag unset in case further combinations

```
machine basis_m  sees basis_c // (generated for SCXML)
variables
 SCXML_iq   // internal trigger queue
 SCXML_eq   // external trigger queue
 SCXML_uc   // run to completion flag
invariants
 SCXML_iq ⊆ SCXML_FutureInternalTrigger  // internal trigger queue
 SCXML_eq ⊆ SCXML_FutureExternalTrigger  // external trigger queue
 SCXML_iq ∩ SCXML_eq= ∅         // queues are disjoint
 SCXML_uc ∈ BOOL          // completion flag
events

 INITIALISATION:
 begin
  SCXML_iq := ∅   //internal Q is initially empty
  SCXML_eq := ∅   //external Q is initially empty
  SCXML_uc := FALSE //completion is initially FALSE
 end

 SCXML_futureExternalTrigger:
 any SCXML_raisedTriggers where
  SCXML_raisedTriggers ⊆ SCXML_FutureExternalTrigger
 then
  SCXML_eq := SCXML_eq ∪ SCXML_raisedTriggers
 end

 SCXML_futureInternalTransitionSet:
 any SCXML_it SCXML_raisedTriggers where
  SCXML_it ∈ SCXML_iq
  SCXML_uc = TRUE
  SCXML_raisedTriggers ⊆ SCXML_FutureInternalTrigger
 then
  SCXML_uc := FALSE
  SCXML_iq := (SCXML_iq ∪ SCXML_raisedTriggers) \ {SCXML_it}
 end

 SCXML_futureUntriggeredTransitionSet:
 any SCXML_raisedTriggers where
  SCXML_uc = FALSE
  SCXML_raisedTriggers ⊆ SCXML_FutureInternalTrigger
 then
  SCXML_uc := FALSE
  SCXML_iq := SCXML_iq ∪ SCXML_raisedTriggers
 end

end
```

Listing 4: Abstract basis machine (part of)

of un-triggered transitions are enabled. All three of these transition events also allow for raising a non-deterministic set of internal triggers. A final abstract event, SCXML_completion, sets the completion flag (TRUE) if it is not already set. At this abstract basis level, this is non-deterministically fired since we do not yet have any detail of what needs to be completed.

The translation of a specific SCXML model comprises two stages as follows. Firstly, all possible combinations of transitions that can fire together are calculated and corresponding events are generated, at appropriate refinement levels, that refine the abstract basis events. If these transitions raise internal triggers, a guard, (e.g. $\{i1, i2, \ldots\} \subseteq$ SCXML_raisedTrigger, where i1, i2, … have been added

```
 1 spi_done__InitialiseSensor_Wait50ms:
 2 refines SCXML_futureInternalTransitionSet
 3 any SCXML_it SCXML_raisedTriggers where
 4 SCXML_it ∈ SCXML_iq
 5 SCXML_uc = TRUE
 6 SCXML_raisedTriggers ⊆ SCXML_FutureInternalTrigger
 7 InitialiseSensor = TRUE
 8 SCXML_it = spi_done  //trigger for this transition
 9 then
10 SCXML_uc := FALSE
11 SCXML_iq := (SCXML_iq ∪ SCXML_raisedTriggers) \ {SCXML_it}
12 InitialiseSensor := FALSE
13 Wait50ms := TRUE
14 end
```

Listing 5: Event-B event corresponding to internal triggered transition to **Wait50ms** state in refinement level 1 shown in Fig. 2a

to the internal triggers set), is introduced that defines the raised triggers parameter. The subset constraint leaves it open for more raised triggers to be added by later refinements. For triggered transition combinations, the trigger is specified in a guard (see line 8 of Listing 5) that provides a value for the trigger parameter.

Secondly, the SCXML state-chart is translated into a corresponding iUML-B state-machine whose transitions elaborate (i.e. add state change details to) the possible transition combination events that the transition may be involved in. A transition may fire in parallel with transitions of parallel nested state-machines that have the same (possibly null) trigger. Fig. 4 shows the generated iUML-B first refinement level corresponding to the IDS described in Fig. 2b. The main rules for the translation of SCXML features to iUML-B/Event-B can be summarized as follow:

Top level SCXML model: Generates a refinement chain of Event-B machines each containing an initialisation event and a iUML-B state-machine. The depth of the refinement chain is found by searching the SCXML for the maximum refinement annotation.

