This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

SAND2021-10691J

Formal verification and validation of run-to-completion style state-charts using Event-B

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

[1] Sandia National Laboratories, 7011 East Avenue Livermore, California 94550, USA
{knmorri**,rob,ghulett}@sandia.gov
[2] ECS, University of Southampton, Southampton SO17 1BJ, United Kingdom
{cfs,t.s.hoang,mjb}@soton.ac.uk

Abstract. State-chart notations with 'run to completion' semantics, are popular with engineers for designing controllers that react to environment events with a sequence of state transitions but lack formal refinement and rigorous verification methods. State-chart models are typically used to design complex control systems that respond to environmental triggers with a sequential process. The model is usually constructed at a concrete level and verified and validated using animation techniques relying on human judgement. 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. Abstraction and formal verification provides greater assurance that critical (e.g., safety or security) properties are not violated by the control system. In this paper, we introduce a notion of refinement into a 'run to completion' state-chart modelling notation, and leverage Event-B's tool support for theorem proving. We describe the difficulties in translating 'run to completion' semantics into Event-B refinements and suggest a solution. We illustrate our approach and show how models can be validated at different refinement levels using our scenario checker animation tools. We show how critical invariant properties can be verified by proof despite the reactive nature of the system and how behavioural aspects of the system can be verified by testing the expected reactions using a temporal logic, model checking approach. To verify liveness, we outline a proof that the run to completion is deadlock free and converges to complete the run.

Keywords: run-to-completion, state-charts, refinement, Event-B

1 Introduction

Reactive State-charts are open systems capable of receiving potentially non-deterministic input. State-charts provide a graphical language, generalized from

** Corresponding author, ORCID iD: https://orcid.org/0000-0002-0146-3176, telephone: +1(925) 294-3287

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

state machines, that is popular with engineers. Variants appear in Matlab Simulink/Stateflow [14] and the Ansys tools. It is particularly attractive, to provide accessibility to abstraction/refinement via Rodin/Event-B which has an intuitive metaphor in the State-chart semantics [16,18,17]. The hope is that engineers can better understand the origin of proof obligations in refinements and achieve formal guarantees earlier in their designs where it is most tractable. Our approach is focused on a mapping to Event-B where safety properties preservation is key. In our version of State-chart semantics, refinement means a subset of traces from an abstraction. This has the beneficial effect of preserving safety properties from abstraction to refinement and permits proofs to be discharged at the highest tractable level of abstraction where they are the easiest to discharge.

This manuscript is structure as follow; section 2 discusses related work on refinement and what are the different types. Section 3 provides background material. Section 4 discusses the State-chart concept of 'run to completion' and how it can be specified in Event-B. Section 5 states the different state-chart refinement rules we use to construct models. Section 6 introduces our example case study; a drone. Section 7 gives an outline of our translation from State-Chart XML (SCXML) to Event-B via UML-B. Section 8 describes the use of UML-B animation and Scenario Checking tools to validate translated SCXML models. Section 9 illustrates our approach to verifying safety invariant properties. Section 10 illustrates our approach to verifying control responses, and Section 11 concludes.

2 Related Work

Many incompatible definitions of refinement have been posed by others [5,13] and that can lead to confusion. Though these separate refinements have different goals, all of which may be attractive to systems designers in different ways, they will not always preserve safety properties. From the Event-B vernacular it might be better to relabel these other approaches not as methods of model "refinement", but rather methods of model "elaboration". Preservation of safety properties across refinement requires only a few restrictions to the original [6] State-charts (e.g., transitions cannot cross containment boundaries arbitrarily), but still allows for both parallel and hierarchical composition. With these restrictions composition becomes a refinement, but not all refinements are compositions. Such a unification of composition and refinement can lead, not only to code reuse, but reuse of proofs.

If an Event-B model B can be shown (via the construction rules of the Event-B language as well as the proof obligations) to refine another Event-B model A, then we know that every behavior of B is also a behavior of A. This definition yields a useful principle of preservation of safety – if we can show that a bad thing never happens in A, then we can add detail via refinements in B, knowing that the bad thing will continue to never happen in B. That is, Event-B refinements preserve safety properties in the sense adopted by Lamport [11]. This makes refinement a useful technique in developing safety-critical systems:

one can analyze a simpler abstract model for critical safety properties and then add detail to the model via refinements, secure in the knowledge that the safety properties will be preserved. While Event-B refinements have also been shown to preserve some liveness properties under certain conditions [8], there are not yet efficient supporting tools for the technique. Instead, we can express the property in Linear Temporal Logic (LTL) and use the ProB[3] model checker to verify it, as we have shown in previous work [17]. In this paper, we outline a proof of liveness properties that relies on reasoning about deadlock-freeness and event convergence.

A method that is closely related to Event-B and also supports reasoning about safety and liveness properties is TLA+ [12]. TLA+ is supported by the TLA+ Toolbox [10]. On the one hand, temporal properties (both safety and liveness) are *explicitly* stated as properties of the TLA+ models and reasoning about them often requires applying proof rules related to properties of traces. On the other hand, Event-B defines proof obligations based on the underlying trace semantics [1,8,9], hence reasoning about *implicit* temporal properties in Event-B simply involves discharging the relevant proof obligations. Furthermore, at the time of writing, the TLA+ Proof System (part of the TLA+ Toolbox) does not fully support the reasoning with many temporal operators.[4]

3 Background

3.1 Event-B

Event-B [1,7] is a formal method for system design. It uses *refinement* to introduce system details gradually into the formal model. An Event-B model contains two parts: *contexts* and *machines*. Contexts contain *carrier sets*, *constants*, and *axioms* constraining the carrier sets and constants. Machines contain *variables* v, *invariants* $I(v)$ constraining the variables, and *events*. An event consists of a guard denoting its enabled-condition and an action defining the value of variables after the event is executed. In general, an event e has the form: **any** t **where** $G(t, v)$ **then** $S(t, v)$ **end** where t are the event parameters, $G(t, v)$ is the guard of the event, and $S(t, v)$ is the action of the event.

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 by replacing some (abstract) variables by new (concrete) variables (*data refinement*). Refinement in Event-B is reasoned on an event basis. A (concrete) event f refines an (abstract) event e if whenever f is enabled then e is also enabled (guard strengthening), and the action of f is the same or equivalent to e (where equivalence is given by some relationship defined in the invariants). New events are said to refine 'skip' (an implicit abstract event

[3] ProB is an animator, constraint solver and model checker for the B-Method. https://www3.hhu.de/stups/prob

[4] http://tla.msr-inria.inria.fr/tlaps/content/Documentation/Unsupported_features.html (accessed June 2021)

that did nothing), and therefore do not alter abstract variables. More information about Event-B refinement can be found in [1]. Event-B is supported by the Rodin Platform (Rodin[5]) [2].

Proof obligations are generated to ensure the consistency of Event-B models. An important proof obligation in Event-B is invariant preservation to prove that safety properties (encoded as invariants of the models) will not be violated for any reachable states. In this paper, we also make use of other proof obligations in Event-B such as (relative) deadlock-freeness and (conditional) event convergence to construct our proof of liveness properties under some fairness assumptions.

For the trace semantics corresponding to Event-B machines and the interpretation of LTL properties over traces, we refer the readers to [8]. Here, we recall the notation for fairness assumptions underlying event-based formalisms such as Event-B [11,9]. Given an event e, a weak-fairness assumption $\mathsf{WF}(\mathsf{e})$ states that if e is enabled continually, then it must occur infinitely often. Similarly, a strong-fairness assumption $\mathsf{SF}(\mathsf{e})$ states that if e is enabled infinitely often, then it must occur infinitely often. Formally,

$$\mathsf{WF}(\mathsf{a}) \Leftrightarrow (\mathsf{FG}\,\mathsf{enabled}(\mathsf{e}) \Rightarrow \mathsf{GF}\,[\mathsf{e}])\ , \text{ and}$$
$$\mathsf{SF}(\mathsf{a}) \Leftrightarrow (\mathsf{GF}\,\mathsf{enabled}(\mathsf{e}) \Rightarrow \mathsf{GF}\,[\mathsf{e}])\ ,$$

where G and F are the temporal operators denoting *globally*, and *finally*, respectively; and $\mathsf{enabled}(\mathsf{e})$ denotes that event e is enabled and $[\mathsf{e}]$ denotes an occurrence of event e.

3.2 UML-B State-machines

UML-B [19] 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. 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. While the UML-B translation deals with the basic data formalisation of state-machines it differs significantly from the semantics discussed in this manuscript. UML-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 UML-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 UML-B transitions.

[5] 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.

3.3 SCXML

SCXML is a modelling language based on Harel state-charts with facilities for adding data elements that are modified by transition actions and used in conditions for their firing [21]. SCXML follows a 'run to completion' semantics, where trigger events[6] 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 firing the un-triggered transitions that 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. Note that the enabledness of transitions is calculated batch-wise at each step, not after each and every transition. Hence the set of parallel transitions that are enabled by a trigger is calculated and then only those are fired, irrespective of whether firing one may disable or enable another. Similarly, the set of parallel untriggered transitions to be fired is calculated at each iteration before any is fired. There are two kinds of triggers: internal triggers are raised by transitions and external triggers are raised by the environment (non-deterministicly for the purpose of our analysis). An external trigger may only be consumed when the internal trigger queue has been emptied.

