SAND2020-7636C

Formal verification of run-to-completion style statecharts 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, Livermore, California, U.S.A.
[2] ECS, University of Southampton, Southampton, United Kingdom

Abstract. Although popular in industry, state-chart notations with 'run to completion' semantics 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. We introduce a notion of refinement into a 'run to completion' statechart 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 critical (e.g. safety) invariant properties can be verified by proof despite the reactive nature of the system. We also show how behavioural aspects of the system can be verified by testing the expected reactions using a temporal logic model checking approach.

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

1 Introduction

Statecharts provide a graphical language, generalized from state machines, that is popular with engineers. Variants appear in Matlab Simulink/Stateflow [11] and the Ansys tools. Particularly attractive is providing accessibility to abstraction/refinement via Rodin/Event-B which has an intuitive metaphor in the Statechart semantics [12,13]. 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 preservation is key. In our version of Statechart 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.

Many incompatible definitions of refinement have been posed by others [4,10] 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 across refinement requires only a few restrictions to the original [5] Statecharts (e.g. transitions cannot cross containment boundaries arbitrarily), but still allows for both parallel and hierarchical composition.

The work we will present here includes three refinement rules.

1. *Rule A:* Guard conditions on a transition can be strengthened; 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.

Via the translation explained in Section 5, these rules rely on the usual Event-B proof obligations to ensure that they do indeed yield refinements in the Event-B semantics. 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 of [9]. 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.

Although the autonomous drone example in this paper is based on the example described in [4], 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 [4] "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 (more details are in Section 4). 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.

Section 2 provides background material. Section 3 discusses the Statechart concept of 'run to completion' and how it can be specified in Event-B. Section 4 introduces our example case study; a drone. Section 5 gives an outline of our translation from State-Chart XML (SCXML) to Event-B. Section 6 illustrates our approach to verifying safety invariant properties. Section 7 illustrates our approach to verifying control responses, and Section 8 concludes.

2 Background

2.1 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 [16]. SCXML follows a 'run to completion' semantics, where trigger events[3] may be needed to enable transitions. Trigger events are queued when they are raised, and then one is de-queued and consumed by firing all the transitions that it enables, followed by any (un-triggered) transitions that then become enabled due to the change of state caused by the initial transition firing. This is repeated until no transitions are enabled, and then the next trigger is de-queued and consumed. There are two kinds of triggers: internal triggers are raised by transitions and external triggers are raised by the environment (non-deterministicly for the purpose of our analysis). An external trigger may only be consumed when the internal trigger queue has been emptied. 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.

2.2 Event-B

Event-B [1,6] 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 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[4]) [2].

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

[4] An extensible toolkit which includes facilities for modelling, verifying the consistency of models using theorem proving and model checking techniques, and validating models with simulation-based approaches.

2.3 UML-B State-machines

UML-B [14] 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.

3 Run To Completion

The run to completion semantics is specified via an abstract basis that is extended by the model [12,13]. 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 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. Each refinement partitions these trigger sets further to introduce concrete triggers, leaving a new abstract set to represent the remaining triggers yet to be introduced. For clarity, we use sets to abstractly represent the trigger queues. This does not affect safety verification but forces us to introduce fairness assumptions regarding trigger consumption in order to verify liveness properties. It would be relatively straight forward to properly model the trigger queues which are an implementation of this fairness property.

Each of the transitions in the basis (see Figure 1) represents an abstract event of the basis machine 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

