SCXML semantics with refinement in Event-B

# SecBot Example

This attempt is based on a Sec Bot example State chart model, which is introduced in 3 levels. The example adds state chart details as well as introducing internal trigger events before defining where they are raised.

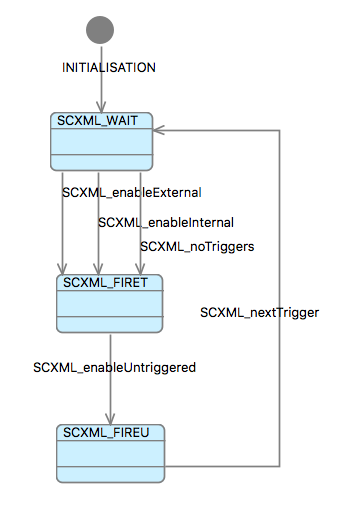
## SCXML semantics

The SCXML semantics are described operationally here:-

Our (abstract) interpretation is as follows

1. Initialise
2. Take one of the following options:
   1. If the external queue is not empty, take one event from the internal queue and fire the set of transitions that are enabled by it at that time. I.e. they must be enabled at the time it checks, not subsequently after firing one or more of the set of transitions.
   2. the internal queue is empty but
   3. If the internal and external queues are both empty proceed to step 4)
3. Fire the set of un-triggered transitions that are enabled after step 1). Again they must be enabled at the time it checks, not subsequently.
4. repeat from 1)

## SCXML engine

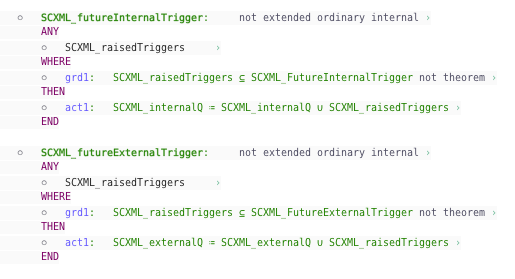
A *basis* machine and context are provided to define the generic elements of the SCXML engine. Specific translated SCXML models start by refining/extending this basis. The basis provides a cyclic engine representing an abstraction of the semantics of SCXML. There are two trigger queues; internal and external. (Currently these are modelled as sets. It would be more accurate to make them queues).

Initially the engine waits for triggers. When a trigger is present, internal triggers have precedence over external ones. If no triggers are present, the engine proceeds to the next step of enabling untriggered transitions. A trigger is consumed and used to evaluate the enabledness of the transitions according to the current state of the SCXML application state chart and any other conditions involving system data. A Boolean flag is constructed for each transition recording its enabledness. This flag is used as the guard for the Event-B event that represents the corresponding SCXML transition. The engine then waits for all the transitions it enabled to fire. The transitions set new state and data as well as resetting their enabled flag. They may also raise new internal triggers by adding them to the internal queue.

When all flags are reset to FALSE, the engine evaluates the enabledness of the transitions that are not triggered. This is done in a similar way using Boolean flags. When all un-triggered transitions have fired the engine goes back to the start and consumes another trigger from the internal queue if there is one, or from the external queue if not. If there are no triggers available, the engine repeats this step of evaluating the enabledness of the transitions that are not triggered. This allows the system to progress through a sequence of untriggered behaviour while allowing triggered behaviour to interrupt.

In the basis machine, since the specific SCXML model and transitions are not present, the flags and guards evaluation is missing. The basis provides a starting point so that these engine events can be extended with the specific model information described above.

## Future Triggers

 The basis also provides a mechanism for introducing new triggers. Since these triggers require changes to the trigger queues, which are present from the abstract basis, they cannot be handled by ‘new’ events. Abstract ‘futureInternalTrigger’ and ‘futureExternalTrigger’ events, which add an unspecified set of triggers to the relevant queue, are provided for future transitions to refine. A set of ‘FutureInternalTrigger’ and ‘FutureExternalTrigger’ are provided as an abstraction of the triggers that may be introduced in the future.

## 

New triggers are introduced by partitioning this set, leaving a residual for further future triggers.



At each refinement step we further partition the original set of triggers by introducing a smaller residual set.



Note that we could have partitioned the previous residual set but this does not result in such a clean enumeration when animating with ProB.

