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

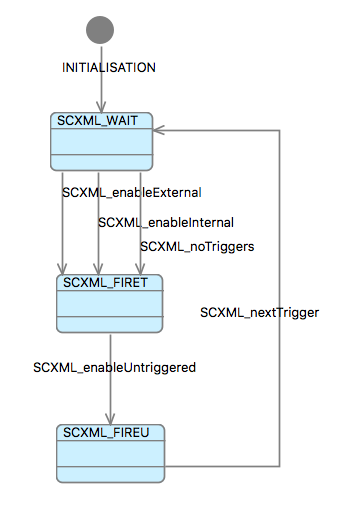
OLD version:

1. 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. If the internal queue is empty but the external queue is not empty, take one event from the external queue and fire the set of transitions that are enabled by it at that time. Again they must be enabled at the time it checks, not subsequently after firing one or more of the set of transitions.
   3. If the internal and external queues are both empty proceed to step 2)
2. 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.
3. repeat from 1)

New version

1. Initialise
2. If there are any untriggered transitions that are enabled fire these transitions that are enabled 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. (Although since the next iteration will again prioritise untriggered transitions, subsequently enable untriggered transitions will fire in the next cycle).
3. Take one of the following options:
   1. If there are no untriggered transitions that are enabled and the internal 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. Again they must be enabled at the time it checks, not subsequently after firing one or more of the set of transitions.
   2. If there are no untriggered transitions that are enabled and the internal queue is empty and the external Q is not empty, take one event from the external queue and fire the set of transitions that are enabled by it at that time. Again they must be enabled at the time it checks, not subsequently after firing one or more of the set of transitions.
4. Repeat from 1).

## SCXML engine

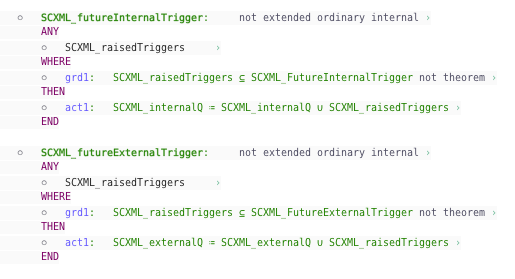
A *basis* machine and context are provided to define the generic elements of the SCXML engine. Specific translated SCXML models start by refining/extending this basis. The basis provides a cyclic engine representing an abstraction of the semantics of SCXML. There are two trigger queues; internal and 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.

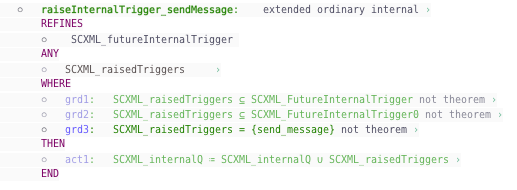
## Refinement of triggers

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.

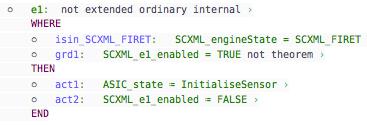


## Raising external triggers

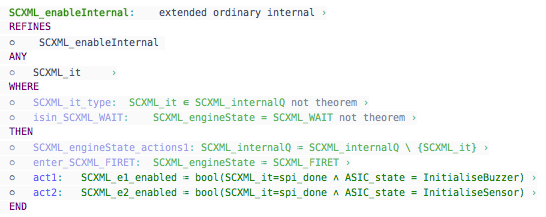
External triggers are always raised non-deterministically by introducing events similar to the *raiseInternalTrigger* events above. Usually only a singleton set containing one trigger would be raised unless there is a real-life case where several external triggers occur simultaneously.

## 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\_

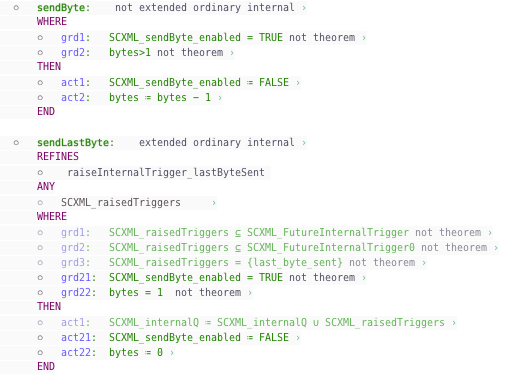
If another transition incremented *cnt*, it would not satisfy this invariant so the invariant provides verification that there is no potential conflict in the system.