State-charts, with 'run to completion' semantics, are considered to be a synchronous language in the sense that the external triggering event waits for the behaviour that it enables to complete before making any further progress. In contrast, Event-B has an asynchronous semantics due to the non-deterministic selection of events to fire. Of course, synchronous behaviour can be explicitly modelled by the addition of control variables that define the enabledness of events (i.e., remove the non-determinism). This is how we can define the translation suggested in this paper. UML-B state-machines constrain the firing of transitions to some extent but, like Event-B, do not have an underlying fully synchronous semantics. The advantage of an asynchronous semantics is its flexibility. However, when we wish to model processes that are essentially synchronous in nature, the need to explicitly add the synchronous semantics to each model becomes a burden, obscuring the particular problem being modelled. Since many components (e.g., controllers) used in a system are based on synchronous behaviour, we are interested in adapting a modelling language with run to completion semantics to support Event-B style refinement. We chose SCXML as our source language because it is relatively simple compared to some run to completion modelling languages yet has a well defined action language and simulation tool support.

Listing 1 shows a pseudocode representation of the run to completion semantics as defined within the latest W3C recommendation document [21]. Here IQ and EQ are the triggers present in the internal and external queues respectively. We adopt the commonly used terminology where a single transition is called a *micro-step* and a complete run (between de-queueing external triggers) is referred to as a *macro-step*.

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

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

Listing 1. Pseudocode for 'run to completion'

SCXML does not contain any notion of refinement. A single model contains all details to the finest level hence making it difficult to verify and validate that the model behaves safely and as intended. Our aim in this work is to support formal refinement of SCXML models so that verification can be carried out at abstract levels before all details are present. However, applying the refinement rules in the presence of the run to completion semantics is not straightforward. For example, if we apply Rule A (see Section 5) to a transition, it is more easily disabled causing the run to complete earlier (i.e., completion has a weaker guard), breaking Rule A.

4 Run To Completion

The run to completion semantics is specified via an abstract basis that is extended by the model [16,18]. Figure 1 shows a state-chart representation of how the basis enforces the run to completion semantics on the model transitions.

The specification of 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, shown in Listing 2, introduces a set of all possible triggers, SCXML_TRIGGER which is partitioned into internal and external triggers (e.g FutureInternalTrigger and FutureExternalTrigger respectively), some of which will be introduced in future refinements. At each refinement these trigger sets are further partitioned to introduce more concrete triggers, leaving a new abstract set to represent the remaining triggers yet to be introduced.

The context also models *sequences* of triggers as a data type to be used for the trigger queues. Our initial work modelled queues abstractly as sets of triggers which was adequate for most verification purposes but does not enforce fairness

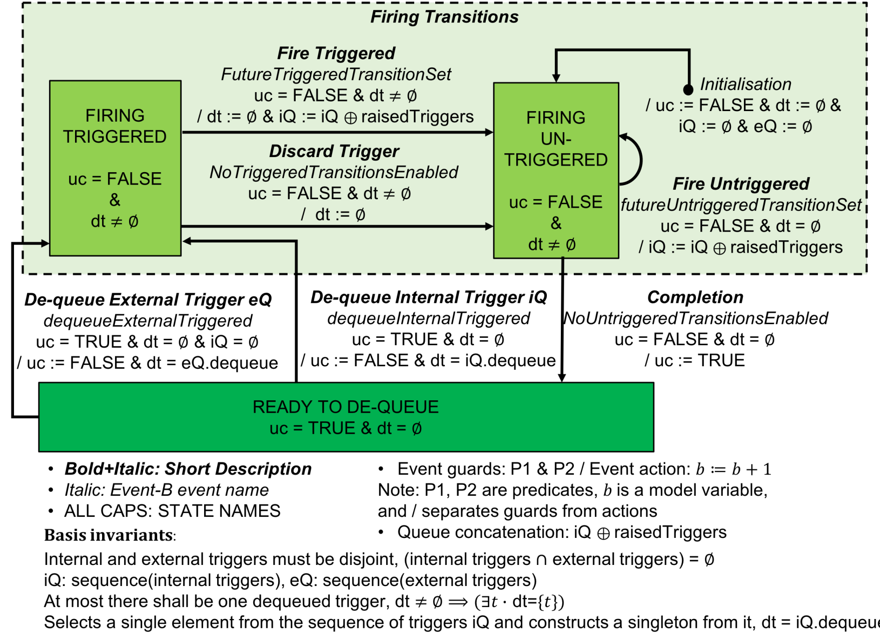

Fig. 1. Abstract representation of run to completion basis

on trigger consumption [16,18,17]. Hence we were forced to introduce fairness assumptions regarding trigger consumption in order to verify liveness properties. In this paper we introduce sequences to properly model the trigger queues which are an implementation of this fairness property. Note that the queue also enables the same trigger to be raised twice in the queue which was not possible in a set. The constant Seq returns the set of all possible sequences of a given subset of triggers and is defined using lambda calculus. Constant functions are also defined for the usual operations on sequences: *length* of a given sequence, *append* a trigger to the end of a sequence to give a new sequence, *concat*enate two sequences to give a new sequence, return the trigger at the *head* of a sequence, return the sequence that makes up the *tail* of a sequence and return the *content* (set of triggers) involved in a sequence. The Basis context also defines several theorem properties about sequences that are needed to discharge proof obligations. These are omitted from Listing 2 for brevity.

Each of the transitions in the basis (see Figure 1) represents an abstract event of the basis machine (Listing 3) that describes the generic behaviour of models under a run to completion semantics. These events provide an abstraction that defines the altering of trigger queues and completion flag. Event-B refinement rules prohibit new events from modifying abstract variables (i.e., new events refine 'skip'). Hence, since SCXML transitions need to modify the trigger queues etc., used to capture the SCXML run to completion semantics, all events generated by translation of the specific SCXML model, must refine abstract events introduced for this purpose in the basis. The basis machine also declares variables that correspond to the currently dequeued trigger, dt, the queue of internal triggers raised by actions within the model, iQ, the queue of external triggers

```
1  context
2      basis_c                          // (generated for SCXML)
3  sets
4      SCXML_TRIGGER                    // all possible triggers
5  constants
6      FutureInternalTrigger            // all possible internal triggers
7      FutureExternalTrigger            // all possible external triggers
8      Seq                              // return all possible sequences of given set of triggers
9      length                           // return the length of a sequence of triggers
10     append                           // return the result of appending trigger to a sequence
11     concat                           // return the concatenation of two sequences of triggers
12     head                             // return the trigger at head of a sequence of triggers
13     tail                             // return the sequence at tail of a sequence of triggers
14     content                          // return the set of triggers in a sequence of triggers
15     InternalQueueType                // type of internal queues
16     ExternalQueueType                // type of external queues
17
18 axioms
19     partition(SCXML_TRIGGER, FutureInternalTrigger, FutureExternalTrigger)
20     Seq = (λT · T ⊆ SCXML_TRIGGER | {n, s · n ∈ ℕ ∧ s ∈ 0 .. n − 1 → T | s})
21     length = (λs · s ∈ Seq(SCXML_TRIGGER) | card(s))
22     append = (λ s ↦ t · s ∈ Seq(SCXML_TRIGGER) ∧ t ∈ SCXML_TRIGGER |
23         {i↦v | i ∈ 0 .. length(s) ∧ (i < length(s) ⇒ v = s(i)) ∧ (i = length(s) ⇒ v = t) } )
24     concat = (λ s1 ↦ s2 · s1 ∈ Seq(SCXML_TRIGGER) ∧ s2 ∈ Seq(SCXML_TRIGGER) |
25         {i↦v | i ∈ 0 .. length(s1) + length(s2) − 1 ∧ (i < length(s1) ⇒
26                            v = s1(i)) ∧ (i ≥ length(s1) ⇒ v = s2(i − length(s1))) } )
27     head = (λs · s ∈ Seq(SCXML_TRIGGER) ∧ s ≠ ∅ | s(0))
28     tail = (λs· s∈Seq(SCXML_TRIGGER) ∧ s≠∅ | {i↦v | i∈0..length(s)−2 ∧ v=s(i+1)} )
29     content = (λs · s ∈ Seq(SCXML_TRIGGER) | ran(s))
30     InternalQueueType = Seq(FutureInternalTrigger)
31     ExternalQueueType = Seq(FutureExternalTrigger)
32 end
33
```

Listing 2. Abstract basis context

raised by the environment, eQ, and a flag, uc, that signals when a run to completion macro-step has been completed (no un-triggered transitions are enabled). Note that, for convenience, the currently dequeued trigger, dt, is modelled as a singleton set which may be empty (i.e., consumed) or contain the single trigger to be consumed.

The trigger queues and dequeued trigger are initialised to empty and uc is set to FALSE so that any enabled un-triggered transitions are dealt with via the futureUntriggeredTransitionSet event when the system first starts (see Listing 1). This will subsequently enable completion and reset the uc flag to TRUE. The abstract event futureRaiseExternalTrigger represents the raising of an external trigger (not shown in Figure 1). After completion, a queued trigger can be prepared for consumption by moving it to the dequeued trigger, dt. Internal triggers have a higher priority, since the external trigger queue is only dequeued if the iQ is empty (see dequeueExternalTriggered and dequeueInternalTriggered in Figure 1). The abstract event futureTriggeredTransitionSet represents a combination of parallel transitions that may be simultaneously triggered by the dequeued trigger, dt. When the actual example SCXML is translated, a separate refinement of this abstract event will be generated for each subset of the set of parallel transitions that could fire in parallel in order to cater for all possibilities of enablement, however, as the model is refined some combinations may be eliminated as the

guards are strengthened. This approach to generating an event for each possible combination of each set of transitions that could fire in parallel is needed because of the batch enabling semantics of the SCXML run to completion (see Section 3.3). The actions of these transitions may also raise triggers of their own in the internal trigger queue iQ.