State: Generates a state in the iUML-B state-machine that has been produced from the container of the SCXML state. A refined state is also added in all of the refinements of the parent iUML-B state-machine. E.g. Fig. 2b, **Initialise Buzzer** → Fig. 4, InitialiseBuzzer.

State invariant: Generates an invariant in the iUML-B state corresponding to the SCXML state that contains the invariant. Added only at the refinement level defined in the invariant (defaults to first level at which containing iUML-B state is introduced). E.g. Fig. 4, Idle = TRUE is generated from an invariant attached (not shown) to the state **Go** of Fig. 2b.

Parallel Region: Generates an iUML-B state-machine in the state corresponding to the owner of the parallel region. The nested iUML-B state-machine is added starting from the refinement level that is annotated on the parallel region and continuing throughout subsequent refinements. E.g. Fig. 2b, right-hand region → Fig. 4, lower nested state-machine.

Initial: Generates an iUML-B initial state, and a transition from it to the iUML-B state indicated in the SCXML initial attribute. The iUML-B initial state and iUML-B transition are added at all refinement levels. The iUML-B transitions are set to elaborate the Event-B INITIALISATION event for that refinement level. E.g. Fig. 2b, initial state and transition in right-hand region $\rightarrow$ Fig. 4, initial state and transition in lower nested state-machine.

Final: Generates an iUML-B state with a transition to a final state in the state-machine that has been generated from the containing SCXML state. The transition elaborates the same events that are linked to the transitions that exit the parent iUML-B state. The iUML-B state, final state and transition are also added as refined elements to all of the refinements of the parent iUML-B state-machine. (Not used in our example).

Transition: Generates an iUML-B transition in the state-machine that has been generated from the containing SCXML state. The iUML-B transitions source and target are those that have been generated from the SCXML transitions source and target states. The transition elaborates generated Event-B events according to the rules given in Section 5. The iUML-B transition and elaborated Event-B events are also added as corresponding refined elements in all of the refinements of the parent iUML-B state-machine. E.g. Fig. 2b, **send_message** $\rightarrow$ Fig. 4, send_message__Idle_SendingMessage.

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 iUML-B state-machine is subsequently translated into Event-B using the standard iUML-B translation [18] which provides variables to model the current state and guards and actions to model the state changes that transitions perform.

6 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 $\mathsf{Go} = \mathsf{TRUE} \Rightarrow \mathsf{Idle} = \mathsf{TRUE}$) 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

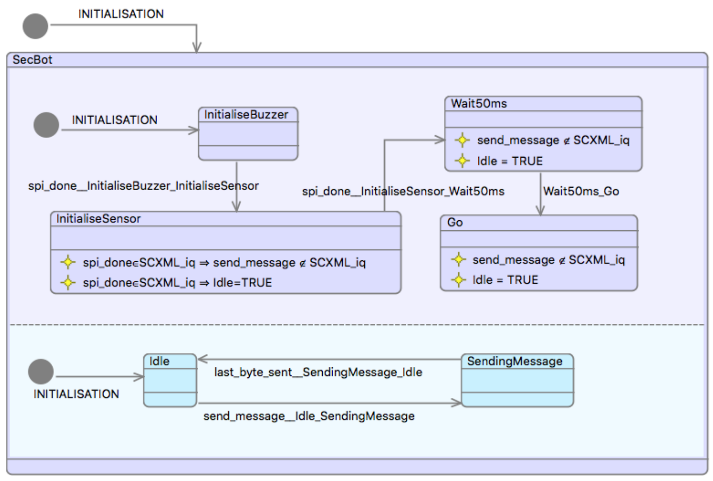

INITIALISATION
SecBot
INITIALISATION
InitialiseBuzzer
Wait50ms
send_message ∉ SCXML_iq
Idle = TRUE
spi_done__InitialiseBuzzer_InitialiseSensor
spi_done__InitialiseSensor_Wait50ms
Wait50ms_Go
InitialiseSensor
spi_done∈SCXML_iq ⇒ send_message ∉ SCXML_iq
spi_done∈SCXML_iq ⇒ Idle=TRUE
Go
send_message ∉ SCXML_iq
Idle = TRUE
Idle
last_byte_sent__SendingMessage_Idle
SendingMessage
INITIALISATION
send_message__Idle_SendingMessage