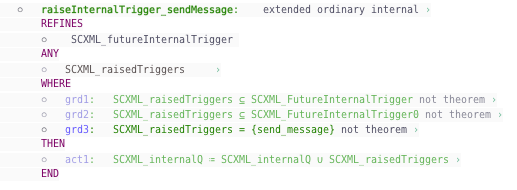
## Refinement of triggers

Note that internal triggers must be introduced as such. They cannot be introduced as external triggers and then later refined as internal ones. This is because the categorisation affects the priority and ordering of their consumption and hence the trace of events which is the definition of refinement. I.e. it is inherently not a refinement to replace an external trigger with an internal trigger.

## Non-deterministically raised internal triggers

To allow for future refinement, we might wish to introduce and use an internal trigger but not yet describe how it is raised. To do this we introduce an unguarded event that refines *futureInternalTrigger*. This means that it could be raised at any time to enter the event in the queue. Later this can be refined by a transition event in order to specify how the internal triggers are raised. If several triggers will be raised together by a single transition, they must be introduced in a single such non-deterministic event. If they were introduced in individual events, the final raising transition would not be able to refine them all.

A naming convention is adopted for these non-deterministic Event-B events: They are prefixed *raiseInternalTrigger\_* followed by the name of the raised triggers (separated by underscores if more than one). These events are removed once they are refined by a transition.

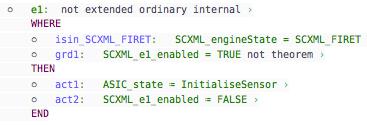


## Raising external triggers

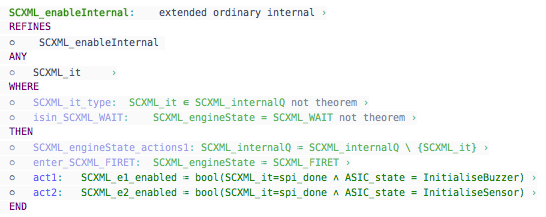
External triggers are always raised non-deterministically by introducing events similar to the *raiseInternalTrigger* events above. Usually only a singleton set containing one trigger would be raised unless there is a real-life case where several external triggers occur simultaneously.

## Introducing users transitions into the ‘engine’

When the users SCXML state-chart is translated into Event-B, new events that represent the users transitions are introduced into a refined version of the basis that describes the engine. These events retain the actions of the SCXML transition including changing to the target state of the SCXML state-chart. However, their guards are all removed and replaced by the enabling flag described above. The event resets the enabling flag to indicate to the engine that it has fired. We also ensure that the event can only fire at the relevant stage of the engine by adding a guard for the engine state. (This is only needed in abstract levels where the engine may proceed without waiting for the event to fire. We discuss this in more detail below).



The guards from the user transition, including any trigger requirement, source state and user entered conditions, are used by the engine to set the enabling flag of the event.



## Proof that transitions satisfy invariants

Removing the guards to the SCXML engine means that it is sometimes less obvious that transitions satisfy invariants.

For example, the transition *incCount* is enabled by the engine when *cnt<max*. and *cnt* is not increased elsewhere. However, the prover cannot initially discharge the PO concerning the range of *cnt*.

Adding an invariant as follows allows automatic proof :-   
 SCXML\_incCount\_enabled = TRUE ⇒ cnt<max\_

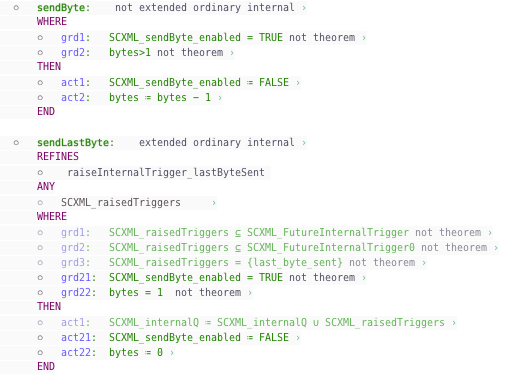
If another transition incremented *cnt*, it would not satisfy this invariant so the invariant provides verification that there is no potential conflict in the system.

## Conditional execution

The recommended way to model conditional behaviour in Event-B is with several events having disjoint guards.