Completion of triggered and untriggered transitions may be non-deterministically premature to allow future refinements to strengthen the guards of transitions (i.e., to disable them resulting in an earlier completion). In the process of refining a model, a designer takes advantage of this non-determinism in the abstraction by adding nested sub-states and explicit guards to transitions. When a refinement level is reached where the designer wants to enforce a requirement (i.e., prevent it being bypassed by a non-deterministic completion), the model needs to be *finalised* (see Section 7 for more on finalisation). The SCXML translation tool will then automatically strengthen the guards of events noTriggeredTransitionsEnabled and noUntriggeredTransitionsEnabled, to ensure that the run to completion sequence is not interrupted by non-deterministic behaviour. To do this we need to guard completion so that it cannot happen while any relevant transition is still enabled. To finalise a triggered transition, the guard of noTriggeredTransitionsEnabled is strengthened by adding the conjunction of the negated guards of all transitions that can fire in parallel with the transition being finalised. Similarly, to finalise an untriggered transition, the guard of noUntriggeredTransitionsEnabled is strengthened by adding the conjunction of the negated guards of all untriggered transitions that can fire in parallel. It may seem that finalisation could cause an unmanageable explosion of guards. However, to fire in parallel, transitions must be contained in parallel regions and also be enabled by the same trigger (or be un-triggered). In practice, since most systems do not contain many parallel regions, the number of transitions that can fire in parallel is limited. Transition finalisation can be left until it is needed for the proof of a particular property and does not generate any new proof obligations since adding guards is a trivial refinement step. Finalisation is also needed in order to remove non-deterministic behaviours when the model is animated for validation purposes.

5 State-chart Refinement

The work presented here includes three refinement rules.

1. *Rule A:* Guard conditions on a transition can be strengthened (but not weakened); this can be done by adding textual guards to the transition, or changing the source of the transition to a nested state.
2. *Rule B:* Transitions can have additional actions, provided they do not modify variables appearing in the abstraction; this can be accomplished by adding textual action to the transition or by changing the target to nested state.
3. *Rule C:* A state-chart can be embedded within a state of another state-chart – sometimes called hierarchical composition or hierarchical refinement.

The application of these rules is illustrated in figure 2. *Rule A* is applied to refine the abstract transition from SA to SB after adding child states SC

```
1  MACHINE basis                        // (generated for SCXML)
2  SEES   basis_ctx
3  VARIABLES
4    iQ                                 // internal trigger queue
5    eQ                                 // external trigger queue
6    uc                                 // run to completion flag
7    dt                                 // dequeued trigger for this run
8  INVARIANTS
9    iQ ∈ InternalQueueType             // internal queue
10   eQ ∈ ExternalQueueType             // external queue
11   uc ∈ BOOL                          // completion flag
12   dt ⊆ SCXML_TRIGGER                 // dequeued trigger
13   dt≠∅ ⇒(∃t·dt={t})                  // at most one dequeued trigger
14 EVENTS
15  INITIALISATION                      // queues empty, completion false, no dequeued triggers
16    iQ, eQ, uc, dt := ∅, ∅, FALSE, ∅
17  END
18
19  futureRaiseExternalTrigger          //basis of future event to raise an external trigger
20   ANY  raisedTrigger WHERE raisedTrigger ∈ FutureExternalTrigger
21   THEN  eQ:=append(eQ ↦ raisedTrigger)
22  END
23
24  dequeueInternalTriggered            //event to dequeue an internal trigger
25   WHEN  iQ≠∅ ∧ dt=∅ ∧ uc=TRUE
26   THEN  dt, iQ, uc := {head(iQ)}, tail(iQ), FALSE
27  END
28
29  dequeueExternalTriggered            //event to dequeue an external trigger
30   WHEN  eQ≠∅ ∧ dt=∅ ∧ uc=TRUE ∧ iqQ∅
31   THEN  dt, eQ, uc := {head(eQ)}, tail(eQ), FALSE
32  END
33
34  futureTriggeredTransitionSet        //basis of future event representing triggered transitions
35   ANY trigger, raisedTriggers WHERE
36    trigger ∈ dt ∧ uc = FALSE ∧ raisedTriggers ∈ Seq(FutureInternalTrigger)
37   THEN  dt, iQ := ∅ , concat(iQ↦raisedTriggers)
38  END
39
40  noTriggeredTransitionsEnabled       //event to fire when no triggered transitions enabled
41   WHEN  uc=FALSE ∧ dt≠∅
42   THEN  dt := ∅
43  END
44
45  futureUntriggeredTransitionSet      //basis of future event representing untriggered transitions
46   ANY raisedTriggers  WHERE uc=FALSE ∧ dt=∅ ∧ raisedTriggers∈Seq(FutureInternalTrigger)
47   THEN iQ := concat(iQ↦raisedTriggers)
48  END
49
50  noUntriggeredTransitionsEnabled     //event fired when no untriggered transitions enabled
51   WHEN  uc=FALSE ∧ dt=∅
52   THEN  uc := TRUE
53  END
54 END
55
```

Listing 3. Abstract basis machine

and SD. The refinement strengthens the guard of the transition by restricting it to SD. On the other hand, *Rule B* refines the abstraction by introducing a new concrete variable, x, into the model. The abstract transition is refined by the actions associated with this new variable. Finally *Rule C* constructs a

refinement introducing state-charts SC, SD and SG, SH through hierarchical and parallel composition respectively.

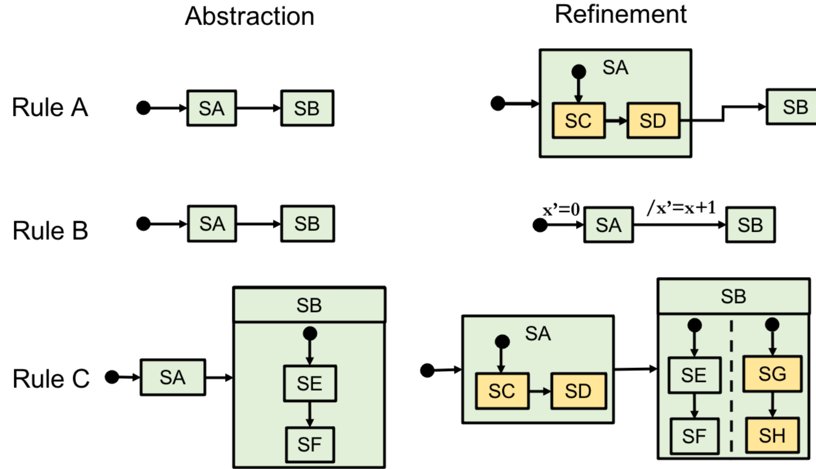

Fig. 2. State-chart refinement rules

Via the translation explained in Section 7, these rules rely on the usual Event-B proof obligations to ensure that they do indeed yield refinements in the Event-B semantics.

6 Description of the Sample Application

To illustrate the development and analysis process of a design using the previously described state-chart semantics, we will discuss a quadrotor helicopter or quadrotor application similar to the one presented by Syriani et al. [5]. The application will focus on the incremental design of some of the drone's required functionality. The constructed model must obey state-chart refinement rules listed in Section 5, these rules are proven within the Rodin tool. The structure of the state-chart for this model at each subsequent abstraction level restricts further the development of the model to refinements that obey the rules. This will allow us to prove properties of the model in a very strategic fashion, as properties proven of early abstraction levels are preserved in later refinements. Unlike other case studies previously presented [15,16,18], this drone example illustrated the model construction and analysis of a more complex system with several parallel components and refinement levels.

The initial abstraction and first refinement of the model, shown in Figure 3, capture the basic functionality of the drone. The abstract model is shown in blue; the model's initial state is OFF and as a result of the on and toTakeoff external

triggers it transitions to the START and OPERATIONAL states respectively[7]. The drone reacts to the off external trigger by shutting down and subsequently transitioning to the OFF state. The first refinement is constructed using *Rule C*, which adds details within the OPERATIONAL state (gray states in Figure 3). Within the OPERATIONAL state the drone will transition to FLY or DESCEND after the internal trigger toFly or toLand is raised, respectively. In refinement level one, these internal triggers are raised non-deterministically in the system by functionality not currently defined. As additional details are incorporated into the model in later refinements some of that non-determinism is removed and replaced by transitions with actions that raised the previously defined internal triggers. A further external trigger, landed, directs the system to progress to the LANDED state. It should be noted that this abstraction of the drone model includes a transition from TAKEOFF to DESCEND (dashed transition in Figure 3). This allows for the drone to respond to a toLand trigger if it encounters some problems while in the TAKEOFF state. Syriani et al. [5] introduces this transition in later refinements under Rule 8 *path refinement rule*. This rule is inconsistent with our rules of refinement as it results in a concrete event with no corresponding behavior in the abstraction.

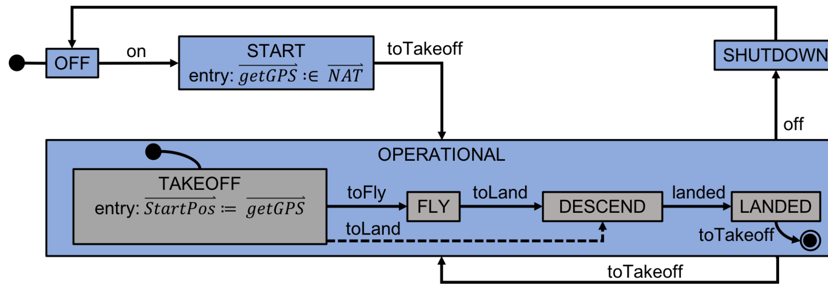

Fig. 3. State-chart of drone application. Abstract level including only generic start/shutdown behavior (shown in blue). The first refinement introducing main operational sub-states, is shown in gray.

