Formal Language Semantics for Triggered Enable Statecharts with a Run-to-Completion Scheduling

Karla Vanessa Morris Wright[1][0000−0002−0146−3176], Thai Son Hoang[2][0000−0003−4095−0732], Colin Snook[2][0000−0002−0210−0983], and Michael Butler[2][0000−0003−4642−5373]

[1] Sandia National Laboratories, 7011 East Avenue Livermore, California 94550, USA
`knmorri@sandia.gov`[**]

[2] ECS, University of Southampton, Southampton SO17 1BJ, United Kingdom
`{cfs,t.s.hoang,m.j.butler}@soton.ac.uk`

Abstract. The increased complexity of high-consequence digital system designs with intricate interactions between numerous components has placed a greater need on ensuring that the design satisfies its intended requirements. This digital assurance can only come about through rigorous mathematical analysis of the design. This manuscript provides a detailed description of a formal language semantics that can be used for modeling and verification of systems. We use Event-B to build a formalised semantics that supports the construction of triggered enable statecharts with a run-to-completion scheduling. Rodin has previously been used to develop and analyse models using this semantics.

Keywords: Run-To-Completion · statecharts · Formal Semantics · SCXML · Event-B

1 Introduction

Motivation. Statechart notations, with event-triggered, '*run-to-completion*' semantics [5,6,10], provide an intuitive and visual modelling interface with which engineers can design and communicate. SCXML [15] is an example of such a statechart notation. Formal refinement, as in Event-B [1], provides a safe way to abstract important properties and introduce details in a manageable way. Hence we have proposed the introduction of refinement into SCXML in previous work [11,12,13]. These case studies presented examples of systems modeled using this proposed semantics. The focus then was on illustrating the incremental model construction process, the support of a divide and conquer approach for the verification of model properties leveraging refinement rules as defined in Event-B, and the translation from SCXML to Event-B. In order to define our proposals for refinement in SCXML more precisely, we first need to formally specify the semantics of SCXML itself. In this paper, we focus on the formalization of SCXML semantics using Event-B. SCXML can be viewed as

[**] Corresponding author, telephone: +1(925) 294-3287

2 superimposed behaviours: a) the underlying behaviour of how the statechart is structured and how it changes its active state and b) the run-to-completion schedule that dictates the order in which enabled transitions should be fired. We approach the formalisation by addressing these issues as separate Event-B models and then compose them using the CamilleX extension [9] to obtain the complete formalisation of the SCXML semantics.

Constribution. Our contributions are as follows.

- We provide a formalization of the run-to-completion semantics of SCXML statecharts in Event-B. Although we refer to SCXML as an example, we believe that it is typical of other similar statechart notations and we have not relied on specific SCXML definitions.
- The work illustrates how Event-B with its automatic proof obligation generators and theorem provers can be used to define formal semantics of notations. Here we explain the Event-B models and outline the more interesting proofs that required manual intervention.
- The work also illustrates how the composition feature of CamilleX can be used to structure such semantics definitions into more manageable sub-issues.

As far as we know, this is the first tool-supported semantics model for SCXML statecharts supporting features such as parallel regions.

Structure. In Section 2, we provide a high-level description of statecharts, and the run-to-completion execution model, as well as, the Event-B modeling method used to formalized the semantics of our triggered statechart modeling language. Section 3 introduces a turnstile running example which will be used in subsequent sections to illustrate the construction and analysis of a system using our formalized semantics. We develop the semantics model of the run-to-completion statecharts in three separate steps: (1) first, we model the semantics the untriggered statecharts (Section 4); and (2) we then specify the run-to-completion triggering mechanism (Section 5); and (3) we compose the two individual semantics models to construct the triggered statechart semantics (Section 6). A summary of our findings and concluding remarks are discussed in Section 7.

2 Background

2.1 Statecharts

Statecharts have been used for visual representation in the specification and design of complex systems for several decades. Different statechart formalizations exist depending on the features supported and specific characteristics of the modeled systems [5,6,4]. These diagrams provide a compact representation for modeling hierarchy, concurrency and communication in a systems design. Statecharts were originally developed to address unbounded modeling complexity in other state diagrams.

2.2 Run-to-Completion and SCXML

State Chart eXtensible Markup Language (SCXML) [3,15] is a general-purpose event-based statemachine language that combines concepts from Call Control eXtensible Markup Language (CCXML) and Harel State Tables. Harel State Tables are included in the Unified Modeling Language (UML). The concrete syntax for SCXML is based on XML. Hence, SCXML is an XML notation for UML style statemachines extended with an action language that is intended for call control features in voice applications. SCXML uses a run-to-completion semantics, also known as macro-step/micro-step semantics. This means that trigger events 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 non-deterministically by the environment. An external trigger may only be consumed when the internal trigger queue has been emptied. This means that an external trigger is only consumed when no transition can be taken without doing so.