While it is possible to write actions that are conditional this can make proof more difficult. Also, in our example we wish to conditionally raise a trigger by refining a previously non-deterministic raise event. Making this conditional would break the refinement simulation.

We therefore use two alternative events to express the two different conditions. Both conditions are driven from the SCXML engine by the same enabling event. A guard is added directly to the events to express the condition for each. **This is the only situation where we place user guards into events directly**.



## Strengthening guards in refinements

We wish to be able to strengthen guards in refinements. This is a common tactic for making the conditions for behaviour more precise. Since the guards on our events are mostly now calculated in the SCXML engine, changing the guard predicates breaks the refinement simulation PO of the Boolean enable flags.

Therefore, in order to perform guard strengthening we data refine the flags with a new flag and provide a gluing invariant that the prover can use to establish that the transition guards have not been weakened.

SCXML\_e1\_enabled\_1 = TRUE ⇒ SCXML\_e1\_enabled = TRUE

Unfortunately however, the subsequent step of the run to completion engine which waits for the transitions to fire, is guarded by the negation of these guards which have been weakened by the above. I.e. it is **not** true that

SCXML\_e1\_enabled\_1 = FALSE ⇒ SCXML\_e1\_enabled = FALSE

This problem is discussed in more detail in the next section. The solution results in relaxing the abstract levels to allow the engine to continue to the next ‘big step’ without waiting for the enabled transitions to fire.

# Refinement of run to completion semantics

There seems to be an inherent problem with refining ‘run to completion’ semantics. The semantics require that every microstep that can be done, is completed before the next macrostep is started. The inherent problem is that in a refinement we often strengthen the conditions for the microstep. However, by making the microsteps more constrained we make their completion more easily achieved. I.e. we make the guard for taking the next macrostep weaker. This is illustrated by a simpler Event-B example in Appendix B.

<TODO: describe

1. the idea of the solution.. allowing the engine to non-deterministically cycle without waiting for completion.. this is an abstraction of the cycles that will be added in the future refinements.
2. the mechanism of disabling/enabling the completion guards. The engine is allowed to continue to cycle without firing an enabled event. Unless conditions have changed to disable it, the event should be re-enabled and can be fired on the next cycle.

**>**

<FIXME: should re-evaluate the enabledness on each iteration. This means recording the trigger or only consuming it when the event does fire.>

# Appendix A - Answers to Questions

* *Highest Level*
  + *At this level of abstraction, spi\_done looks like an external event. It won’t be; it comes from elsewhere in the ASIC. However, leaving the transition unguarded makes it look like it should exit the state as soon as it enters it, which is also quite false. What is the appropriate way to represent this?*

In Event-B it is best to leave it unguarded since it is up to you how you interpret the event and when it is likely to occur. I.e. in this case you can interpret the transition as happening sometime in the future as a response to spi\_done. However, the SCXML semantics defines an engine that drives the transition, forcing its execution when its guard is true and removing some of the non-determinism. It cannot be an external event since removing an event from the queue later would break the refinement. (It might be possible to refine the event queues with new ones but the execution traces would also change). In fact it is best to model it as an internal event right from the start and add an always enabled transition to raise spi\_bot non-deterministically. However, I am not sure how best to do the transition in SCXML

* + *Similarly, how can we indicate that the transition from the Wait 50 ms state to the Go state isn’t dependent on an external event, but nonetheless doesn’t happen right away?*

As above, SCXML isn’t designed for refinement. There is a possible way suggested in the next answer. But for Wait50ms it might be best to model it as an external event representing a hardware interrupt.

* + *Perhaps we leave all the transitions unguarded. Then is there any way at this level of detail to express some of those details to be filled in later? (I.e. you know that it’s going to move on when some event indicates it’s ready, you just haven’t captured that behavior yet)*

You could perhaps have some Boolean data items that are used as guards and set them non-deterministically using some un-guarded transition, t. Later the Boolean would be refined by some other data or internal event and the transition t would be refined by the added detail that sets that data or raises the event.

* *First Refinement*
  + *Adding a parallel state is not a vertical refinement. However, this type of design process, wherein the designer starts with the high level flow and then adds details of the communication protocol later, is quite natural. It would not make sense to represent the SPI subsystem as a substate of any of the states on the left-hand side, because it would then have to be duplicated multiple times – once for each state that sends a SPI message.*

Adding an independent state-machine is a valid refinement. Less clear about a parallel state but I think it is ok because the only way it affects the existing state is to restrict its enabledness.