Figure 4 builds on Figure 3 to show three more refinements to the drone model. The second refinement (shown in green in Figure 4) extends the capabilities within OPERATIONAL by using *Rule C* to make it a parallel state that controls flying and battery related functionality. This is the same as Rule 4 *and-state rule* defined by Syriani et al. [5]. The charge within the drone battery is monitored by the parallel BATTERYOP state. A new ancillary variable, charge, is introduced to keep track of the amount of charge left in the drone. It is decreased by a self-transition on state BATTERYOK in response to an external trigger decreaseCharge. If the battery monitor works correctly we would expect the battery charge to have at least 20% capacity while in the state BATTERYOK. This can be expressed as an invariant property:

$$(\mathsf{BATTERYOK} = \mathsf{TRUE}) \Rightarrow \mathsf{charge} > 20\% \ .$$

[7] Transitions in Figures 3–4 are labeled with trigger names (e.g., toTakeoff, toFly) not with event names as it is in UML-B.

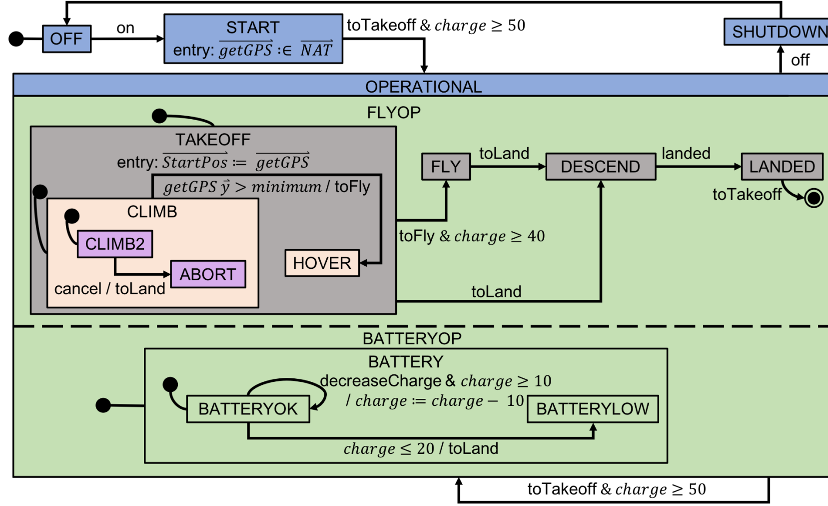

Fig. 4. State-chart of drone application. 2nd refinement for battery monitoring functionality (shown in green). 3rd refinement introducing details for take off (shown in beige). 4th refinement level to allow cancelling during take-off (shown in lilac).

When the monitored charge drops to 20% or less, the BATTERY state-chart raises the internal trigger toLand, which will cause a reaction in the FLYOP start-chart to bring it out of TAKEOFF or FLY and into DESCEND (hence removing some of the non-determinism concerning where toLand is raised). While in the TAKEOFF state we would expect the battery monitor to be in the BATTERYOK state or to have raised a toLand trigger.

$$(\mathsf{TAKEOFF} = \mathsf{TRUE}) \Rightarrow (\mathsf{BATTERYOK} = \mathsf{TRUE} \lor \mathsf{toLand}) \ .$$

To ensure the drone only enters TAKEOFF or FLY with enough battery power we strengthen the guards of transitions to the FLY and TAKEOFF states (*Rule A*). We will discuss how these state invariant properties are verified in Section 9.

The third refinement of the model (shown in beige) refines the state TAKEOFF by applying *Rule B and C*. Under these rules we introduce child states and new model variables, similar to Rule 2 *basic-to-or state rule* defined by Syriani et al. [5] As part of this refinement we introduced an untriggered transition responsible for raising the toFly internal trigger, and therefore removed some of the non-determinisms concerning this trigger.

The fourth refinement of the drone model (shown in lilac) uses *Rule C* to introduce additional implementation details to allow a take-off to be cancelled in response to an external trigger cancel. If the trigger is raised, the climbing process must be aborted and the drone descending sequence shall start. This refinement level is done differently to Syriani et al. [5], which follows Rule 7 *state extension rule*. The aforementioned rule requires a data remapping of the abstract states TAKEOFF, CLIMB and HOVER, which should be distinct from the states in this refinement, as the state ABORT is introduced. In contrast, we

implement this refinement using a rule similar to Syriani et al.'s Rule 2 *basic-to-or state rule*, which introduces the concrete states CLIMB2 and ABORT to the abstract state CLIMB.

Although the autonomous drone example in this paper is based on the example described in [5], the definition of refinement used in that work is quite different from our own. This forces some differences in our refinement rules and consequently the way the example is developed. In [5] "refinement" is a transformation of the model which preserves reachability of a state with respect to sequences of inputs. However, this also allows the possibility of introducing new behaviors in the concrete model that the abstraction does not exhibit. While this notion of refinement seems useful in certain contexts, unlike refinement in Event-B it does not guarantee preservation of safety properties. Therefore, it should be considered less suited to development of safety-critical systems.

7 SCXML Translation to UML-B/Event-B

The translation of a specific SCXML model to UML-B and Event-B, comprises the following stages:

- Firstly, a basis machine and context are created to embody the semantics of the SCXML language (as described in Section 4). The basis provides variables and events to model the queue of triggers as well as abstract versions of events to model transitions firing. The basis is independent of the particular SCXML model which is added in subsequent refinements. Hence it is not necessary to re-prove any of the proof obligations associated with this basis.
- Secondly, all possible combinations of each set of transitions that can fire together are calculated and corresponding events are generated, at appropriate refinement levels (given by the refinement annotations embedded in the SCXML model), that refine the abstract basis events. The transitions that can fire together are those that are triggered by the same trigger (or are both untriggered) and are in different parallel ('and') sub-states. For example, the untriggered transitions shown in the parallel states FLYOP and BATTERYOP of figure 4 are combined into an event in the Event-B representation of the model, through a conjunction of the guards and actions of each of the transitions. If these transitions raise internal triggers, a guard, $\{\mathsf{i1}, \mathsf{i2}, \ldots\} \subseteq \mathsf{content}(\mathsf{raisedTriggers})$ (where $\mathsf{i1}, \mathsf{i2}, \ldots$ have been added to the internal triggers set), is introduced to define the raised triggers parameter. The subset used in the guard retains non-determinism to allow more triggers to be raised in later refinements. For triggered transitions, the trigger is specified by a guard that defines the value of the trigger parameter.
- Thirdly, at each refinement level, the SCXML state-chart is translated into a corresponding UML-B state-machine whose transitions elaborate (i.e., add state change details to) the 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.

– Finally the UML-B state-machine is translated into Event-B by programmatically invoking the UML-B translator.

A previous version of the translator was described in [16,17] New features of the translation added since [16,17] are as follows:

Trigger queues in basis: The encoding of trigger queues in the abstract basis context and machine has been improved so that a queue is properly modelled as a sequence of triggers. This more accurately reflects the SCXML semantics.

Dequeing triggers from queues: The abstract basis machine has been improved so that triggers are properly dequeued before potential use, which allows triggers to be discarded if the controller cannot respond to them. This more accurately reflects the SCXML semantics and was necessary in order to model the new drone case study properly.

Finalisation: Transitions can be flagged as finalised which means their guards can not be strengthened in subsequent refinements. This allows them to 'enforced' when they are enabled (i.e., completion cannot occur until they have fired) which is needed for verification.

Restricted raising of internal triggers: Once a trigger is introduced it must immediately be raised at that refinement level by any transitions that wish to do so. It cannot be raised in later refinements except by newly introduced transitions. This restriction was necessary to make simulation more useful by removing non-deterministic raising of triggers in anticipation of refinements.

Context instantiation: The axioms of the basis context, that allow future triggers to be added, has been improved so that ProB[8] can automatically create an instantiation.

A tool to automatically translate SCXML source models into UML-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 UML-B state-machine is subsequently translated into Event-B using the standard UML-B translation which provides variables to model the current state and guards and actions to model the state changes that transitions perform.

Figure 5 shows the UML-B model of the drone at refinement level 2 (equivalent to Figure 4 without the detail inside TAKEOFF). The structure of the state-machine is similar to the SCXML version with purple shading indicating the previously added states and light blue shading indicating the detail added at this refinement level. State invariants (properties that should hold while that state is active) are shown in TAKEOFF, FLY and BATTERYOK. Verification of these invariants is discussed in Section 9.

[8] ProB is an animator, constraint solver and model checker for the B-Method. https://www3.hhu.de/stups/prob

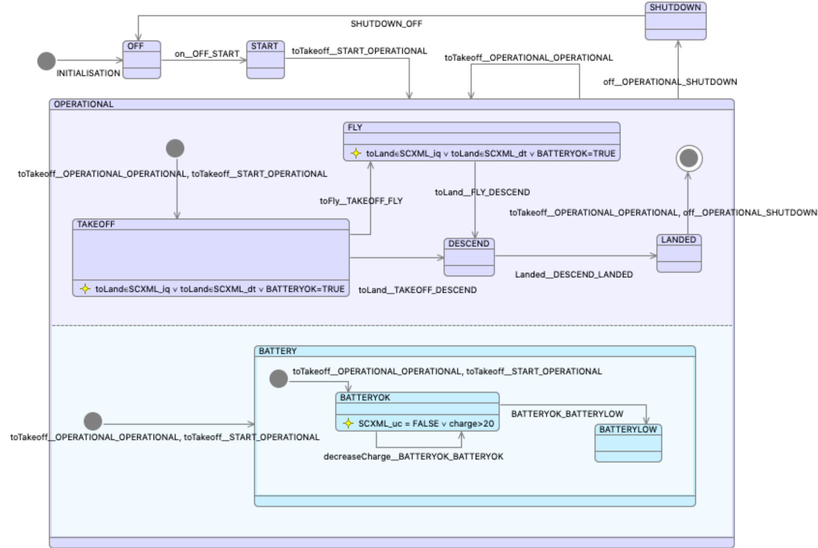

Fig. 5. Generated UML-B State-machine for drone refinement level 2.