2.3 Event-B

Event-B [1] is a formal method used to design and model software systems, of which certain properties must hold, such as safety properties. This method is useful in modelling safety-critical systems, using mathematical proofs to show consistency of models in adhering to its specification. Models consist of constructs known as machines and contexts. A *context* is the static part of a model, such as *carrier sets* (which are conceptually similar to types), *constants*, and *axioms*. Axioms are properties of carrier sets and constants which always hold. A context can *extend* one or more contexts by adding more carrier sets, constants and axioms. The following listing shows the context for a simple systems with two states.

```
context c
constants SA SB
sets states
axioms @def−states: partition(states, {SA}, {SB})
```

Machines describe the dynamic part of the model, that is, how the state of the model changes. The state is represented by the current values of the *variables*, which may change values as the state changes. *Invariants* are declared in the machine, stating properties of variables which should always be true, regardless of the state. *Events* in the machine describe state changes. Events can have *parameters* and *guards* (predicates on variables and event parameters); the guard must hold true for event execution. Each event has a set of *actions* which happen simultaneously, changing the values of the variables, and hence the state. Every machine has an initialisation event which sets initial variable values. The listing below shows a machine example with a single variable st.

```
machine m, sees c
variables st // state
invariants @typeof−st: st ∈ {SA, SB}
events
 event INITIALISATION
 then @init−st: st := SA
 end
 event t1
 where @source: st = SA  // event t1 guard
 then @target−st: st := SB // event t1 action
end
```

Contexts can be extended and machines can be refined to introduce details of the formal model gradually. Event extensions enable refinement of abstract events, the refined event will implicitly have all the parameters, guards and actions of the abstract event. We utilise context and event extension in this paper. An important set of proof obligations are invariant preservation. They are generated and required to be discharged to show that no event can potentially change the state to one which breaks any invariant, a potentially unsafe state. Event-B is supported by the extensible Rodin platform [2]. Extensions of Event-B and Rodin such as CamilleX [9] facilitate development of complex systems by allowing composition of existing models, and the reuse of modelling and proving efforts. This model composition capability enables machine inclusion with refinement and correct-by-construction proofs [7]. For example, when machine A includes another machine B, A will inherit all the invariants and discharged proofs of B (without the need for reproving). A can only modify the state variables of B by "synchronizing" with B's events. Direct modification of B's variables in A will introduce inconsistencies that result in unprovable proof obligations.

3 Turnstile Example

A turnstile is used to illustrate the construction of a system model under the developed semantics. Figure 1 shows the statechart diagram for the turnstile. The system has two sub-components that manage the operations performed by the gate and card reader in the turnstile. These components are represented by the *GATE* and *CARD_READER* parallel regions, respectively.

The model has two external triggers (*OnOff*, and *CardIn*), which are signals provided by the environment under which the system operates. In addition, there are internal triggers (*CardOk*, *Unblock*, *Block*, and *Reset*) that are raised by the components in the system. Transitions from source to target state are guarded by the specified conditions (i.e., [Unblock], *Unblock* trigger is in the queue). Actions associated with a specific transition are expressed as \raise Trigger, which result in adding the trigger to the queue. In the current model some of the internal triggers are raised non-deterministically (e.g. *CardOk*, *CardError*), as the details

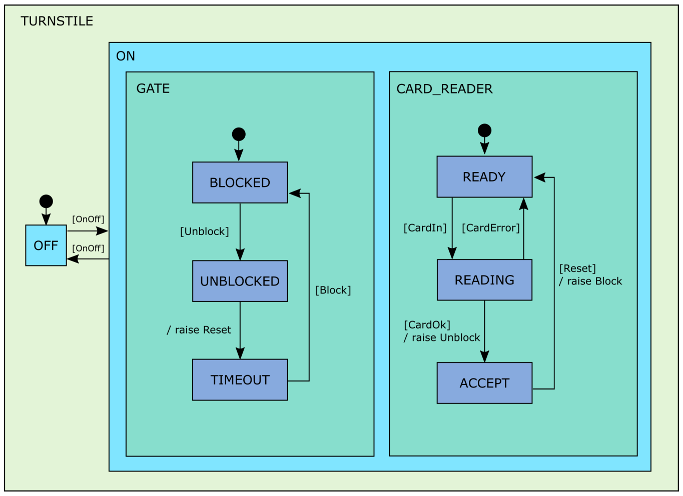

Fig. 1. Design model for a turnstile system

of the actual mechanisms responsible for the generation of the signal are not fully specified. Even in the presence of this non-determinism the system must satisfy certain requirements. Refinement of the abstract model presented in Figure 1 can be used to incorporate implementation details. For example, a nested statechart could be added to the *READING* state to specify the process by which the aforementioned triggers are raised. This manuscript focuses on formalizing the modeling language semantics for triggered statecharts of this form, and leaves the extensions required for refinement proofs to a future publication.

4 Formalization of Untriggered Statecharts

In this section, we formalise the syntactic elements and semantics of untriggered statecharts. This includes finding sufficient conditions for well-defined untriggered statecharts to guarantee consistent semantic behaviour of such models. The main ideas for our formalisation are (1) to use the Event-B *contexts* to capture the *syntactic elements* of the model with axioms ensuring that the model is well-defined (Section 4.1), and (2) to use the Event-B *machines* to capture the *semantics* of the models (Section 4.2).

4.1 Formalisation of the Untriggered Statechart Syntactic Elements

As the syntactic elements of untriggered statecharts are fairly complex, we develop them gradually, together with their well-definedness conditions, using Event-B context extension in the following steps.