## 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

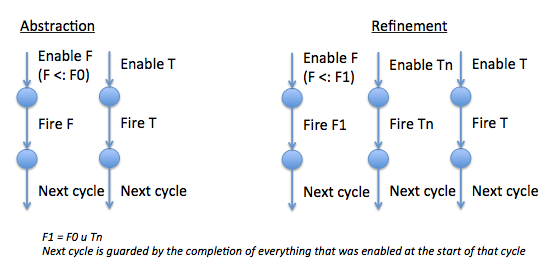
This problem is discussed in more detail in the next section. The solution results in relaxing the abstract levels to allow the engine to continue to the next ‘big step’ without waiting for the enabled transitions to fire.

# Refinement of run to completion semantics

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

In our first attempt to solve this problem we simply disabled the guards that require enabled transitions to complete before the next macro step. The idea is to allow the SCXML engine to continue cycling before it actually takes the enabled transitions. These preliminary cycles represent preliminary (stuttering) steps that will be added in future refinements leading up to the enabling of the enabled transitions. We found that this solution does prove as a refinement (since the troublesome weakened guard is disabled) but the behaviour of this abstraction is not very useful. If several transitions are enabled simultaneously, some may be taken before the next macro step is taken and others not. This leads to many permutations of behaviour that are not permitted in the system. We consider that when transitions are simultaneously enabled, they should all be completed within the same macro step cycle (or none at all). The problem stems from the fact that the abstract ‘phantom’ cycles occur after the enabling decision whereas in the refinement the preliminary stuttering will delay the enabling decision. This causes even more problems for triggered transitions because, if they are not fired they cannot be re-enabled because the trigger has been consumed. We tried several approaches to try to find a better abstraction but were unable to find a way to allow for the introduction of preliminary stuttering macro-steps that retains the important behaviour of ensuring several parallel transitions are fired within one cycle.

The following is a simplified example to illustrate what we are trying to achieve and why it does not work. Each vertical chain of arrows is a macro step and macro steps are executed from left to right. Each arrow is implemented by a set of one or more Event-B events. Capital letters, F, T represent sets of user transitions that are fired in parallel in the same macro cycle. In this example a set T of user transitions are enabled and all fired in a single macro-step of the abstract model. The abstraction allows for future preliminary macro-steps by modelling the execution of a non-deterministic subset of F0, the set of transitions to be introduced in future refinements. In the refinement a preliminary macro-step is introduced which fires a specific set Tn of newly defined user transitions. and the future transitions is reduced correspondingly. In all cases Next cycle should not be enabled until all the user transitions that were enabled at the start of this macro-cycle have fired.



1. the guard for Next cycle is weakened in the refinement due to strengthening the guard for Enable T.
2. Could the guard for Next cycle be done in such as way to avoid specific reference to the enabled transitions? E.g. Enabled = {} where enabled is a set of tokens for each of the enabled transitions. Unfortunately, the guard predicates for each user transition are specific to that transition so it is difficult to see how they can be converted into a generic set of tokens.
3. Could the transitions T be combined into a single event? (this would avoid the problem with the first approach of not waiting for completion). No because the selection of which transitions fire is dynamic depending on their guards.

A possible solution:-

We could generate an event for each combination of transitions. This will be ‘verbose’ but many combinations are not possible because of the event triggering mechanism. I.e. we only consume one trigger at a time and we can use this to reduce the combinations that we need events for. For each trigger, we could generate events for all the combinations of transitions that are enabled by that trigger. Then we also need events for all the combinations of transitions that are un-triggered.