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

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, Idle = 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 send_message $\notin$ 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 automatically 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 Go = TRUE $\Rightarrow$ send_message $\notin$ SCXML_raisedTriggers needs to be automatically added to the three 'basis' events, SCXML_futureUntriggeredTransitionSet, SCXML_futureInternalTransitionSet and SCXML_futureExternalTransitionSet 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 spi_done$\in$SCXML_iq $\Rightarrow$ Idle=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 properties at an abstract level we constrain the behaviour to be added in later refinements. For example, we needed to add a guard to specify that a transition does not raise a particular trigger in any future refinement. The abstract constraints should not appear in later refinements when the details have been finalised. To do this we could introduce ranges into our refinement attributes.

7 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.

Related Work. Refinement of UML Statecharts has been studied previously in [14,21,15,6,11]. In [14], the authors consider a coalgebraic description of UML Statecharts, and define an equivalence relationship and a behavioural refinement notion between Statecharts. In [21], 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 [15,11], formal refinement rules are developed for SysML, including Statecharts, based on the corresponding process refinement rules of the Compass Modelling Language. *The issue of run to completion with respect to refinement is not considered explicitly nor shown in any examples.* 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 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 [14,15,6,11] 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.

Future Work. 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.

We will also demonstrate the scalability of the translation on more realistic industrial examples. The Haemodialysis Machine case study [13] from the ABZ 2016 conference would make a good test case since its highly sequential processes are natural for a state-chart representation and results can be compared with existing iUML-B solutions [10]. The ERTMS Hybrid Level 3 case study [9] from the ABZ 2018 conference is also an industrial example which would test the method. This case study would require lifting of the output models to a generalised set of instances using a model composition technique that we have been developing for this purpose.

All data supporting this study are openly available from the University of Southampton repository at `https://doi.org/10.5258/SOTON/D0693`

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. Thai Son Hoang, Michael Butler, and Klaus Reichl. The hybrid ertms/etcs level 3 case study. In Michael Butler, Alexander Raschke, Thai Son Hoang, and Klaus Reichl, editors, *Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pages 251–261, Cham, 2018. Springer International Publishing.
10. Thai Son Hoang, Colin Snook, Lukas Ladenberger, and Michael Butler. Validating the requirements and design of a hemodialysis machine using iuml-b, bmotion studio, and co-simulation. In *Proceedings of the 5th International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z - Volume 9675*, ABZ 2016, pages 360–375, Berlin, Heidelberg, 2016. Springer-Verlag.
11. Lucas Lima, Alvaro Miyazawa, Ana Cavalcanti, Márcio Cornélio, Juliano Iyoda, Augusto Sampaio, Ralph Hains, Adrian Larkham, and Vaughan Lewis. An integrated semantics for reasoning about SysML design models using refinement. *Software & Systems Modeling*, 16(3):875–902, Jul 2017.
12. F. Maraninchi. The Argos language: Graphical representation of automata and description of reactive systems. In *In IEEE Workshop on Visual Languages*, 1991.
13. Atif Mashkoor. The hemodialysis machine case study. In *Proceedings of the 5th International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z - Volume 9675*, ABZ 2016, pages 329–343, Berlin, Heidelberg, 2016. Springer-Verlag.
14. 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.
15. Alvaro Miyazawa and Ana Cavalcanti. Formal refinement in SysML. In Elvira Albert and Emil Sekerinski, editors, *Integrated Formal Methods*, pages 155–170. Springer International Publishing, 2014.
16. Karla Morris and Colin Snook. Reconciling SCXML statechart representations and Event-B lower level semantics. In *HCCV - Workshop on High-Consequence Control Verification*, 2016.
17. James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004.
18. C. Snook. iUML-B statemachines. In *Proceedings of the Rodin Workshop 2014*, Toulouse, France, 2014. `http://eprints.soton.ac.uk/365301/`.
19. 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.
20. 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.
21. Nora Szasz and Pedro Vilanova. Behavioral refinements of UML-Statecharts. Technical Report RT 10-13, Universidad de la Rep'ublica, Montevideo, Uruguay, 2010.
22. W3C. State chart XML SCXML: State machine notation for control abstraction. http://www.w3.org/TR/scxml/, September 2015.