8 Validation

One of the attractions of 'run to completion' style modelling languages such as SCXML is their execution semantics which provides a method for animating models to validate their behaviour. Our approach to SCXML refinement results in a single SCXML final model which can be animated using the existing SCXML animation tools. However, we would like to validate the developing UML-B model at intermediate refinement levels.

In previous work [20] we have developed a scenario-based approach to formal modelling using abstract scenarios to validate abstract models. The method is supported by a 'Scenario Checker' tool, based on the ProB model checker, that allows scenarios to be recorded and then replayed to check that important state has not changed since the original run of the scenario. The Scenario Checker supports the concept of a controller executing a process in response to changes in the environment which is similar to the run to completion concept addressed in our work here. Events may be annotated as *internal* to indicate that, when enabled, they should be fired automatically until none remain. Internal events may also be prioritised to give a simple representation of process order in the controller (even if it is left non-deterministic in the model). The user only has to select external events that trigger the controllers responses. Since our SCXML derived models already contain an implementation of run to completion the support provided by the Scenario Checker is sufficient to validate this behaviour. If desired, internal variables that represent the controllers processing (e.g., the variables that model the SCXML run to completion variables) can be annotated as *private* so that only the application state is checked during replay. To help

visualise the state of the model, the generated UML-B state-machine is animated during the scenario validation.

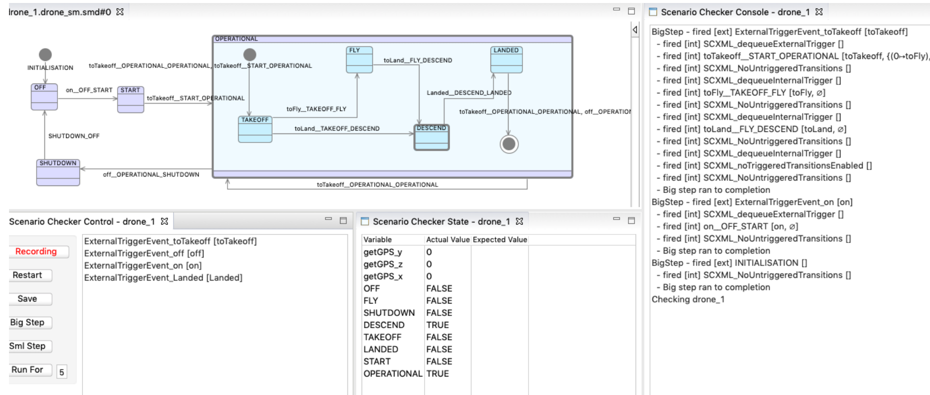

Fig. 6. Using Scenario Checker to validate behaviour of refinement 1- recording

Figure 6 shows a scenario being recorded. The main (top-left) editing view shows the state-machine being animated; the model is currently in the DESCEND state. The bottom left view is the scenario checker control panel where external events can be fired to start a run to completion. In our model, only the external-trigger-raising events (representing the environment) are enabled. The main button to be used is the *Big Step* button which fires the selected external event and then automatically fires internal events until non are enabled. The right hand view shows the scenario checker console, listing each big step and its run to completion in terms of internal events. The bottom centre view shows the state of the system at the end of the last run.

Figure 7 shows the recorded scenario being played back. In the control panel, external events are greyed out as they are being selected from the recording each time the *Big Step* button is pressed. The state view shows a discrepancy from when the scenario was recorded, and the state-machine is in the FLY state instead of the DESCEND state. Comparing the history in the console panels reveals that cause: an internal trigger, toLand, was non-deterministically raised during recording but not during playback. This is because the model allows for future raising of internal triggers in later refinements.

Figure 8 shows the scenario being played back at a later refinement where the raising of the toFly and toLand internal triggers has been defined. While the scenario checker allows us to animate that the main expected run to completion behaviour is possible, recall that, unless the transitions have all been finalised (i.e., no further refinement is permitted), other behaviours are possible due to the non-deterministic completion incorporated in case transition guards are later strengthened.

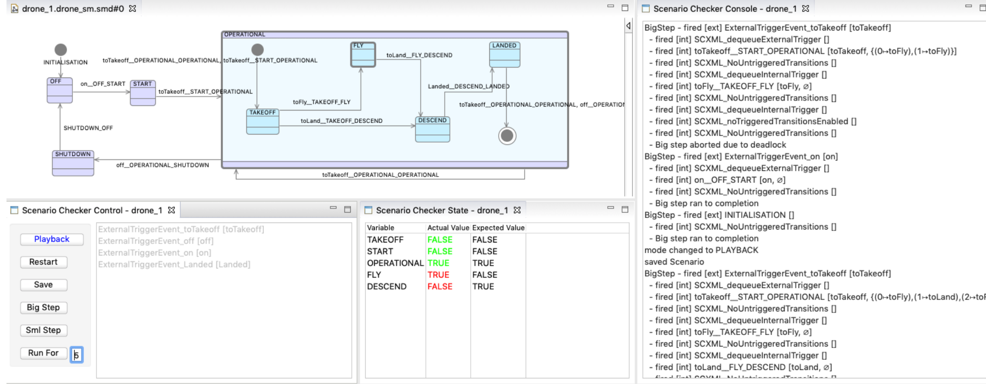

Fig. 7. Using Scenario Checker to validate behaviour of refinement 1 - playback

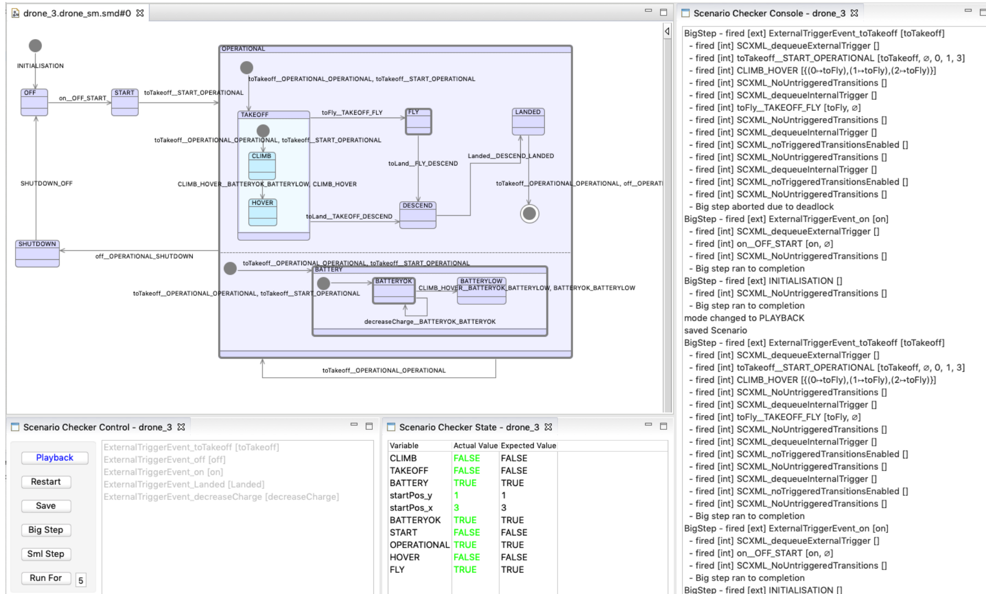

Fig. 8. Using Scenario Checker to validate behaviour of refinement 3 - playback

9 Verification of Safety Properties

In a state-chart model we naturally wish to verify properties of the form; a property P that is expected to hold true in a particular state S. Hence, all of the safety properties that we consider are captured as invariants of the form: S=TRUE ⇒ P, where the antecedent is implicit from the containment of P within S. There are two kinds of properties that we might want to verify in an SCXML state-chart; 1) properties concerning the values of auxiliary data maintained by the system and 2) constraints about the state of another parallel state-chart

region. SCXML models represent components that react to received triggers and cannot be perfectly synchronised with changes to the monitored properties. Hence, P may be temporarily violated until the system reacts by leaving the state S in which the property is expected to hold. To cater for this we express P in a modified form P' that allows time for the reaction to take place. There are two forms of reaction that can be used to exit S; a) an untriggered transition, or b) a transition that is triggered by an internally raised trigger. For a), the modified property P' becomes P $\vee$ *untriggered transitions are not complete*, and for b) P' becomes P $\vee$ *trigger* t *is in the internal queue or dequeued* (where t is the internal trigger raised when the violation of P is detected). Hence P is checked only in stable states that are reachable according to the run-to-completion semantics.

In this section we illustrate a typical example of the type of properties that we imagine could be verified in a reactive SCXML system. All of the proof obligations are automatically discharged for our example[9]. Since our models are strictly structured and proof obligations will always have this common form, we are optimistic that proofs will always discharge automatically. We model the safety property features at an early level of refinement where the models are relatively simple, so that the validity of verification conditions is clear. Detail is then added in later refinements which are proven (automatically) to preserve the previously verified safety properties. In our example, some auxiliary data is monitored by one state-chart region and while a parallel region refers to the state of the monitoring region. Hence the reaction consists of an un-triggered transition in the monitoring region which sends an internal trigger to the other region when it leaves the desired monitor state.

For our drone model, the safety property that we wish to verify is that the control system does not continue to take off or fly if the battery charge drops below a certain threshold (say 21%). By refinement level 1 we have developed the drone's state to the point where we distinguish the TAKEOFF and FLY states (Figure 3). In refinement level 2 we therefore introduce the battery charge monitoring function along with the associated safety properties. A parallel state-chart region, with sub-states BATTERYOK and BATTERYLOW, is added to the state OPERATIONAL (Figure 4). The BATTERYOK sub-state is used in the safety invariant of the TAKEOFF and FLY states. Thus we split the verification into two parts: a *type b* proof to show that the system reacts to the battery charge decreasing below 21% (an external event) by leaving the BATTERYOK sub-state, and a *type a* proof to show that when the system leaves the BATTERYOK state it subsequently (within the run to completion) leaves the FLY or TAKEOFF states. Both parts are described in more detail as follows.