Note that these combinations must be transitions in different parallel machines. When two transitions in the same statemachine are triggered by the same trigger, only one of them can be taken at a time. If both are enabled, we could either make a choice based on priority, or provide a non-deterministic choice. Whichever fires first will disable the other and enabled the next cycle. We need to provide a set of combinations for each of these alternative transitions, but exclude combinations that involve them both.

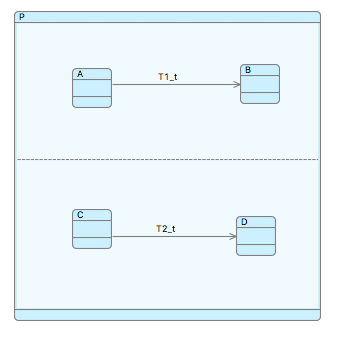
In a refinement we may add a parallel state-machine with more transitions that use the same trigger. This would mean adding more combinations to the set of events covering that trigger. These events would refine the corresponding combination event without the new transition. (The event for the new transition alone would be a new event).

For untriggered transitions we need to make the same analysis of combinations as above but without the exclusions provided by triggering.

TBD:

1. try this strategy in a small abstract example,
2. try this strategy in the SecBOT example (enhance the example to illustrate the point if necessary)

## Parallel Triggered Transitions – Using transition combinations approach

T1 and T2 are both triggered by trigger *t*.

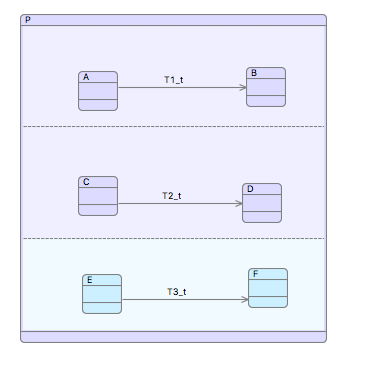
They are in parallel state-machines so could be triggered together, or one at a time depending on other guards such as source state. We could represent these options as 3 events: *(Note that the guards are shown with the events but, for now, are actually used to calculate enabling flag as before. The actions are shown as the transition name, e.g. T1. This is meant to represent all of the transitions actions including changing state, consuming t and any explicitly added actions).*

E1: [A & ~C & t]/ T1

E2: [~A & C & t]/ T2

E3: [A & C & t]/ (T1 || T2)

*(The reason for explicitly modelling T1 || T2 rather than allowing them to fire in non-deterministic order, has to do with the difficulty of refining SCXML semantics and is discussed above).*



In a refinement, another parallel transition T3, also triggered by *t*, is added. It may also fire independently or in combination with the other parallel triggered transitions. We need to add all of these options as a further 4 events. T3 alone refines skip while the other combinations are a case splitting of the three abstract events:

E1: [A & ~C & ~E & t]/ T1

E2: [~A & C & ~E & t]/ T2

E3: [A & C & ~E & t]/ (T1 || T2)

E4: [~A & ~C & E & t]/ T3 (NOT a refinement)

E5: [A & ~C & E & t]/ (T1 || T3) (refines E1)

E6: [~A & C & E & t]/ (T2 || T3 ) (refines E2)

E7: [A & C & E & t]/ (T1 || T2 || T3) (refines E3)

In all cases additional clauses are added to guards hence strengthening them. However, E4 consumes t (i.e. does not refine skip) but there is no abstract transition for it to refine. (Even if the consumption of t is done at the enabling step resulting in nothing enabled, we need an abstract ‘skip’ version of E4 or the engine will be deadlocked).

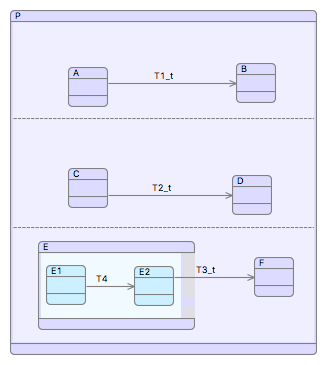
Hence:

E0: [~A & ~C & t]/skip

Is refined by E4. We also retain

E0: [~A & ~C & ~E & t]/skip

In case of further additional transitions that consume t.