1. Model the tree-structure of the states
2. Model the parallel regions
3. Model the transformations between states

Tree-structure States The structure of the states of a statechart is represented by the following constants; states (the set of all states), root (the implicit root state), container (the relationship between a child state and its parent container), leaves (the set of leaf states). These constants satisfy an axiom stating that they form a tree-shaped structure (here $\mapsto$ is the notation for specifying tuples).

$$\mathsf{states} \mapsto \mathsf{root} \mapsto \mathsf{container} \mapsto \mathsf{leaves} \in \mathsf{Tree}$$

Where, Tree is a constant, defined using transitive closure, that formalises the definition of tree-shaped structures. An important derived property of a tree-shaped structured (proved as a theorem) is needed to allow us to prove properties by induction.

Theorem 1 (Tree induction from root). *Consider a property P. If*

1. *The root satisfies P, and*
2. *for every non-root state s, if the container of s, i.e., container(s), satisfies P, then s also satisfies P,*

then every state in the tree satisfies P.

Example 1 (Turnstile Example. Tree-shaped Structure States). Formally, we have the following definitions.

```
// All states of the turnstile examples
partition(states, {root}, {OFF}, {ON}, {BLOCKED}, {UNBLOCKED}, {TIMEOUT},
  {READY}, {READING}, {ACCEPT})
// The container relationship between states
container = {OFF ↦ root, ON ↦ root, BLOCKED ↦ ON, UNBLOCKED ↦ ON,
  TIMEOUT ↦ ON, READY ↦ ON, READING ↦ ON, ACCEPT ↦ ON}
// leaf states
leaves = {BLOCKED, UNBLOCKED, TIMEOUT, READY, READING, ACCEPT, OFF}
```

The partition operator defines an enumerated set, states, where all the elements are explicitly given.

Regions Untriggered statecharts support the parallel composition of two or more nested statechart regions. That is a single state of a statechart may represent several sub components and associate with each component a corresponding region. In the turnsile example, the container state *ON* has two regions *GATE* and *CARD_READER*. We formalise the notion of regions as partitions of the set of non-root states by using the following axioms to constrain the constant regions.

Axiom 1 (Regions are subsets of states) *Each region is a subset of the statechart's states. (Here $\mathbb{P}$ is the notation for powerset.)*

$$\mathit{@region\_type: regions} \subseteq \mathbb{P}(\mathit{states})$$

Axiom 2 (Regions are disjoint) *Every pair of distinct regions does not share any states.*

$$@region\_disjoint: \forall r1, r2 \cdot r1 \in regions \wedge r2 \in regions \wedge r1 \neq r2 \Rightarrow r1 \cap r2 = \varnothing$$

Axiom 3 (Regions cover non-root states) *Every non-root state belongs to a region.*

$$@region\_complete: union(regions) = states \ \ \{root\}$$

Axiom 4 (Region has a unique container) *Every region has a unique container state. Here container[region] is the image of the relation container applying to region.*

$$@region\_same\_parent: \forall\ region \cdot region \in regions \Rightarrow \\ (\exists parent \cdot container[region] = \{parent\})$$

Example 2 (Turnstile Example. Regions). Formally, we have three regions as follows.

```
regions = { {ON, OFF}, // TURNSTILE region
  {BLOCKED, UNBLOCKED, TIMEOUT}, // GATE region
  {READY, READING, ACCEPT} // CARD_READER region
}
```

Note that states ON and OFF implicitly form a region without any sibling parallel region.

Transformations Unlike the common definitions of transitions, which map a source state to a target state, we define transformations, which give an hierarchical view of the set of all simultaneously enabled transitions of the system, from one *enabling* state configuration to the next configuration. There are different types of transformation including *forking* (starting from a state and ending in one or more states in different parallel regions), *joining* (starting from two or more states in different parallel regions and ending in a state), *parallel* (updating parallel regions at the same time), and any combination of these types. To model all transformation types, we formalise each transformation by three sets of states.

– enabling: A transformation is enabled (i.e., can be executed) if its (non-empty) set of enabling states are active.
– exiting: The (possibly empty) set of states that the transformation will exit upon execution.
– entering: The (possibly empty) set of states the transformation will enter upon execution.

Formally, these notions are formalised as constants as follows.

$$enabling \in transformations \rightarrow \mathbb{P}_1(states)$$
$$exiting \in transformations \rightarrow \mathbb{P}(states)$$
$$entering \in transformations \rightarrow \mathbb{P}(states)$$

Example 3 (Turnstile Example. Transformation). We give the enabling, exiting, entering and active states of the following example transformations.

- Consider the transformation from the BLOCKED state to the UNBLOCKED state, we call this BLOCKED_2_UNBLOCKED. We have

  ```
  enabling(BLOCKED_2_UNBLOCKED) = {BLOCKED}
  exiting(BLOCKED_2_UNBLOCKED) = {BLOCKED}
  entering(BLOCKED_2_UNBLOCKED) = {UNBLOCKED}
  ```

- Consider the transformation from the OFF state to the ON state, we call this OFF_2_ON. Notice that the transformation will take into account also the transition from the initial states within the two sub-statecharts (regions). As a result, we have

  ```
  enabling(OFF_2_ON) = {OFF}
  exiting(OFF_2_ON) = {OFF}
  entering(OFF_2_ON) = {ON, BLOCKED, READY}
  ```

- Consider the transformation from the ON state to the OFF state, we call this ON_2_OFF. Notice that the transformation will take into account the non-deterministic exit from the two sub-statecharts (regions). As a result, we have

  ```
  enabling(ON_2_OFF) = {ON}
  exiting(ON_2_OFF) = {ON, BLOCKED, UNBLOCKED, TIMEOUT, READY,
        READING, ACCEPT}
  entering(ON_2_OFF) = {OFF}
  ```

There are several additional constraints (well-definedness conditions) relating the enabling, exiting, and entering states for a transformation. We identified some of the constraints directly. For instance, the following axioms related to exiting states.