System Reacts to the Low Battery Charge An external trigger indicates that the battery charge has dropped by 10% and this is used by a self transition to decrement the controllers data value for charge. The BATTERYOK state is supposed to indicate that the battery charge is ok (>20%) and to ensure that it does, we add a state invariant to this effect (charge>20). When charge decreases

[9] We use a strong prover configuration including AtelierB provers and SMT solvers

to 20 (or less), an untriggered transition immediately reacts by switching to the BATTERYLOW state. To ensure that this reaction is not bypassed by the non-determinism that we incorporated to allow for future refinement, we flag it as finalised at refinement level 2. Finalisation means that we cannot strengthen its guards in future refinements as is normally permitted, since its reaction is needed to ensure the invariant is preserved. If the user forgoes the finalization, the property would not be verifiable at that refinement level and it will need to be verified in later refinements. After translation to Event-B via UML-B the invariant in state BATTERYOK is

$$(\mathsf{BATTERYOK} = \mathsf{TRUE}) \Rightarrow (\mathsf{uc} = \mathsf{FALSE} \lor \mathsf{charge}{>}20)\ .$$

The only events that can break this invariant are ones that make the antecedent become true or the consequent become false and we deal with these as follows: The transitions that enter state OPERATIONAL and initialise the BATTERY region by entering BATTERYOK (hence making the antecedent become true) contain the guard that $\mathsf{charge}{>}50$ (since we do not allow the drone to take off unless the battery is well charged) and hence the invariant is satisfied. The self transition that decreases charge (and hence could potentially falsify the consequent) is guarded by $\mathsf{uc} = \mathsf{FALSE}$ since it is a triggered transition, and hence the disjunction in the consequent ensures it remains true. The completion event NoUntriggeredTransitionsEnabled of the basis machine resets $\mathsf{uc} = \mathsf{TRUE}$ to indicate completion of the cycle and hence could potentially break the invariant. However, finalising the transition BATTERYOK_BATTERYLOW (that leaves BATTERYOK when $\mathsf{charge}{>}20$ becomes false) means that the negation of its guard is added to the completion event by the translation. Since this transition fires when $\mathsf{BATTERYOK} = \mathsf{TRUE}$ (i.e., its source state) and $\mathsf{charge}{\leq}20$ the completion event is guarded by $\neg(\mathsf{BATTERYOK}{=}\ \mathsf{TRUE} \land \mathsf{charge}{\leq}20)$ which means that it does not fire when it could break the invariant (i.e., forcing the untriggered reaction to fire first).

System Subsequently Leaves the FLY or TAKEOFF States The safety property of the TAKEOFF and FLY states can now be simply stated as $\mathsf{BATTERYOK} = \mathsf{TRUE}$. However, since this relies on a particular internal trigger (toLand) to make the appropriate reaction, we also need to specify that trigger as an attribute of the invariant in the SCXML model. After translation to Event-B via UML-B the invariant in state TAKEOFF becomes

$$\begin{gathered}(\mathsf{TAKEOFF} = \mathsf{TRUE})\\ \Rightarrow\\ (\mathsf{toLand} \in \mathsf{content}(\mathsf{iQ}) \lor \mathsf{toLand}{\in}\mathsf{dt} \lor \mathsf{BATTERYOK}{=}\mathsf{TRUE}).\end{gathered}$$

The invariant for the FLY state is similar with a corresponding antecedent. The transitions that enter TAKEOFF (which make the antecedent true) simultaneously enter BATTERYOK ensure the consequent is true. The only transition that enters FLY (which makes the antecedent of the FLY invariant true) comes from the TAKEOFF state and hence the consequent is already true.

The transition that leaves BATTERYOK (making the last disjunct of the consequent false) raises the toLand trigger making the first disjunct true. Some transitions leave the superstates of BATTERYOK but these either simultaneously leave OPERATIONAL (the superstate of TAKEOFF and FLY), or re-enter BATTERYOK. The basis contains an event to dequeue the internal triggers which preserves the overall consequent because establishes the second conjunct as it falsifies the first (i.e., it removes toLand from the iQ but simultaneously adds it to dt). The only events that falsify the second conjunct are the transitions triggered by toLand which leave the TAKEOFF or FLY states making the antecedent false.

Hence, invariant properties that follow these suggested patterns are automatically proven due to simple logic about the changes in state.

10 Verification of Control Responses

A model that has been proven to satisfy some safety (e.g., invariant) properties, may still not behave in a useful way. Therefore, as well as verifying invariant properties, we would like to verify the system's liveness (e.g., responsive) properties. That is, we want to ensure that the controller responds to external triggers and makes appropriate modifications to the system variables. These kind of liveness properties are difficult to prove via invariant preservation since they are temporal properties. In this section, we present our approach to verify the responsive properties of the system.

We first start with some generic properties of our generated Event-B model and the fairness assumption about the executions of the events. We then discuss the proof for termination of responses for external triggers in Section 10.1 and correctness of the responses for external triggers in Section 10.2.

Event Categories. In our Event-B model, the events can be separated into the following categories.

- *External events*: These events raise external triggers.
- *System events*: Events other than external events are called system events. They are the events by which the system responds to the external triggers (by creating different runs or simulation paths). These events can be seen in Figure 1 and are further categorised as follow.
  - *Future system events*: These events might raise internal triggers, i.e., futureTriggeredTransitionSet, and futureUntriggeredTransitionSet. The purpose of these events is to enable future introduction of more system details via refinement.
  - *The dequeue external trigger event* (i.e., dequeueExternalTriggered): These events dequeue the external trigger queue and will start a run.
  - *Internal system events*: These events belong to the internal behavior of the system to accommodate the runs for external triggers. These events can be seen in different groups as in Figure 1.

- * *The dequeue internal trigger event* (i.e., dequeueInternalTriggered): These events dequeue the internal triggers queue and will start a run.
- * *Triggered events*: The events corresponding to the event fired by (external or internal) triggers.
- * *The discard trigger event* (i.e., noTriggeredTransitionsEnabled): This event will move the system from the FIRING TRIGGERED state to the FIRING UNTRIGGERED state in the case where no triggered transition events are enabled.
- * *Untriggered events*: These are all the untriggered events in the systems.
- * *The completion event* (i.e., noUntriggeredTransitionsEnabled): This event will move the system from FIRING UNTRIGGERED state to the READY TO DE−QUEUE state in the case where no untriggered transition events are enabled.

Events from different categories have different roles in our reasoning about responsive properties.

We now present a theorem and a corollary related to (relative) deadlock-freeness properties for a different set of events.

Theorem 1 (Internal System Events are Relative Deadlock-Free). *Under the condition that* $iQ \neq \varnothing \vee uc = FALSE \vee dt \neq \varnothing$*, the internal system events are deadlock-free, i.e., there must be one internal system event enabled.*

Proof. This is based on the generation of our Event-B model (according to the basis structure as seen in Figure 1). In particular, we consider the different cases corresponding to the different "states", i.e., READY TO DE−QUEUE, FIRING TRIGGERED, and FIRING UNTRIGGERED.

- When the system is in the READY TO DE−QUEUE state, we know that $uc = TRUE \wedge dt = \varnothing$. According to our assumption, we then have $iQ \neq \varnothing$, hence dequeueInternalTriggered event is enabled.
- When the system is in the FIRING TRIGGERED state, either one of the triggered events is enabled or the noTriggeredTransitionsEnabled event is enabled.
- Similarly, when the system is in the FIRING UNTRIGGERED state, either one of the untriggered events is enabled or the noUntriggeredTransitionsEnabled event is enabled.

□

Corollary 1 (System Events are Relative Deadlock-Free). *Under the condition that* $eQ \neq \varnothing \vee iQ \neq \varnothing \vee uc = FALSE \vee dt \neq \varnothing$*, the system events are deadlock-free, i.e., there must be one system event enabled.*

Proof. This is based on the generation of our Event-B model (according to the basis structure as seen in Figure 1) and Theorem 1.

- In the case where $iQ \neq \varnothing \vee uc = FALSE \vee dt \neq \varnothing$, according to Theorem 1, one of the internal events is enabled.

– Otherwise, i.e., $\mathsf{iQ} = \varnothing \wedge \mathsf{uc} = \mathsf{TRUE} \wedge \mathsf{dt} = \varnothing$, according to our assumption, $\mathsf{eQ} \neq \varnothing$. In this case, the dequeueExternalTriggered event is enabled.

□

In order to reason about any liveness properties for an event system, we have to make assumptions about how often events will be fired. Here, we assume that all the system events are strongly fair.

Assumption 1 (Fair System Events) *We assume that all internal system event* e *are strongly fair, i.e.,* $SF(e)$ *; and the dequeue external trigger event is weakly fair, i.e.,* $WF(dequeueExternalTriggered)$ *.*

This assumption will ensure that the system will response no matter how often the external triggers are raised by the external events.

10.1 Termination of Responses for External Triggers

We first define the notion of event convergence and event anticipation.

Definition 1 (Event Convergence). *A set of events is said to be convergent if they all* decrease *a variant according to some well-founded order.*

Definition 2 (Event Anticipation). *Given a set of convergent events with respect to a variant, another set of events is anticipated with respect to the same variant if they* do not increase *the variant.*

Note that the anticipated events augment the set of convergent events and respect the variant used to prove the convergence property.

We start first by stating the main theorem about termination of responses for external triggers: it is always the case that the system will comeback to the READY TO DE−QUEUE state and $\mathsf{iQ} = \varnothing$, i.e., the system is ready to dequeue an external trigger (if any). This is stated as the following theorem.