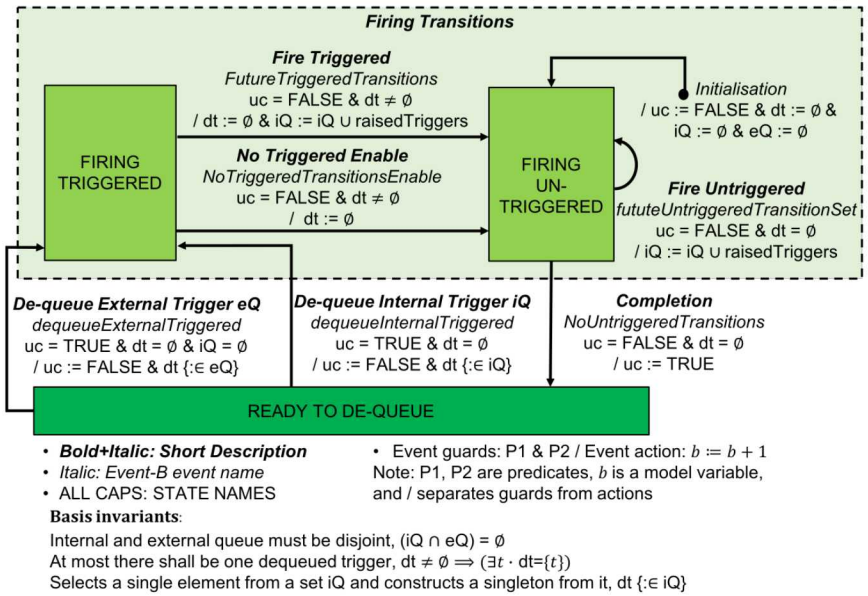

Fig. 1. Abstract representation of run to completion basis

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 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 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 un-triggered transitions are dealt with via the futureUntriggeredTransitionSet event. 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 the diagram). 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 futureTriggeredTransitions represents a combination of transitions that are triggered by the dequeued trigger, dt. 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 *fi-*

nalised (see Section 5 for more on finalisation). The SCXML translation tool will then automatically strengthen the guards of events NoTriggerTransitionEnable and futureUntriggeredTransitionSet, 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 NoTriggerTransitionEnable is strengthened by adding the conjunction of the negated guards of all transitions that can fire in parallel with the transition being finalised. Similarly the guard of futureUntriggeredTransitionSet 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.

4 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. [4]. 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 1, 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.

The first abstraction of the model shown in Figure 2 captures the basic functionality of the drone. 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[5]. The drone reacts to the off external trigger by shutting down and subsequently transitioning to the OFF state. Within the OPERATIONAL state the drone will transition to FLY, DESCEND or LANDED state after the internal trigger toFly, toLand or landed is raised, respectively. In this abstraction, 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. It

[5] Transitions in Figures 2–3 are labeled with trigger names (e.g. toTakeoff, toFly) not with event names as it is in UML-B.

should be noted that this abstraction of the drone model includes a transition from TAKEOFF to DESCEND (dashed transition in Figure 2). This allows for the drone to respond to a toLand trigger if it encounters some problems while in the TAKEOFF state. Syriani et al. [4] 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.

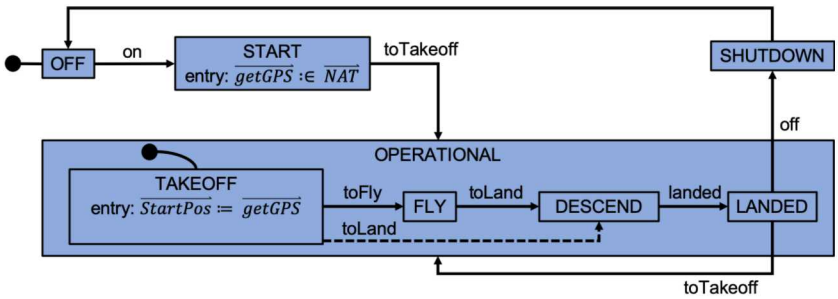

Fig. 2. State-chart of drone application. Abstract level including only generic behavior.

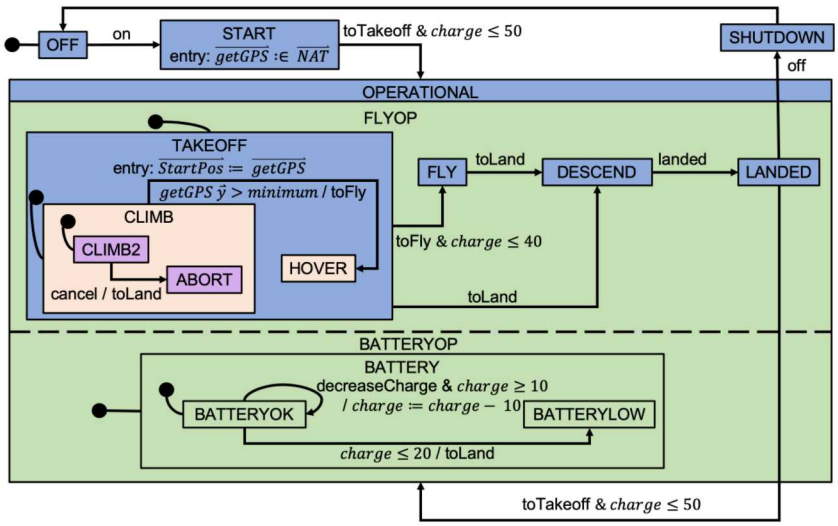

Fig. 3. State-chart of drone application. Refinement level introducing details for take off (shown in beige). Refinement level for battery consumption functionality (shown in green). Refinement level for descending capabilities, in case of emergency (shown in lilac).