In a further refinement a preliminary step T4 is added to T3 which strengthens the guard of T3. Note that T4 cannot fire with any of T1-3 because it is not triggered by t. It is an untriggered transition.

We need to strengthen the guard of all the events involving T3 to ensure they are not enabled until after T4.

Unfortunately this means that T1 and T2 have more scenarios where they can fire without T3. I.e. we should also weaken the guards of events that don’t involve T3:

E1: [A & ~C & ~E2 & t]/ T1 (NOT A REFINEMENT)

E2: [~A & C & ~E2 & t]/ T2 (NOT A REFINEMENT)

E3: [A & C & ~E2 & t]/ (T1 || T2) (NOT A REFINEMENT)

E4: [~A & ~C & E2 & t]/ T3

E5: [A & ~C & E2 & t]/ (T1 || T3)

E6: [~A & C & E2 & t]/ (T2 || T3 )

E7: [A & C & E2 & t]/ (T1 || T2 || T3)

U1: [E1]/ T4

*N.b. E2 => E, but ~E2 /=> ~E Hence E1-3 are not correct refinements*

This is not solved by the previous strategy of allowing the cycle to continue without executing transitions. The weakening is now in the choice of what to enable. Hence, when we strengthen the condition for taking a transition, to include the option of not taking this new transition, we need to refine the enable flag to a more permissive setting predicate. When the flag is used to enable the user transition e.g. E1, a PO is generated to prove that it implies the previous abstract enabling flag which is not true.

To avoid this problem we propose to leave the choices for not taking a transition non-deterministic until we are sure that the guard of that transition will not be strengthened any further.

For example, if the guard of T3 will be strengthened further, we would omit ~E from the guards:

E0: [~A & ~C & t]/skip

E1: [A & ~C & t]/ T1

E2: [~A & C & t]/ T2

E3: [A & C & t]/ (T1 || T2)

E4: [~A & ~C & E & t]/ T3

E5: [A & ~C & E & t]/ (T1 || T3) (refines E1)

E6: [~A & C & E & t]/ (T2 || T3 ) (refines E2)

E7: [A & C & E & t]/ (T1 || T2 || T3) (refines E3)

Hence, T3 can be refined to strengthen its guard from E & t to E2 & t:

E0: [~A & ~C & ~E2 & t]/skip

E1: [A & ~C & ~E2 & t]/ T1

E2: [~A & C & ~E2 & t]/ T2

E3: [A & C & ~E2 & t]/ (T1 || T2)

E4: [~A & ~C & E2 & t]/ T3

E5: [A & ~C & E2 & t]/ (T1 || T3)

E6: [~A & C & E2 & t]/ (T2 || T3 )

E7: [A & C & E2 & t]/ (T1 || T2 || T3)

U1: [E1]/ T4

While the above strategy works for both internal and external triggers, untriggered transitions present an additional problem. We can take the same approach regarding the construction of events for all combinations, but how do we know when no untriggered transitions are enabled and it is time to consume another trigger.

For the untriggered transitions we will again need a skip transition which can be refined by future untriggered transitions.

We can also use a skip transition as a signal event, to indicate that no untriggered transitions are enabled. We propose to add a flag, UC, which indicates that the untriggered transitions are being fired. This flag must be TRUE for any untriggered transition set (including the skip ones) to fire and must be FALSE for any triggered transitions set to fire. It is set TRUE whenever a trigger is consumed. I.e. by the enabling of any triggered transition set (including the skip ones). It is reset FALSE by the untriggered signal event. Appendix D and E show the original SCXML simulation algorithm and how we have derived an equivalent logic that can be represented in Event-B.