Axiom 5 (Exiting a contained region) *If the container of a region is an exiting state, there must be an exiting state within that region. Here container[r], is the image of the relation container applying to region r.*

```
@exiting−contained_region:
  ∀trf, s, r · trf ∈ transformations ∧ s ∈ exiting(trf) ∧ r ∈ regions ∧ container[r] = {s}
    ⇒ exiting(trf) ∩ r ≠ ∅
```

Axiom 6 (Exiting one or all states in a region) *If a region r has an exiting state s then either s is the unique exiting state or all the states in r are exiting states.*

```
@exiting−either_one_or_all_in_a_region:
  ∀trf, s, r · trf ∈ transformations ∧ s ∈ exiting(trf) ∧ r ∈ regions ∧ s ∈ r
    ⇒ exiting(trf) ∩ r = {s} ∨ r ⊆ exiting(trf)
```

The following axioms linking exiting and enabling states was "discovered" during the proof of the invariant preservation proof obligations (see Theorem 3).

Axiom 7 (Exiting a unique enabling state in a region) *Given a region r and an exiting state s in r, if r has states other than s, s must be the unique enabling state in r.*

```
@exiting−unique_enabling_state_in_a_region:
  $\forall trf, s, r \cdot trf \in transformations \land r \in regions \land exiting(trf) \cap r = \{s\} \land r \neq \{s\}$
    $\Rightarrow enabling(trf) \cap r = \{s\}$
```

Axiom 8 (Enabling state is the unique exiting state) *Given a region r with some exiting state, an enabling state s in r, s must the unique exiting state in r.*

```
@enabling−unique_exiting_state_in_a_region:
  $\forall trf, s, r \cdot trf \in transformations \land s \in enabling(trf) \land$
  $exiting(trf) \cap r \neq \varnothing \land r \in regions \land s \in r \Rightarrow exiting(trf) \cap r = \{s\}$
```

The following axiom relates entering with enabling and exiting states. It is also discovered during the proof of the invariant preservation proof obligations (see Theorem 3).

Axiom 9 (Transformation stays within a state) *If a transformation enters a region r but not the container of r (called c), then c is not an exiting state and there is an enabling state which is a descendant of c. Here, cl(container) denote the transitive closure of container relationship, hence the inverse of that (i.e., cl(container)∼ is the descendant relationship between states.*

```
@entering−stay_within_state:
  $\forall trf, r, c \cdot trf \in transformations \land r \in regions \land container[r] = \{c\} \land$
  $entering(trf) \cap r \neq \varnothing \land entering(trf) \cap container[r] = \varnothing$
    $\Rightarrow enabling(trf) \cap cl(container)^{\sim}[\{c\}] \neq \varnothing \land c \notin exiting(trf)$
```

4.2 Formalisation of the Untriggered Statechart Semantics

Given an untriggered statechart (characterized by the tree-shape structured states, the regions, and the transformation), the semantics of the statechart is characterized by the set of active states during its execution. For instance, consider the turnstile example in Figure 1, initially, the turnstile has one active state, namely OFF, i.e., the set of active states is {OFF}

– A transformation OFF_2_ON (from OFF to ON) will change the set of active states to {ON, BLOCKED, READY}.
– A transformation BLOCKED_2_UNBLOCKED will change the set of active states to {ON, UNBLOCKED, READY}.
– A transformation ON_2_OFF changes the set of active states to {OFF}.

We can now formalize the semantics of the untriggered statechart in a machine using a single variable active satisfying $active \subseteq states$. The system's functionality is encoded through one event called transformation that captures how the design transitions from one configuration to the next. Essentially variable active provides a discrete characterization of the information and event transformation represents the operation of the system under analysis.

```
event transformation
any trf where
 @typeof−trf: trf ∈ transformations
 @active−enabling: enabling(trf) ⊆ active
then
 @update−active: active := (active \ exiting(trf)) ∪ entering(trf)
end
```

Guard @active−enabling ensures that the chosen transformation trf is enabled and action @update−active first removes the trf's exiting states then adds trf's entering states. Notice that this action also allows a transformation to exit a state and re-enter that state.

An important aspect for the semantics of statecharts is that it can only transform amongst valid configurations. For example, we want to ensure that the turnstile statechart is never in the configuration where both ON and OFF states are active or in a state where BLOCKED is active, but ON is not active. The following constraints on active specify the valid configuration for a statechart, which we encode as 4 invariants for the machine.