* + *Similar to last slide, there are events whose source is unclear. In this case, send\_message will be raised by some of the states on the left, and last\_byte\_sent will be something that happens inside the Sending Message state. If we leave both of those events out altogether, that will allow the SPI subsystem to churn out messages at arbitrary times. Is that a necessary evil at this level? How should this statechart be represented?*

This is valid and fairly common in abstract models. One way to make the abstract model behave is to use some more abstract data representation such as Boolean flags.

* *Second Refinement*
  + *In some parts of the state machine, a transition must happen and must happen immediately. For example, when send\_message is raised, the SPI subsystem must start transmitting the message, or else the whole statechart will freeze up. It also is necessary that one of the two parallel state machines be able to stay in a single state indefinitely while the other takes various actions. Are these inconsistent? Do they, together, compromise stuttering-invariance and, in turn, refinement?*

It sounds fine to me. Why would the other state-machine need to change state?

The new state-machine contributes new behaviour that expands stuttering in the abstract level. As long as it converges it can stutter as much as it likes.

* + *Is there a scoping problem because send\_message is generated at a different level of refinement than it is responded to?*

Yes, there is. You cannot generate an event without responding to it and then add the response later. (In terms of my Event-B representation, it would mean modifying an abstract variable in a new transition, which is not allowed). You either need to leave out the raising of the send\_message event until later or introduce some abstract transition to respond to it without adding details until later.

Similarly if you respond to a trigger event you need to also introduce the raising of the event (external or internal). This can be done (in the Event-B representation) by introducing an Event-B event that raises the trigger event non-deterministically. I am not sure how this could be shown in the SCXML.

* + *There is an intuitive notion that the message is composed of some integer number of bytes, and each byte should be sent sequentially until the last one. The way it is notated right now though is in terms of an event called last\_byte\_sent, which doesn’t indicate that. If we leave off the events altogether, the looping arrow around Send Byte becomes unnecessary, and it looks like it’s only sending one byte. Or an arbitrary number of bytes. How to notate this?*

Not sure I understand the model here. Where are the events ‘last\_byte\_sent’ and ‘!last\_byte\_sent’ raised? Are events an appropriate way to model this? I assumed some local data and un-triggered guarded events here.. but I may have misunderstood the SCXML transition firing.

* *Because of stuttering, the counter isn’t obligated to count. That makes it pretty useless as a delay element. That’s ok for refinement (after all, the thing will eventually be hardware clocked), but it might be a stumbling point for a designer trying to understand it*

Not sure I understand. Maybe we have different ideas of stuttering. How would you guard the transition to Go if you don’t have the count? Anyway, if it is hardware clocked it is probably better to model it as an external trigger event from the start (that’s what I did before I saw refinement 2).

Appendix B –

In the example, the user's SCXML source model has one transition - i.e. represented here by event  ‘userTransition’ with a guard ‘userVar >0'

in the first refinement the user has strengthened the guard for ‘userTransition’ by adding a second conjunct..  ‘userVar >0 & userVar <10'.

However, to simulate the run to completion big-step we introduce the following events:-

‘enableUserTransitions' which atomically evaluates the guards of all the user transitions and enables those that should fire using BOOL flags (e.g. ‘userGuard1'). [Normally this event would also consume a trigger which affects the guards but i have omitted triggers here].

‘userTransitionsFired’ which waits for all the user transitions to be disabled (i.e. completion) before initiating a new big-step

It is this wait for completion which i think is a problem for refinement because it is guarded by 'not(userGuard1=TRUE)' which is weaker than 'not(userGuard=TRUE)'.

i don’t think the problem is caused by my choice of encoding of the semantics, it was the same when we did a much more abstract representation. Unless you can think of a way to represent this completion semantic without introducing an event that is guarded by the negation of the user guards?

I think it is a fundamental conflict between guard strengthening refinement and run to completion semantics (which is used by all the common Harel state-chart modelling languages s.a. UML, SysML SCXML). The stronger the guards the weaker completion becomes.