Rodin Model supporting 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
knmorris@sandia.gov**
[2] ECS, University of Southampton, Southampton SO17 1BJ, United Kingdom
{cfs,t.s.hoang,m.j.butler}@soton.ac.uk

Abstract. This document contains the pretty-print model of the Rodin model supporting the paper titled "Formal Language Semantics for Triggered Enable Statecharts with a Run-to-Completion Scheduling".

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

1 Introduction

This PDF contains the listing of the Rodin model supporting the paper titled "Formal Language Semantics for Triggered Enable Statecharts with a Run-to-Completion Scheduling". The archive of the model is also available from the same repository. Instruction for installing the tool can be found in the repo README file. The models are fully proved within Rodin. We marked 76 proof obligations as "reviewed" in the triggered statecharts tstc machine that do not require proving of the combine model (as they are correct-by-construction).

Structure . Section 2 presents the formalisation of untriggered statecharts. Section 3 gives a formalisation of the run-to-completion schedule in Event-B. Finally, Section 4 describes the formalisation of triggered statecharts which is achieved by combining the model of untriggered statecharts and run-to-completion schedule.

2 Untriggered Statecharts

2.1 Formalisation of Transtive Closures

The formalisation of transitive closures is done in 3 contexts as follows.

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

```
/*
 * This context declares the set of states
 * @author htson
 */
context utstc_c0_0_states
sets
 STATE // The set of states
end
```

```
/*
 * This context define transitive closure of STATEs and its basic properties.
 */
context utstc_c0_1_closure_def
extends utstc_c0_0_states
constants
 cl
axioms
 /*
  * Definition for (irreflexive) transitive closure:
  * The transitive closure of r is the intersection of
  * all transitive relation containing r
  * The intersection is well-defined (def-cl/WD) since
  * There exists a transitive relation containing r which is
  * the Cartesian product of STATE and STATE
  */
 @def-cl: cl = (λ r · r ∈ STATE ↔ STATE | inter({p | r ⊆ p ∧ p;p ⊆ p}))

 /*
  * Typing information:
  * The transitive closure of a relation between STATEs
  * is also a relation between STATEs
  */
 theorem @typeof-cl: cl ∈ (STATE ↔ STATE) → (STATE ↔ STATE)

 // Transitive closure is ⊆-strengthening
 theorem @thm1-cl_strengthen: ∀ r· r ⊆ cl(r)

 // Transitive closure is transitive
  theorem @thm2-cl_transitivity: ∀ r·cl(r);cl(r) ⊆ cl(r)

  // Transitive closure is minimal
  theorem @thm3-cl_minimal: ∀ r·(∀ p· r⊆p ∧ p;p⊆p ⇒ cl(r)⊆p)

end
```

```
context utstc_c0_2_closure_prop