Theorem 2 (Termination of Internal System Events). *Given that the internal events are convergent and the external events are anticipated, the system's internal queue is always eventually empty and the system transitions to the* Ready to De-queue *state, i.e.,*

$$GF(iQ = \varnothing \wedge uc = TRUE \wedge dt = \varnothing) \ .$$

Proof. Assuming that the properties is not satisfied, i.e., eventually, it is always the case that $\mathsf{iQ} \neq \varnothing \vee \mathsf{uc} = \mathsf{FALSE} \vee \mathsf{dt} \neq \varnothing$. This can be formalised as follows.

$$\mathsf{FG}(\mathsf{iQ} \neq \varnothing \vee \mathsf{uc} = \mathsf{FALSE} \vee \mathsf{dt} \neq \varnothing) \ .$$

Observe that in the states satisfying this condition, the dequeueExternalTriggered event is disabled. Furthermore, according to Theorem 1, the internal events will always be deadlock-free, and as a result, at least one of the internal event is enabled infinitely often, hence under the Assumption 1, this event occure infinitely often. According to Definition 1, the variant will be decreased infinitely

which violate the condition that the variant is defined on an well-founded order. Here, the external events are anticipated to ensure that the variant does not increase, hence does not interfere with the convergence of the internal events.

□

Note that Theorem 2 relies on convergence of internal events and anticipation of external events, which we will prove later.

Theorem 3 (Responsiveness to External Triggers). *If an external trigger is raised, then eventually, it will be dequeued.*

$$\mathsf{G}([\mathsf{externalTrigger.t}] \Rightarrow \mathsf{F}([\mathsf{dequeueExternalTriggered.t}]))\ ,$$

where we use externalTrigger.t *(resp.* [dequeueExternalTriggered.t]*) to denote the occurrence of* externalTrigger *(resp.* dequeueExternalTriggered*) with parameter* t*.*

Proof. Assuming that [externalTrigger.t] hence $\mathsf{t} \in \mathsf{content}(\mathsf{eQ})$, i.e., $\mathsf{eQ} \neq \varnothing$. According to Theorem 2, we have that dequeueExternalTriggered is enabled infinitely often. Since dequeueExternalTriggered is weakly-fair (Assumption 1), it is taken infinitely. We have two cases.

- If $\mathsf{t} = \mathsf{head}(\mathsf{eQ})$, it means we have [dequeueExternalTriggered.t]
- If $\mathsf{t} \neq \mathsf{head}(\mathsf{eQ})$, an occurrence of dequeueExternalTriggered will move t closer to be the head of the external queue eQ and eventually it will become the head of the queue and subsequently be processed eventually.

□

Proof of Convergence and Anticipation The responsiveness to external triggers presented in Theorem 3 relies on Theorem 2, which in turn relies on the proof of convergence for internal system events and anticipation for external events. These proof will need to be done for each individual SCXML state-chart as they do not hold a priori. We present a systematic approach to reason about the proof of convergence and anticipation relying on lexicographic order as follow.

A variant in Event-B can be a natural number (bounded bellow by 0) or a finite set (bounded below by the empty set $\varnothing$). Moreover, for a set of events, the variants $V_1, V_2, ...$ are combined into a lexicographic variant, i.e., $(V_1, V_2, ...)$ with V_1 has a higher priority than V_2, etc. An event is said to decrease this lexicographic variant if it either decreases V_1 or if it keeps V_1 the same and decreases V_2, so on and so forth. Lexicographic variants are supported in the latest Rodin (version 3.5).

Our generic approach to construct a lexicographic variant is according to the following order and the rule for each variant.

1. $V_{externalTrigger} = \mathsf{dt} \cap \mathsf{ExternalTrigger}$. This variant is used to prove the convergence for any externally triggered events (i.e., triggered event by some external trigger). These event remove the external trigger from dt hence "decrease" dt to the empty set.

2. Variants based on the state machine to prove the convergence of internally triggered events and untriggered events. This depends on the SCXML diagram and we will illustrate this on the example later.
3. $V_{dequeueInternalTriggered} = \mathsf{length}(\mathsf{iQ})$. This variant is used to prove the convergence of the dequeueInternalTriggered event. Since this event remove the head of the iQ, it decreases the length of the iQ trivially. While other events might increase iQ, by raising new internal triggers. However, these events should have been proved to converge using higher-priority variants.
4. $V_{noTriggeredTransitionsEnabled} = \mathsf{dt}$. This variant is used to prove the convergence of the noTriggeredTransitionsEnabled event. The event discard the trigger in dt hence "decrease" dt to the empty set.
5. $V_{noUntriggeredTranstitionsEnabled} = \{\mathsf{uc}, \mathsf{TRUE}\}$. This variant is used to prove the convergence of the noUntriggeredTransitionsEnabled event. The event changes uc flag from FALSE to TRUE, hence "decrease" the variant from $\{\mathsf{FALSE}, \mathsf{TRUE}\}$ to $\{\mathsf{TRUE}\}$.

Note that except for the variant related to the internally triggered events and untriggered events, i.e., (2), all other variants, i.e., $V_{externalTrigger}$, $V_{dequeueInternalTriggered}$, $V_{noTriggeredTransitionsEnabled}$, and $V_{noUntriggeredTranstitionsEnabled}$ are generic according to the underlying run-to-completion semantics.

The external events are anticipated according to the above variants trivially since they only modify the external queue eQ. Note that we do not attempt to prove the convergence of any future events here. Instead, we assume that these future events will be proved to be convergent later. While these future events raise new internal triggers (hence will increase variant $V_{dequeueInternalTriggered}$, ultimately, they will be converted to internal triggered and untriggered events. These events will need to be proved to be convergent by state-machine based variants which has higher priority than $V_{dequeueInternalTriggered}$. At the moment, we do not have the the notation of anticipation for a subset of a lexicographic variant. This can be introduced into Event-B in the future.

We now discuss the specific variants for the Drone example based on the actual state-chart as showed in Figure 4. The variants are for the internally triggered events and untriggered events. The lexicographic order of the variant use to prove the convergence of the events depending (1) on the *nested structure* of the state-chart and (2) on the *order of the transitions* with the same state-chart. For example, the variant for proving the convergence for the transition from SHUTDOWN to OFF will have a higher priority than the one for the transition from TAKEOFF to FLY, and this variant subsequently has higher priority than the one for the transition from CLIMB to HOVER. Furthermore, the variant for proving the convergence for the transition from TAKEOFF to FLY has higher priority than the one for the transition from FLY to DESCEND.

The translation of SCXML state-chart into UML-B/Event-B represents each state by a Boolean variable, TRUE if the system in that state and FALSE otherwise. As a result, the variant for proving the convergence of an event going out of a state S can be $\{\mathsf{S}, \mathsf{FALSE}\}$: the transition "decreases" the value of the variant from $\{\mathsf{TRUE}, \mathsf{FALSE}\}$ to FALSE.

Based on the above analysis, the variants that we used for proving the convergence of the internally triggered events and untriggered events in the Drone example are.

- $\{\mathsf{SHUTDOWN}, \mathsf{FALSE}\}$
- $\{\mathsf{TAKEOFF}, \mathsf{FALSE}\}$
- $\{\mathsf{FLY}, \mathsf{FALSE}\}$
- $\{\mathsf{BATTERYOK}, \mathsf{FALSE}\}$
- $\{\mathsf{CLIMB}, \mathsf{FALSE}\}$
- $\{\mathsf{CLIMB2}, \mathsf{FALSE}\}$

This variant proof are available in the Rodin archive at `https://tinyurl.com/ISSE-Drone`.

10.2 Correct Responses to External Triggers

In the previous section, we illustrate the reasoning about the responsiveness of the system to external triggers. However, we also need to prove that the response to the external triggers is correct. In our conference paper [17], we illustrate the use of ProB model checker to reason about such a property. Here, we show how we can prove such a property.

Once again, we assume that the system events are strongly fair as in Assumption 1. In general, our correct-response properties will have the following form:

$$\mathsf{G}([\mathsf{external_trigger_event}] \Rightarrow \mathsf{F}\{\mathsf{predicate}\}) ,$$

where the predicate concerns variables v that the system maintains, and may refer to old values $\mathsf{old}(\mathsf{v})$ that existed when the external trigger occurred. The translator generates a separate 'branch' refinement for each LTL property to be verified. In this special refinement, history variables are added to record the value at the state when the external trigger occurs, of any variables that are referenced as 'old' values.

We illustrate the method with an example of a temporal property that we expect to hold in the drone SCXML system. The liveness property that we wish to verify is that, after an external trigger event $\mathsf{decreaseCharge}$, the battery charge value should decrease in value, i.e.,

$$\mathsf{G}\ ([\mathsf{ExternalTriggerEvent_decreaseCharge}] \Rightarrow \mathsf{F}\ \{\mathsf{charge} < \mathsf{old}(\mathsf{charge})\})\ .$$

As discussed in [17], this above property is too strong and does not hold for the SCXML drone model. We have to weaken the property to state that the expected behaviour is only achieved if the external trigger is raised at the right time, specified as $\{\mathsf{BATTERYOK}{=}\mathsf{TRUE} \land \mathsf{charge}{>}20\}$, and there are no external conflict triggers, here off, in processing or that has been raised, i.e., $\mathsf{off} \notin \mathsf{dt} \cup \mathsf{content}\{\mathsf{eQ}\}$. The property can be formalised as follows

$$\mathsf{G}([\mathsf{ExternalTriggerEvent_decreaseCharge}] \land \{\mathsf{BATTERYOK}{=}\mathsf{TRUE} \land \mathsf{charge}{>}20 \land \mathsf{off}{\notin}\mathsf{dt} \land \mathsf{off} \notin \mathsf{content}(\mathsf{eQ})\} \Rightarrow \mathsf{F}\,\{\mathsf{charge} < \mathsf{old}(\mathsf{charge})\}\})\ .$$

In order to prove the above property, we first recall the definition of **unless** property [3,9].