Invariant 1 (Container active) *If a non-root state is active then its container is also active.*

$$\textit{@container\_active:} \forall s \cdot s \in \textit{active} \setminus \textit{root} \Rightarrow \textit{container}(s) \in \textit{active}$$

Invariant 2 (Content active) *If a container state is active then one of its sub-state must be active.*

$$\textit{@content\_active:} \forall s \cdot s \in \textit{ran}(\textit{container}) \land s \in \textit{active} \Rightarrow \\ \textit{container}\sim[\{s\}] \cap \textit{active} \neq \varnothing$$

where ran(container) is the range of the container relation

Invariant 3 (Unique active state within a region) *There can be at most one active state in a region.*

$$\textit{@active-region-unique:} \forall r, s \cdot r \in \textit{regions} \land s \in r \cap \textit{active} \Rightarrow r \cap \textit{active} \subseteq \{s\}$$

Invariant 4 (Parallel regions are inactive/active at the same time) *All parallel regions are inactive (hence active) at the same time.*

$$\textit{@active-region-parallel:} \forall r1, r2 \cdot r1 \in \textit{regions} \land r2 \in \textit{regions} \land \\ \textit{container}[r1] = \textit{container}[r2] \land r1 \cap \textit{active} = \varnothing \Rightarrow r2 \cap \textit{active} = \varnothing$$

As a consequence of Invariant 1, we can prove the following machine theorem. Note that all proofs related to this work were discharged semi-automatically within Rodin using the Proving Perspective. We present the general structure of a selected subset of these proofs.

Theorem 2 (Ancestors active). *If a state s is active then all ancestors of s are also active.*

$$\textit{@ancestor\_active:} \forall s \cdot s \in active \Rightarrow cl(container)[s] \subseteq active$$

Proof. The proof of the theorem relying on Invariant 1 and the inductive nature of transitive closure. We omit the details here.

We have to prove that event transformation maintains the invariants relying on the well-definedness constraints that we have put as axioms.

Preservation of invariant @active_container The proof obligation for ensuring that invariant @active_container is maintained by event @transformation after simplification can be stated as the following theorem.

Theorem 3 (Event transformation maintains @active_container). *Given a non-root state s such that either (1) s is active but non−exiting state, or (2) s is an entering state, then either (G1) container(s) is active but non-exiting state, or (G2) container(s) is an entering state.*

Proof. Since s is a non-root state, there exists a region r containing s (follows from Axiom 3 @region_complete). We continue the proof by considering Cases 1 and Case 2.

Case 1 (s is active but non-exiting state). We discharge (G1) by proving that (G1-1) container(s) is an active state, and (G1-2) container(s) is a non-exiting state.

– *Proof of (G1-1).* According to Invariant 1 (@container_active), since s is an active state, container(s) is an active state.
– *Proof of (G1-2).* We proceed by considering if r contains any exiting states.
  • *Case 1.1 (r does not contain any exiting states)* Using the contraposition of Axiom 5, we can conclude that container(s) (which is also the container of the region r) must not be an exiting state, which conclude the proof of (G1-2).
  • *Case 1.2 (r contains some exiting states)* The proof continue as follows.
    ∗ According to Axiom 8 (@enabling−unique_exiting_state_in_a_region), s cannot be an enabling state since s is a non-exiting state.
    ∗ We prove this by contradiction, i.e., assuming that container(s) is an exiting state.
      · According to Axiom 5 (@exiting−contained_region), there must be an exiting state in the region r since the state containing r (in this case container(s)) is an exiting state, let us call this state x.
      · According to Axiom 6 (@exiting−either_one_or_all_in_a_region, either x is the unique exiting state in r or all states in r are exiting states.
      · Since s is a non-exiting state, x must be r's unique exiting state.
      · From Axiom 7 (@exiting−unique_enabling_state_in_a_region), x is the unique enabling state in r.
      · According to guard @active−enabling of transformation, x (being an enabling state) must be an active state.

  · We now have two distinct active states x and s in r, which contradicts Invariant 3 (@active−region−unique).

Case 2 (s is an entering state). In this case, we proceed with the proof by assuming that (G2) does not hold (i.e., container(s) is not a container state) and prove (G1).

– According to Axiom 9, the transformation entering region r but not the container of r (i.e., container(s)), hence container(s) is not an exiting state and has a enabling descendant state, let us call this x.
– According to guard @active−enabling of transformation, x (being an enabling state) must be an active state.
– According to Theorem 2, container(s) (being an ancestor of x) must be an active state.
– We therefore have container(s) is active but non-exiting state (G1).

□

We omit the proof of the preservation of other Invariants 2, 3, and 4 due to limited space. The proofs is done within the Rodin tool utilising several axioms and the tree induction theorem (Theorem 1). Details can be found in [8].

5 Formalization of the Run-to-Completion Schedule

Similar to the previous section, the formalization is done using an Event-B context to capture the syntactic elements (Section 5.1) and an Event-B machine to model the semantics (Section 5.2).

5.1 Formalization of the Run-to-completion Syntactic Elements

To define run to completion execution we first specify the syntactic elements involved. Triggers are partitioned into either internal or external triggers. We define the internal and external trigger queues as sequences of internal and external triggers respectively. Sequences and their allowed operations (e.g. append trigger, head of sequence) are constructively defined via a series of theorems and axioms which are omitted here.

Figure 2 shows the state machine for the run-to-completion schedule. From the *Ready to de-queue* state, an internal or (if internal queue is empty) external trigger is de-queued from the corresponding queues and the system moves to the *Firing Triggered* state. A trigger step that consumes the dequeued trigger is then fired and the system moves to the *Firing Untriggered* state. From this state, untriggered steps can be fired repeatedly or the system moves back to the *Ready to de-queue* state. Both triggered and untriggered steps can raise more internal triggers, which will need to be handled in the future runs.