extends utstc_c0_1_closure_def
axioms
 /*
  * Copy of the theorems from c0_closure_def.
  * The extra properties in this context is derivable from these theorems
  */
 // Transitive closure is ⊆−strengthening
 theorem @thm1−cl_strengthen: ∀ r· r ⊆ cl(r)

 // Transitive closure is transitive
  theorem @thm2−cl_transitivity: ∀ r·cl(r);cl(r) ⊆ cl(r)

  // Transitive closure is minimal
  theorem @thm3−cl_minimal: ∀ r·(∀ p· r⊆p ∧ p;p⊆p ⇒ cl(r)⊆p)

 /*
  * Extra properties for transitive closure
  */
 // Transitive closure is ⊆−monotonic
 // Proof: instantiate @thm3−cl_minimal with r <− r and p <− cl(p)
 theorem @thm4−⊆_monotonic: ∀ r,p·r⊆p ⇒ cl(r) ⊆ cl(p)

 // Transitive closure right maximal
 // Proof: Consequence of @thm1−cl_strengthen and @thm2−cl_transitivity
  theorem @thm5−right_maximal: ∀ r·cl(r);r ⊆ cl(r)

  // Transitive closure left maximal
 // Proof: Consequence of @thm1−cl_strengthen and @thm2−cl_transitivity
  theorem @thm6−left_maximal: ∀ r·r;cl(r) ⊆ cl(r)

 // Head recursion
 // Proof:
 // − LHS is included in RHS: Instantiate @thm3−cl_minimal with r <− r and
      p <− r ∪ (r;cl(r))
 // then consequence of @thm1−cl_strengthen, @thm6−left_maximal and
      @thm2−cl_transitivity
 // − RHS is included in LHS: Consequence of @thm1−cl_strengthen and
      @thm6−left_maximal
  theorem @thm7−head_recursion: ∀r·cl(r) = r ∪ (r;cl(r))

 // Tail recursion
 // Proof:
 // − LHS is included in RHS: Instantiate @thm3−cl_minimal with r <− r and
      p <− (cl(r);r) ∪ r
 // then consequence of @thm6−right_maximal, @thm2−cl_transitivity, and
      @thm1−cl_strengthen
```

```
43  // − RHS is included in LHS: Consequence of @thm6−right_maximal and
        @thm1−cl_strengthen
44   theorem @thm8−tail_recursion: ∀r·cl(r) = (cl(r);r) ∪ r
45
46  // Induction principle
47  // Proof: Instantiate @thm3−cl_minimal with r <− r and p <− (T × T) ∪ ((
        STATE \ T) × STATE)
48   theorem @thm9−induction: ∀r,T · r[T]⊆T ⇒ cl(r)[T]⊆T
49
50  // Converse
51  // Proof:
52  // − LHS is included in RHS: Instantiate @thm3−cl_minimal with r <− r∼
        and p <− cl(r)∼,
53  // then consequence of @thm1−cl_strengthening and @thm2−cl_transitivity
54  // − RHS is included in LHS: Instantiate @thm3−cl_minimal with r <− r and
        p <− cl(r∼)∼,
55  // then consequence of @thm1−cl_strengthening (r <− r∼)and @thm2−
        cl_transitivity (r <− r∼)
56   theorem @thm10−converse: ∀r·cl(r∼) = cl(r)∼
57
58   // Transitive closure preserves domain
59   // Proof: Consequence of @thm7−head_recursion
60   theorem @thm11−domain_preservation: ∀r· dom(r) = dom(cl(r))
61
62   // Transitive closure preserves range
63   // Proof: Consequence of @thm8−tail_recursion
64   theorem @thm12−range_preservation: ∀r· ran(r) = ran(cl(r))
65
66  end
```