The following table shows the four kinds of events that can fire and the generic guards and actions involved in the sequencing. There is a single Completion event and sets of events representing all the combinations of Untriggered transitions, all the combinations of Transitions for each internal trigger and all the combinations of events for each external trigger. The table does not show the individual events nor the enablement and firing of the selected user transitions. Also not shown are the events that model the arrival of a trigger into the external queue.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Type of Event:** | **Completion** | **Untriggered(R)** | **Internal(t,R)** | **External(t,R)** |
| **Guards:** | UC = FALSE | UC = FALSE | UC = TRUE | UC = TRUE |
|  |  | R <: IT | R <: IT | R <: IT |
|  |  |  | t : IQ | IQ = {} |
|  |  |  |  | t : EQ |
|  |  |  |  |  |
| **Actions:** | UC := TRUE | UC := FALSE | UC := FALSE | UC := FALSE |
|  |  | IQ := IQ \/ R | IQ := IQ \/ R \ {t} | IQ := IQ \/ R |
|  |  |  |  | EQ := EQ \ {t} |
|  |  |  |  |  |

The strategy above has been developed and described based on the idea of a simulation engine that represents the run to completion semantics. We now observed that it may be possible to discard the ‘engine’ and just present the events as described above with the guards directly in the events that represent the transitions combinations removing the need for enabling flags. This will slightly increase the complexity of the guards because the guards of the event type (engine) will have to be repeated in with each combination guard, however, removing the ‘engine’ will simplify the model overall.

# Proving properties in SCXML

Notes:

1. Properties about the synchronisation of parallel state-machines (such as ASIC=Go => SPI=IDLE and ASIC=Wait50ms => SPI=IDLE) seem to be important and not easily verified in SCXML. This is a good motivation for using the Event-B translation.  We would like to prove such things at abstract levels.
2. To prove POs about not raising specific internal triggers in abstract 'future' events – the translation can 'look ahead' at the future refinements and add a guard excluding specific internal triggers from being raised in a state if they are not raised in any contained substates/transitions. Alternatively, 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. If it is not obeyed by future transtions, guard strengthening GD proof obligations will make it obvious where the problems lie. For example, the guard…

Go = TRUE ⇒ send\_message ∉ SCXML\_iq’   
 (where SCXML\_iq’ is the new value to be assigned to SCXML\_iq in the event’s actions)  
needs to be automatically added to all the future transitionSet events to prove they do not break the property in 1. This could be automatically generated and added to satisfy all user invariants concerning the raising of internal triggers. If it is not obeyed by future transtions, guard strengthening GD proof obligations will make it obvious where the problems lie.

1. Add a way to exclude specific internal triggers from ever being raised by a user transition (e.g. a doesn't raise element) - Or can this be automatically generated like 1). For example, the guard…   
    send\_message ∉ SCXML\_raisedTriggers  
   needs to be added to the transition Wait50ms\_Go in order to prove that it does not violate the property ASIC=Go => SPI=IDLE. However, if this is to be done manually, we would need a notation that avoids the user having to know about the internal basis parameter, SCXML\_raisedTriggers.
2. There are still things to consider about proving these properties are true at an abstract level - that is, you cannot prove them unless you add the detail that makes them true!! (example :- detail about when we raise send\_message is in level 2 but we need it to prove no messages get in the queue after SPI raises the second spi\_done). However, invariants and guards can be added at the abstract level in order to abstractly add the necessary constraints to make a proof. If these constraints are not maintained in later levels, simpler proof obligations about guard strengthening will be unprovable. These abstract guards should be removed at later refinements when the details have been specified. This implies using ranges in refinement attributes which we have considered but not used in the past.

# 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. [Sept18,2017: with the new translation the later refinements allow the translation to predict that spi\_done will be a internal trigger and it is defined as such and even in the early refinements these transitions have to wait for spi-done to appear in the internal queue. The events that are provided to allow for future refinements ‘FutureInternalTriggerSet’ and ‘FutureExternalTriggerSet’ can be used to raise spi\_done until the refinement where its raising is defined.]

~~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.