Definition 3 (Unless Properties). *An* unless property *of the following form*

$$P \textbf{ unless } Q$$

means that if P holds then it will hold continuously unless Q hold.

We restate the *Unless rule* (Theorem 1 in [9]) here.

Theorem 4. *An event system satisfies the unless property P **unless** Q, if for every event, if it starts in a state satisfying $P \land \neg Q$, it will reach a state satisfying $P \lor Q$.*

Coming back to our example, we first prove that the Event-B model satisfies the following important **unless** property.

Theorem 5. *The drone system satisfies the following **unless** property.*

$$\begin{gathered} BATTERYOK = TRUE \land charge > 20 \land decreaseCharge \in content(eQ) \land off \notin dt \land \\ (\forall i \in dom(eQ) \,.\, eQ(i) = off \Rightarrow (\exists j \in dom(eQ) \,.\, eQ(j) = decreaseCharge \land j < i)) \\ \textbf{unless} \\ BATTERYOK = TRUE \land charge > 20 \land dt = \{decreaseCharge\} \end{gathered}$$

Proof (Sketch). The proof relies on Theorem 4, i.e., reasoning per event. The encoding of the proof obligations are available from the Rodin archive. We informally explain why this property holds for different class of events below.

- *External events*: The external events raise a new external trigger and append the new trigger to eQ. Given that decreaseCharge is already in eQ, even if the new trigger is off, this trigger cannot over take decreaseCharge in the queue, i.e., it is always behind decreaseCharge. These external events therefore maintains the left-hand side of the **unless** property.
- *Dequeue external trigger*: If the dequeued external trigger is decreaseCharge, we will have $\mathsf{dt} = \{\mathsf{decreaseCharge}\}$, hence we establish the right-hand side of the unless property. If it is not decreaseCharge (that is descreaseCharge is still in the queue), the dequeued trigger also cannot be off (as any off trigger in eQ has to be behind a decreaseCharge trigger). As a result, the condition that off is behind decreaseCharge in the remaining queue is maintained and off cannot be in dt after the event execution.
- *Internal system events*: For the internal systems events, we separate them into those that are outside the BATTERY state and those that are inside of the BATTERY state.
  - For those that are outside the BATTERY state, they maintain the left handside of the unless property trivially (by leaving external queue unchanged and does not alter the relevant state, i.e., BATTERY or the charge).

- For the self-transition which is triggered by decreaseCharge inside the BATTERY state, the proof of the unless property is trivial, since we assume the negation of the right-hand side of the unless property, including that that decreaseCharge is not in dt. For the transition from BATTERYOK to BATTERYLOW, the proof of the unless property is also trivial, since we assume the left-hand side of the unless property, including that charge > 20.

□

Theorem 5 means that the system will continuously satisfy the following conditions:

– in the BATTERYOK state ,
– charge is more than 20 ,
– decreaseCharge is in the external queue eQ .
– off is not in dt .
– if off is in the external queue eQ then it is behind a decreaseCharge trigger .

unless it reaches a state satisfying the following conditions:

– in the BATTERYOK state ,
– charge is more than 20 ,
– decreaseCharge is in dt.

Coming back to the proof for our correct-response property and assume that the system is at the right time and the [ExternalTriggerEvent_decreaseCharge] event happens. Notice that at that particular moment, the left-hand side of the progress property in Theorem 5 is also satisfied. According to Theorem 3, eventually, [dequeueExternalTriggered.decreaseCharge] is fired (and dt $= \{$decreaseCharge$\}$), i.e., decreaseCharge is dequeued from the eQ into dt. And at that time, according to Theorem 5, we also have BATTERYOK $=$ TRUE and charge > 20. That ensures the triggered transition event for decreaseCharge is enabled (and it is the only internal event enabled) and will be eventually taken, hence decrease the charge's value accordingly.

11 Conclusion

Reactive State-charts are useful and widely used by engineers for modelling the design of control systems. Event-B provides an effective language for formally verifying properties via incremental refinements. However, it is not straightforward to apply the latter to the former. We have demonstrated a technique for introducing refinement of reactive State-charts that can be translated to Event-B for verification. Invariant properties about the expected coordination of states can be added and are interpreted with additional allowance for the reactions to take place. That is, they hold only after the reaction has taken place. Such invariants prove automatically with the existing Rodin theorem provers. We also

demonstrate a complementary process for verifying expected reactions to environmental triggers that uses the LTL model checker. We show how liveness can be verified to show that the 'run' converges to completion. i.e., transition loops and raised internal triggers do not introduce endless live-lock, but eventually terminate to allow the next external trigger to be consumed. This convergence proof uses lexicographic variants which (at our suggestion) have recently been added to the Rodin toolset for Event-B. These verifications do not validate that the model behaviour is useful. For this, the SCXML model should be animated so that its behaviour can be observed by a domain expert. Elsewhere [20] we have developed a 'Scenario Checker' tool and methods for animating pre-defined domain specific scenarios at various levels of abstraction. We demonstrate the use of this tool for automatically executing the run to completion in order to validate that the expected behaviour is emerging and is useful.

In future work, we intend to formalise the semantics of our extended SCXML notation in order to define its notion of refinement and correspondence to Event-B.

All data supporting this study are openly available at https://tinyurl.com/ISSE-Drone

Acknowledgements Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

References

1. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press (2010)
2. Abrial, J.R., Butler, M., Hallerstede, S., Hoang, T., Mehta, F., Voisin, L.: Rodin: An open toolset for modelling and reasoning in Event-B. Software Tools for Technology Transfer **12**(6), 447–466 (Nov 2010)
3. Chandy, K.M., Misra, J.: Parallel program design - a foundation. Addison-Wesley (1989)
4. Eclipse Foundation: Sirius project website. https://eclipse.org/sirius/overview.html (Mar 2016)
5. Eugene Syriani, Vasco Sousa, L.L.: Structure and behavior preserving statecharts refinements. Science of Computer Programming **170**(15), 45–79 (Jan 2019), https://doi.org/10.1016/j.scico.2018.10.005
6. Harel, D.: Statecharts: A visual formalism for complex systems. Sci. Comput. Program. **8**(3), 231–274 (Jun 1987). https://doi.org/10.1016/0167-6423(87)90035-9
7. Hoang, T.S.: An introduction to the Event-B modelling method. In: Industrial Deployment of System Engineering Methods, pp. 211–236. Springer-Verlag (2013)
8. Hoang, T.S., Schneider, S., Treharne, H., Williams, D.: Foundations for using linear temporal logic in event-b refinement. Formal Aspects of Computing **28** (04 2016). https://doi.org/10.1007/s00165-016-0376-0
9. Hudon, S., Hoang, T.S., Ostroff, J.S.: The Unit-B method — refinement guided by progress concerns. Software and System Modeling **15**(4), 1091–1116 (Oct 2016), http://dx.doi.org/10.1007/s10270-015-0456-2
10. Kuppe, M.A., Lamport, L., Ricketts, D.: The TLA+ toolbox. In: Monahan, R., Prevosto, V., Proença, J. (eds.) Proceedings Fifth Workshop on Formal Integrated

Development Environment, F-IDE@FM 2019, Porto, Portugal, 7th October 2019. EPTCS, vol. 310, pp. 50–62 (2019). https://doi.org/10.4204/EPTCS.310.6, https://doi.org/10.4204/EPTCS.310.6
11. Lamport, L.: Proving the correctness of multiprocess programs. IEEE Transactions on Software Engineering SE-3 **2**, 125–143 (March 1977)
12. Lamport, L.: Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley (2002), http://research.microsoft.com/users/lamport/tla/book.html
13. Maraninchi, F.: The Argos language: Graphical representation of automata and description of reactive systems. In: In IEEE Workshop on Visual Languages (1991)
14. MATLAB: 9.7.0.1190202 (R2019b). The MathWorks Inc., Natick, Massachusetts
15. Morris, K., Snook, C.: Reconciling SCXML statechart representations and Event-B lower level semantics. In: HCCV - Workshop on High-Consequence Control Verification (2016), http://www.sandia.gov/hccv/_assets/documents/HCCV_2016_Morris.pdf
16. Morris, K., Snook, C., Hoang, T.S., Armstrong, R., Butler, M.: Refinement of statecharts with run-to-completion semantics. In: Artho, C., Ölveczky, P.C. (eds.) Formal Techniques for Safety-Critical Systems. pp. 121–138. Springer International Publishing, Cham (2019)
17. Morris, K., Snook, C., Hoang, T.S., Hulette, G., Armstrong, R., Butler, M.: Formal verification of run-to-completion style statecharts using event-b. In: In: Muccini H. et al. (eds) Software Architecture. ECSA 2020. Communications in Computer and Information Science. vol. 1269. Springer, Cham. (2020), https://doi.org/10.1007/978-3-030-59155-7_24
18. Morris, K., Snook, C., Hoang, T.S., Hulette, G., Armstrong, R., Butler, M.: Refinement and verification of responsive control systems. In: Raschke, A., Méry, D., Houdek, F. (eds.) Rigorous State-Based Methods. pp. 272–277. Springer International Publishing, Cham (2020)
19. Snook, C., Butler, M.: UML-B: Formal modeling and design aided by UML. ACM Trans. Softw. Eng. Methodol. **15**(1), 92–122 (Jan 2006). https://doi.org/10.1145/1125808.1125811
20. Snook, C., Hoang, T.S., Dghaym, D., Fathabadi, A.S., Butler, M.: Domain-specific scenarios for refinement-based methods. (to be published in) Journal of Systems Architecture (2020)
21. W3C: State chart XML SCXML: State machine notation for control abstraction. http://www.w3.org/TR/scxml/ (Sep 2015)