Figure 3 shows three subsequent refinements to the drone model. The first refinement of the model is shown in beige, as we refine the parent 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. [4] 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 in the abstraction.

The second refinement, the details of which are shown in green in Figure 3, 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. [4]. The charge within the drone battery is control by the parallel BATTERYOP state. The functionality is modeled by introducing a new model variable, charge, which is decreased as a response to the internal trigger decreaseCharge. The aforementioned trigger, is raised non-deterministically by some unspecified internal functionality. Our state-chart semantics supports transition refinement, as such we are able to modified previously defined transitions. In particular, this type of refinement allow us to add guards and/or actions to previously defined transitions. The strengthening of guards, *Rule A*, or additional actions, *Rule B*, are expressed in term of new model variables that contribute implementation details to the model. To ensure the drone operates with enough battery power we strengthen the guards of transitions to the FLY and TAKEOFF states. As part of this design stage we introduce a requirement to constrain drone operation to a battery charge of at least 20% capacity. This can be expressed as

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

Figure 3 shows the third refinement of the drone model, with features added in lilac. At this stage we use *Rule C* to introduce additional implementation details to ensure that under special circumstances (e.g. sensing of adverse environment or unexpected battery dropped) the drone is able to circumvent flying and proceed to an emergency landing. The previously described requirement can be expressed as

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

To implement this new capability in the design the internal trigger cancel is introduced. The internal trigger cancel can be raised non-deterministically by some sensing capability, the details of which are not currently implemented. If the trigger is raised, the climbing process must be aborted and the drone descending sequence shall start. This refinement level is done differently from Syriani et al. [4], 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.

5 SCXML Translation to 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 (Section 3). 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.
- Secondly, all possible combinations of each set of transitions that can fire together are calculated and corresponding events are generated, at appropriate refinement levels, 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. If these transitions raise internal triggers, a guard, (e.g. $\{i1, i2, ...\} \subseteq raisedTrigger$, where $i1, i2, ...$ have been added to the internal triggers set), is introduced to define the raised triggers parameter. The subset allows 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, 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 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 [3] 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. Further details of the translation are given in [12,13].

6 Verification of Safety Properties

In a state-chart model we naturally wish to verify properties P that are expected to hold true in a particular state S. Hence, all of the safety properties that we consider are of the form: $S{=}TRUE \Rightarrow P$, 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. 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 2). 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 3). 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 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

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

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 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 uc = FALSE since it is a triggered transition, and hence the disjunction in the consequent ensures it remains true. The completion event NoUntriggeredTransitions of the basis machine resets uc = TRUE to indicate completion of the cycle and hence could potentially break the invariant. However, finalising the transition BATTERYOK_BATTERYLOW (that leaves BATTERYOK when 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 BATTERYOK = TRUE (i.e. its source state) and charge≤20 the completion event is guarded by ¬(BATTERYOK= TRUE ∧ charge≤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 BATTERYOK = 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

$$(\mathsf{TAKEOFF} = \mathsf{TRUE}) \Rightarrow (\mathsf{toLand} \in \mathsf{iQ} \vee \mathsf{toLand} \in \mathsf{dt} \vee \mathsf{BATTERYOK} = \mathsf{TRUE}).$$

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 always automatically proven due to simple logic about the changes in state.

7 Verification of Control Responses

A model that has been proven to satisfy some invariant (e.g. safety) properties, may still not behave in a useful way. Therefore, as well as verifying invariant properties, we would like to verify the system's responsiveness. That is, we want to ensure that the controller responds to external triggers to make appropriate modifications to the system variables. These kind of live responses are difficult to prove via invariant preservation since they are temporal properties. While Event-B refinements have also been shown to preserve some liveness properties under certain conditions [7], 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[6] model checker to verify it.