Appendix C – Code

import java.util.Arrays;

public class Combination {

public static void main(String[] args){

String[] arr = {"A","B","C","D","E","F"};

combinations2(arr, 3, 0, new String[3]);

}

static void combinations2(String[] arr, int len, int startPosition, String[] result){

if (len == 0){

System.out.println(Arrays.toString(result));

return;

}

for (int i = startPosition; i <= arr.length-len; i++){

result[result.length - len] = arr[i];

combinations2(arr, len-1, i+1, result);

}

}

}

Appendix D – SCXML code for run to completion algorithm

procedure mainEventLoop():

    while running:

        enabledTransitions = null

        macrostepDone = false

        # Here we handle eventless transitions and transitions

        # triggered by internal events until macrostep is complete

        while running and not macrostepDone:

            enabledTransitions = selectEventlessTransitions()

            if enabledTransitions.isEmpty():

                if internalQueue.isEmpty():

                    macrostepDone = true

                else:

                    internalEvent = internalQueue.dequeue()

                    datamodel["\_event"] = internalEvent

                    enabledTransitions = selectTransitions(internalEvent)

            if not enabledTransitions.isEmpty():

                microstep(enabledTransitions.toList())

        # either we're in a final state, and we break out of the loop

        if not running:

            break

        # or we've completed a macrostep, so we start a new macrostep  
        # by waiting for an external event

        # Here we invoke whatever needs to be invoked.  
        # The implementation of 'invoke' is platform-specific

        for state in statesToInvoke.sort(entryOrder):

            for inv in state.invoke.sort(documentOrder):

                invoke(inv)

        statesToInvoke.clear()

        # Invoking may have raised internal error events and we iterate  
        # to handle them

        if not internalQueue.isEmpty():

            continue

        # A blocking wait for an external event.    
        # Alternatively, if we have been invoked

        # our parent session also might cancel us.  
        # The mechanism for this is platform specific,

        # but here we assume it’s a special event we receive

        externalEvent = externalQueue.dequeue()

        if isCancelEvent(externalEvent):

            running = false

            continue

        datamodel["\_event"] = externalEvent

        for state in configuration:

            for inv in state.invoke:

                if inv.invokeid == externalEvent.invokeid:

                    applyFinalize(inv, externalEvent)

                if inv.autoforward:

                    send(inv.id, externalEvent)

        enabledTransitions = selectTransitions(externalEvent)

        if not enabledTransitions.isEmpty():

            microstep(enabledTransitions.toList())

    # End of outer while running loop.    
    # If we get here, we have reached a top-level final state  
    # or have been cancelled

    exitInterpreter()

Appendix E – Derivation of Event-B to represent the R2C algorithm

This is a simplified version of the code in Appendix D to focus on the logic of the algorithm

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

Here is an equivalent logic suitable for modelling with guarded events in Event-B. In event-B we already have an implicit outer ‘while true’ loop (we omit the ‘running’ condition and allow the simulation to idle forever). We omit the completion flag since this is equivalent to an empty internal queue and no untriggered transitions being enabled.

while true //implicit loop of Event-B  
 if untriggered\_enabled  
 execute(untriggered())  
 elseif IQ /= {}  
 execute(internal(IQ.dequeue))   
 elseif EQ /= {}  
 execute(EQ.dequeue)   
 endif  
endwhile

removing the else clauses

while true //implicit loop of Event-B  
 if untriggered\_enabled  
 execute(untriggered())  
 if not(untriggered\_enabled) & IQ /= {}  
 execute(internal(IQ.dequeue))   
 if not(untriggered\_enabled) & IQ = {} & EQ /= {}  
 execute(EQ.dequeue)   
 endif  
endwhile

Representing the condition ‘untriggered enabled’ is cumbersome since we would need to write a conjunction of all the possible untriggered guards. Instead we introduce a dummy untriggered event that is fired when no other untriggered event is enabled and this sets a Boolean flag, UC, to indicate that none of the real untriggered events was fired.

UC := FALSE  
while true //implicit loop of Event-B  
 if UC = FALSE  
 execute(untriggered()) // only the dummy event will set UC := TRUE  
 if UC = TRUE & IQ /= {}  
 execute(internal(IQ.dequeue)) || UC := FALSE  
 if UC = TRUE & IQ = {} & EQ /= {}  
 execute(EQ.dequeue) || UC := FALSE  
 endif  
endwhile