The context defining the syntactic elements for a run to completion is shown in Listing 1. An important notion for the run to completion schedule are steps. A triggered step is taken when an internal/external trigger is consumed and a non-deterministic number of untriggered steps may be taken subsequently. We will show how these steps relate to triggered/untriggered transitions later.

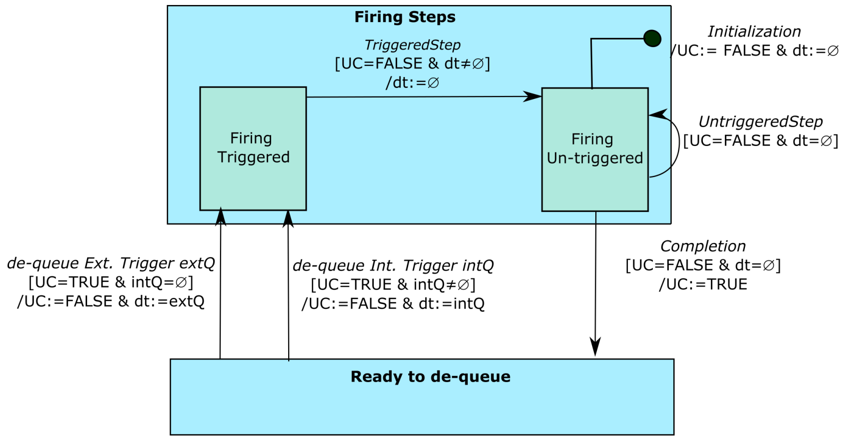

Fig. 2. State diagram for run to completion scheduling

```
context r2c_ctx extends r2c_c0_2_dequeue
sets Steps
constants InternalTriggers ExternalTriggers Triggers StepTrigger StepRaised
axioms
 @typeof_IT: InternalTriggers ⊆ InternalTriggerType
 @typeof_XT: ExternalTriggers ⊆ ExternalTriggerType
 @def_T: Triggers = InternalTriggers ∪ ExternalTriggers
 @typeof_StepTrigger: StepTrigger ∈ Steps ⇸ Triggers
 @typeof_StepRaised: StepRaised ∈ Steps → Seq(InternalTriggers)
end
```

Listing 1. Context for Run-to-Completion Semantics

Here StepTrigger (as a partial function) defines the required trigger for a Step (if any), and StepRaised defines the sequence of internal triggers that will be raised for a step (including an empty sequence). Steps that do not have any required trigger will be untriggered steps.

5.2 Formalization of the Run-to-Completion Semantics

Given the syntactic elements defined earlier, the machine defining the semantics for the run-to-completion schedule models the trigger queues in the system according to Figure 2. The dynamic status of the run-to-completion schedule is represented by the variables int_q, ext_q, dt (dequeue trigger), and completed with the following invariants.

```
@int_q: int_q ∈ Seq(InternalTriggers)
@ext_q: ext_q ∈ Seq(ExternalTriggers)
@dequeue_trigger: dt ∈ DeQueueType
@dequeue_triggerwd: dt ⊆ InternalTriggers ∪ ExternalTriggers
@firingTriggered: dt ≠ ∅ ⇒ completed=FALSE
```

Where int_q and ext_q represent the sequences of internal and external triggers that need to be handled, dt keeps track of the trigger that has been removed from the queues (de-queue) to be consumed by a *TriggeredStep*. Note that dt is a singleton set of trigger when the system is in the *Firing Triggered* state and empty otherwise (this is also the definition of DeQueueType). Finally, variable completed (denoted as UC in Figure 2) is TRUE indicates that the system is in the *Ready to de-queue* state.

```
event dequeueInternalTrigger
any trigger where
 @grd1: completed = TRUE
 @grd2: int_q ≠ ∅
 @grd3: trigger = Seq_head(int_q)
then
 @act1: dt := {trigger}
 @act2: int_q := Seq_tail(int_q)
 @act3: completed := FALSE
end
```

```
event dequeueExternalTrigger
any trigger where
  @grd1: completed = TRUE
  @grd2: ext_q ≠ ∅
  @grd3: trigger = Seq_head(ext_q)
  @grd4: int_q = ∅
then
  @act1: dt := {trigger}
  @act2: ext_q := Seq_tail(ext_q)
  @act3: completed := FALSE
end
```

Starting from the *Ready to de-queue* state (where completed = TRUE), the system will de-queue an internal/external trigger if there are any. Note that an external trigger is only de-queued when the internal queue is empty. The system then moves to the *Firing triggered* state (where completed = FALSE and dt is not empty).