2.2 Formalisation of a Tree-Shaped Structure

The formalisation of a tree-shaped structure using transitive closure and is specified in 2 contexts as follows.

```
1  /*
2   * This context defines the notion of trees on a (sub−)set of states
3   */
4  context utstc_c0_3_tree_def
5  extends utstc_c0_2_closure_prop
6
7  constants
8   Tree
9  axioms
10  /*
11   * For each set @Sts, the set of trees on @Sts is the set of tuples
12   * from root @rt to the parent relation @prn satisfying:
```

```
13  * − @rt is in @Sts
14  * − every node except @rt has a parent in @Sts
15  * − @rt is an ancestor of all other nodes in @Sts via @prn
16  */
17  @def−Tree: Tree = {
18    Sts ↦ rt ↦ prn ↦ lvs |
19    Sts ⊆ STATE ∧
20    finite(Sts) ∧ // finite tree
21    rt ∈ Sts ∧
22    prn ∈ Sts \ {rt} → Sts ∧
23    (∀ n · n ∈ Sts \ {rt} ⇒ rt ∈ cl(prn)[{n}]) ∧
24    lvs = Sts \ ran(prn)
25   }
26
27  /*
28   * For a tree represented by @rt and @prn on a set of states @Sts,
29   * The set of all states @Sts is the root @rt and its descendants.
30   *
31   * Proof:
32   * − LHS is included in RHS: To prove that every non−root node x in S
33   * is a descendants of @rt. Property of trees ensure that the root is
34   * an ancestor of any non−root node, instantiate closure's @thm10−converse
35   * with r <− prn∼ to obtain cl(prn∼) = (cl(prn))∼
36   * − RHS is included in LHS: instantiate closure property
37   * @thm12−range_preservation with r <−− prn∼ to have ran(prn∼) = ran(cl(
         prn∼))
38   */
39  theorem @all_node: ∀ Sts, rt, prn, lvs · Sts ↦ rt ↦ prn ↦ lvs ∈ Tree ⇒
40    Sts = cl(prn∼)[{rt}] ∪ {rt}
41
42  /*
43   * Main induction principle on tree
44   * For a tree represented by @rt and @prn on a set of states @Sts,
45   * if the root @rt satisfies properties T (base case), and
46   *   the children of a state satisfying T also satisfies T (inductive case)
47   * then every state in @Sts satisfies T
48   *
49   * Proof:
50   * Instantiate closure's @thm9−induction with r <− prn∼ and T <− T.
51   * Instantiate @all_node to obtain Sts = cl(prn∼)[{rt}] ∪ {rt}
52   */
53  theorem @induction−Tree: ∀ Sts, rt, prn, lvs · Sts ↦ rt ↦ prn ↦ lvs ∈ Tree ⇒
54    (∀ T · rt ∈ T ∧ prn∼[T] ⊆ T ⇒ Sts ⊆ T)
55  end
```

```
1  context utstc_c0_4_tree_prop
```

```
extends utstc_c0_3_tree_def

//constants
// SubTree // Sub-tree binary relationships

axioms
 /*
  * For a tree represented by @rt and @prn on a set of states @Sts,
  * there are no cycles.
  *
  * Proof:
  * Instantiate tree's @induction-Tree with Sts <- Sts, rt <- rt,
  * prn <- prn, and T <- (Sts \ T) ∪ {rt}. Then do case "rt is in T" or not.
  */
 theorem @no-cycles: ∀ Sts, rt, prn, lvs · Sts ↦ rt ↦ prn ↦ lvs ∈ Tree ⇒
  (∀ T · T ⊆ Sts ∧ T ⊆ prn~[T] ⇒ T = ∅)

 /*
  * For a tree represented by @rt and @prn on a set of states @Sts, the
  * transitive closure of @prn is irreflexive.
  *
  * Proof:
  * Instantiate @no-cycles with Sts <- Sts, rt <- rt, prn <- prn.
  * Continue the proof by contradiction, i.e. assuming that there is a state
  * x where x ↦ x is cl(prn).
  * Instantiate @no-cylces with T <- cl(prn)~[{x}]
  */
 theorem @closure_irreflexive: ∀ Sts, rt, prn, lvs · Sts ↦ rt ↦ prn ↦ lvs ∈ Tree ⇒
  cl(prn) ∩ id = ∅

end
```

2.3 Formalisation of a Untriggered Statechart

Syntatical Elements The syntactical elements for an untriggered statechart is formalised in three layers: tree-shaped structured, regions, and transformations. For each layer, we split the syntatic elements and their constraints into two contexts.

Tree-shaped Structure States

```
context utstc_ctx0
extends utstc_c0_0_states

constants
 states // The set of states
 root // The root state
```

```
  container // The container relationship between states
  leaves // The set of leaf states

axioms
  theorem @typeof_states: states ⊆ STATE
  theorem @typeof_root: root ∈ STATE
  theorem @typeof_container: container ∈ STATE ↔ STATE
  theorem @typeof_leaves: leaves ⊆ STATE
end
```

```
context utstc_ctx0_tree
extends
  utstc_ctx0
  utstc_c0_3_tree_def
axioms
  // @root, @leaves and @container form a tree on @states
  @tree_structure: states ↦ root ↦ container ↦ leaves ∈ Tree

  // Derivable properties from definition of Tree
  theorem @typeof_root: root ∈ states
  theorem @typeof_container: container ∈ states \ {root} → states
  theorem @root_reachable: ∀ state · state ∈ states \ {root} ⇒ root ∈ cl(container)
      [{state}]
  // The set of leaf states is defined as those in @states and
  // does not contain any other states.
  theorem @def_leaves: leaves = states \ ran(container)
  theorem @leaves_are_states: leaves ⊆ states
end
```