In general, our liveness 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 old(v) that existed when the external trigger occurred. To specify a liveness property to be verified, a special LTL element is added to the SCXML model with attributes, property (a string of the above form) and refinement (an integer indicating the refinement level at which the property should be verified). 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. A text file is automatically generated containing the LTL property to be checked. In this generated version, an assumption of strong fairness is added for all other events in the model. Without this assumption, the system may never achieve the expected response to a trigger. Therefore it corresponds to a requirement that the system can always make satisfactory progress and not become live locked. For simplicity we omit this assumption from the remaining examples.

$$\mathsf{SF}[\mathsf{e1}] \wedge \mathsf{SF}[\mathsf{e2}] \ldots \Rightarrow \mathsf{G}([\mathsf{external\_trigger\ \_event}] \Rightarrow \mathsf{F}[\mathsf{predicate}])$$

This property can be added into the ProB model checker LTL formula text field.

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

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

wish to verify is that, after an external trigger event decreaseCharge, the battery charge value should decrease in value.

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

However, we could not verify this property. The counter example traces that ProB provided gave us a better understanding of the reasons why. The property as stated is too strong (i.e. not true) for our model; there are additional conditions that need to be considered and added as part of the antecedent.

- Our model represented the trigger queues abstractly as sets which meant that the decreaseCharge trigger may never be dequeued. The standalone version of ProB allows strong fairness to be specified for particular parameter values but this does not work in the Rodin plug-in for ProB. In any case, a more accurate (concrete) representation of the queue fixes the problem and improves our model.
- The charge is not always decreased in response to the decreaseCharge trigger. The controller only monitors battery charge while in the BATTERYOK state and discards the trigger in other states. Also, the controller stops decreasing charge when it approaches 0. To cater for this we added a pre-condition $\mathsf{BATTERYOK} = \mathsf{TRUE} \wedge \mathsf{charge} \geq 10$ to the LTL property.
- Even if this pre-condition is true when the trigger is raised, another trigger (e.g. off) may already be in the queue and take the controller out of BATTERYOK before the decreaseCharge trigger is dequeued. Again we strengthen the pre-condition $\mathsf{off} \notin \mathsf{dt} \cup \mathsf{eQ}$ of the LTL expression to avoid this situation.

After making these changes the final form of the LTL property, which ProB was able to exhaustively check and confirm was as follows:

$$\mathsf{G}([\mathsf{ExternalTriggerEvent\_decreaseCharge}] \wedge \{\mathsf{BATTERYOK}{=}\mathsf{TRUE} \wedge \mathsf{charge}{\geq}10 \wedge \mathsf{off}{\notin}\mathsf{SCXML\_dt}{\cup}\mathsf{SCXML\_eq}\} \Rightarrow \mathsf{F}\,\{\mathsf{charge} < \mathsf{old}(\mathsf{charge})\})\ .$$

8 Conclusion

Reactive Statecharts 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 Statecharts 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. Another kind of liveness property that would be useful to verify is that the 'run' converges to completion.

I.e. transition loops and raised internal triggers do not introduce endless livelock, but eventually terminate to allow the next external trigger to be consumed. This could also be verified using the LTL model checker, however, in future work we will adopt the techniques suggested in [8] to verify liveness properties using the theorem provers.

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 [15] we have developed a 'Scenario Checker' tool and methods for animating pre-defined domain specific scenarios at various levels of abstract. In future work we will demonstrate the use of this tool for automatically executing the run to completion. In future work, we also 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 from the University of Southampton repository at https://doi.org/10.5258/SOTON/D1475

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. Eclipse Foundation: Sirius project website. https://eclipse.org/sirius/overview.html (Mar 2016)
4. 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
5. 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
6. Hoang, T.S.: An introduction to the Event-B modelling method. In: Industrial Deployment of System Engineering Methods, pp. 211–236. Springer-Verlag (2013)
7. 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
8. 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
9. Lamport, L.: Proving the correctness of multiprocess programs. IEEE Transactions on Software Engineering SE-3 **2**, 125–143 (March 1977)
10. Maraninchi, F.: The Argos language: Graphical representation of automata and description of reactive systems. In: In IEEE Workshop on Visual Languages (1991)
11. MATLAB: 9.7.0.1190202 (R2019b). The MathWorks Inc., Natick, Massachusetts

12. 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)
13. 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)
14. 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
15. 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)
16. W3C: State chart XML SCXML: State machine notation for control abstraction. http://www.w3.org/TR/scxml/ (Sep 2015)