The behaviour of the *TriggeredStep* and *UntriggeredStep* are formalised by the corresponding events as follows. Both events can raise a sequence of internal triggers which is concatenated to the the internal queue.

```
event triggeredStep
any Step where
  @grd1: Step ∈ dom(StepTrigger)
  @grd2: StepTrigger(Step) ∈ dt
then
  @act1: dt := dt \ {StepTrigger(Step)}
  @act2: int_q := Seq_concat(int_q ↦
      StepRaised(Step))
end
```

```
event untriggeredStep
any Step where
  @grd1: Step ∈ Steps \ dom(StepTrigger)
  @grd2: dt = ∅
then
 @act1: int_q := Seq_concat(int_q ↦
     StepRaised(Step))
end
```

While the internal triggers are raised through steps, external triggers can be raised non-deterministically by event raiseExternalTrigger and appended to the external queue. Note that at this stage (without the statemachine), there is a non-determinism between untriggeredStep and completion. When we combine the untriggered statechart and the run-to-completion schedule in the next section, we will distinguish the two cases.

```
event raiseExternalTrigger
any trigger where
  @grd1: trigger ∈ ExternalTriggers
then
  @act1: ext_q := Seq_append(ext_q ↦
      trigger)
end
```

```
event completion
where
  @grd1: dt = ∅
  @grd2: completed = FALSE
then
  @act1: completed := TRUE
end
```

The model of the run-to-completion schedule maintains its invariants straightforwardly (relying on operations of sequence manipulation).

6 Formalization of Triggered Statecharts

The triggered statecharts are a unified statechart model representation based on SCXML [15]. To formalize the complete semantics we compose the previous two models, of the untriggered statechart (Sections 4) and of the run-to-completion schedule (Section 5). The composition is performed by, using the inclusion mechanism built into the CamilleX extension [9] of the Rodin platform.

6.1 Triggered Statechart Syntactic Elements

Since a *triggered* statechart is a combination of an *untriggered* one and a run-to-completion schedule, the former is a syntactic extension of the latter (Listing 2). We introduce some syntactic elements *connecting* the sub-context together. Namely, relationships linking untriggered statechart transformations and run-to-completion steps, to form transitions.

```
context tstc_ctx extends r2c_ctx utstc_ctx
constants transitions discardSteps
axioms
 @typeof−transitions: transitions ∈ transformations ⤖ Steps \ discardSteps
 @discardSteps−Triggers: discardSteps ◁ StepTrigger ∈ discardSteps ⤖ Triggers
 @discardSteps−Raised: StepRaised[discardSteps] = { ∅ }
end
```

Listing 2. Context for Triggered Statechart

At the same time, when we combine the two sub-models, we need to ensure an important aspect of triggered statecharts which is their responsiveness. In particular, if a trigger (internal or external) is de-queued, but no enabled transformation that can consume the trigger, we need to ensure that the system can still progress. More precisely, the system will need to *discard* the problematic trigger in order to continue. Constants discardSteps are introduced to capture the special steps that discarding triggers.

Axiom @typeof−transitions specifies that transitions is a one-to-one correspondence between transformations and non-discarding steps (⤖ is the symbol for bijective functions). Axioms @discardSteps−Triggers and @discardSteps−Raised ensure that there is a discard step for every trigger and the discard steps do not raise any trigger ($\lhd$ is the symbol for domain restriction, [..] is a relational image, and $\{\varnothing\}$ the singleton set of the empty set).

6.2 Triggered Statechart Semantics

The semantics of a triggered statechart is captured by a machine that includes both the untriggered statechart and the run-to-completion schedule. We also use prefixing mechanism, e.g., **includes** r2c **as** r2c, so that all modelling elements of the included machine are prefixed accordingly.

The following events are *lifted* from the run-to-completion machine (unchanged): raiseExternalTrigger, dequeueExternalTrigger, dequeueInternalTrigger. Essentially, they only concern the management of trigger queues and do not relate to the statechart's status.

```
event triggeredTransition
synchronises r2c.triggeredStep
synchronises utstc.transformation
where
 @grd1: transitions(utstc_trf) =
     r2c_Step
end
```

```
event untriggeredTransition
synchronises r2c.untriggeredStep
synchronises utstc.transformation
where
 @grd1: transitions(utstc_trf) =
     r2c_Step
end
```

Events to model the transitions (triggeredTransition and untriggeredTransition) *synchronises* with the events from the untriggered statechart and the run-to-complete schedule. The additional guard of the events ensure that the chosen transformation (from the untriggered statechart) and the step (from the run-to-completion model) corresponds with each other.

As discussed before, we need to introduce the events to discard triggers discardTrigger (in the case where triggeredTransition is not available). This condition is formalised as discardTrigger's grd3. We also strengthen the guard of completion to ensure that the system will complete a run only when untriggeredTransition is not available (see completion's grd1).

```
event discardTriggered
any trigger
synchronises r2c.triggeredStep
where
  @grd1: r2c_Step ∈ discardSteps
  @grd2: trigger = StepTrigger(r2c_Step)
  @grd3: ∀trf · transitions(trf) ∈
     StepTrigger∼[{trigger}] ⇒ ¬enabling
     (trf) ⊆ utstc_active
end
```

```
event completion
synchronises r2c.completion
where
  @grd1:
    ∀ r2c_Step · r2c_Step ∈ Steps \
      dom(StepTrigger)
  ⇒
  ¬ (enabling(transitions∼(r2c_Step))
      ⊆ utstc_active)
 end
```

The use of composition as supported by the inclusion mechanism in Rodin results in the triggered statechart semantics inheriting the invariants from the submachines without the need to prove them as they are correct-by-construction.

7 Conclusions

We formalize the semantics of SCXML run-to-completion statecharts using Event-B. We formalize the syntactic elements using Event-B contexts and dynamic semantics using Event-B machines. The semantics model is built in a compositional fashion: the semantics of (untriggered) statecharts and run-to-completion schedule is developed independently and composed to create the SCXML statechart's semantics model. This approach allows us to reduce the complexity of the consistency reasoning by focusing on different parts of the models. The combined model inherits the consistency of the sub-models by construction.

In order to ensure the consistency of the semantics, several well-definedness conditions on the syntactic elements have been identified. They are encoded as axioms in the formal models. These well-definedness conditions can be used as the specification of a validation tool to ensure the consistency of SCXML models. Given the semantic models are consistent, any instantiation will inherit this consistency without the need for reproving. For instance, the model of the turnstile example can have consistency about the active states and the triggering mechanism. Often, these consistency checks make up the majority of the proof obligations, only a small number are related to the specific model properties.

We plan to extend this work to formalize the semantics of refinement as described in [11,12]. In particular, we will formalise the syntactic constraints that ensure consistent refinement of SCXML statecharts, proving the consistency of the refinement rules, e.g., in [13]. The consistency of the semantic models focuses on safety properties, expressed as invariants. Furthermore, we model some of the syntactic elements in our formal models at a fairly abstract level, e.g., the notion of enabling, exiting, entering states for transformation. Our abstract semantics supports the majority of typical statechart features such as transitions, hierarchical structure, clustering, concurency, start and stop states. We do not cover history or timeout mechanisms. Our composition approach means that our untriggered statechart semantics, which is common to most statechart notations, can be reused regardless of their triggering semantics (or lack thereof). E.g. UML-B [14] is untriggered. Furthermore, since SCXML is based on the widely used Harel statechart semantics [5], our run to completion semantics can also be used for such notations and where notations deviate in their run semantics (e.g. [4]), we at least encapsulate the extent of re-work required.

All data supporting this study are openly available from the University of Southampton repository at `https://doi.org/10.5258/SOTON/D2791`.

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. J-R. Abrial. *Modeling in Event-B: System and Software Engineering.* Cambridge University Press, 2010.

2. J-R Abrial, M. Butler, S. Hallerstede, T.S. Hoang, F. Mehta, and L. Voisin. Rodin: An open toolset for modelling and reasoning in Event-B. *Software Tools for Technology Transfer*, 12(6):447–466, 2010.
3. Jim Barnett. *Introduction to SCXML*, pages 81–107. Springer International Publishing, Cham, 2017.
4. Rik Eshuis. Reconciling statechart semantics. *Science of Computer Programming*, 74(3):65–99, January 2009.
5. David Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.
6. David Harel and Eran Gery. Executable object modeling with statecharts. In *Proceedings of IEEE 18th International Conference on Software Engineering*, pages 246–257. IEEE, 1996.
7. Thai Son Hoang, Dana Dghaym, Colin Snook, and Michael Butler. A composition mechanism for refinement-based methods. In *2017 22nd International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 100–109, 2017.
8. Thai Son Hoang, Colin Snook, Karla Morris, and Michael Butler. SCXML semantics model in Event-B, 2023. https://doi.org/10.5258/SOTON/D2791.
9. Thai Son Hoang, Colin F. Snook, Dana Dghaym, Asieh Salehi Fathabadi, and Michael J. Butler. Building an extensible textual framework for the Rodin platform. In Paolo Masci, Cinzia Bernardeschi, Pierluigi Graziani, Mario Koddenbrock, and Maurizio Palmieri, editors, *Software Engineering and Formal Methods. SEFM 2022 Collocated Workshops - AI4EA, F-IDE, CoSim-CPS, CIFMA, Berlin, Germany, September 26-30, 2022, Revised Selected Papers*, volume 13765 of *Lecture Notes in Computer Science*, pages 132–147. Springer, 2022.
10. Gerald Lüttgen, Michael von der Beeck, and Rance Cleaveland. A compositional approach to statecharts semantics. *SIGSOFT Softw. Eng. Notes*, 25(6):120–129, nov 2000.
11. Karla Morris, Colin F. Snook, Son Hoang, Robert C. Armstrong, and Michael J. Butler. Refinement of statecharts with run-to-completion semantics. In *International Workshop on Formal Techniques for Safety-Critical Systems*, 2018.
12. Karla Morris, Colin F. Snook, Son Hoang, Geoffrey C. Hulette, Robert C. Armstrong, and Michael J. Butler. Refinement and verification of responsive control systems. *Rigorous State-Based Methods*, 12071:272 – 277, 2020.
13. Karla Morris, Colin F. Snook, Thai Son Hoang, Geoffrey C. Hulette, Robert C. Armstrong, and Michael J. Butler. Formal verification and validation of run-to-completion style state charts using Event-B. *Innov. Syst. Softw. Eng.*, 18(4):523–541, 2022.
14. Colin F. Snook, Michael J. Butler, Thai Son Hoang, Asieh Salehi Fathabadi, and Dana Dghaym. Developing the UML-B modelling tools. In Paolo Masci, Cinzia Bernardeschi, Pierluigi Graziani, Mario Koddenbrock, and Maurizio Palmieri, editors, *Software Engineering and Formal Methods. SEFM 2022 Collocated Workshops - AI4EA, F-IDE, CoSim-CPS, CIFMA, Berlin, Germany, September 26-30, 2022, Revised Selected Papers*, volume 13765 of *Lecture Notes in Computer Science*, pages 181–188. Springer, 2022.
15. W3C. SCXML specification website. http://www.w3.org/TR/scxml/, September 2015.