Regions

```
context utstc_ctx1
extends utstc_ctx0
constants
  regions // The region (parallel states)
axioms
  theorem @region_type: regions ⊆ ℙ(STATE)
end
```

```
context utstc_ctx1_region
extends utstc_ctx1 utstc_ctx0_tree
axioms
  // regions partition the states accept root
```

```
@region_type: regions ⊆ ℙ(states)
@region_disjoint: ∀r1, r2 · r1 ∈ regions ∧ r2 ∈ regions ∧ r1 ≠ r2 ⇒ r1 ∩ r2 = ∅
@region_complete: union(regions) = states \ {root}

// all the states in the same region have the same parent
@region_same_parent: ∀ region · region ∈ regions ⇒ (∃parent · container[region]
    = {parent})

// every region is non−empty
theorem @region_non−empty: ∀ region · region ∈ regions ⇒ region ≠ ∅

// If a region r contains a child of s then every state in r must have s as its
    container
theorem @region−unique_container: ∀r, s · r ∈ regions ∧ r ∩ container∼[{s}] ≠ ∅
    ⇒ r ◁ container = r × {s}
theorem @region−unique_container2: ∀s · s ∈ ran(container) ⇒ (∃r·r ∈ regions ∧
    r ◁ container = r × {s})
end
```

Transformations

```
context utstc_ctx2
extends utstc_ctx1
sets
 transformations // The set of transformations
constants
 enabling // The enabling states of each transformation
 exiting // The exiting states of each transformation
 entering // The entering states of each transformation
axioms
 theorem @typeof−enabling: enabling ∈ transformations ↔ ℙ(STATE)
 theorem @typeof−exiting: exiting ∈ transformations ↔ ℙ(STATE)
 theorem @typeof−entering: entering ∈ transformations ↔ ℙ(STATE)
end
```

```
/*
 * This context defines notion of transformations, captured in terms of
 * "enabling" states, "exiting" states, "entering" states.
 */
context utstc_ctx2_transformation
extends utstc_ctx2 utstc_ctx1_region
axioms
 @enabling−type: enabling ∈ transformations → ℙ1(states)

  @exiting−type: exiting ∈ transformations → ℙ(states)
```

```
11
12  @exiting_either_one_or_all_in_a_region:
13   ∀trf, s, r ·
14    trf ∈ transformations ∧
15    s ∈ exiting(trf) ∧
16    r ∈ regions ∧
17    s ∈ r
18   ⇒
19    exiting(trf) ∩ r = {s} ∨ r ⊆ exiting(trf)
20
21
22  // if the container state s of a region r is an exiting state, there must some
       exiting state in r.
23  @exiting_contained_region:
24   ∀trf, s, r ·
25    trf ∈ transformations ∧
26    s ∈ exiting(trf) ∧
27    r ∈ regions ∧
28    container[r] = {s}
29   ⇒
30    exiting(trf) ∩ r ≠ ∅
31
32  // if s is an enabling state in a region r then s is the unique exiting state in r.
33  @enabling_unique_exiting_states_in_a_region:
34   ∀trf, s, r · trf ∈ transformations ∧
35    s ∈ enabling(trf) ∧
36    exiting(trf) ∩ r ≠ ∅ ∧// This to allowed the transformation where only
       enabling states are defined.
37    r ∈ regions ∧
38    s ∈ r
39   ⇒
40    exiting(trf) ∩ r = {s}
41
42   @exiting_unique_enabling_states_in_a_region:
43   ∀trf, s, r · trf ∈ transformations ∧
44    r ∈ regions ∧
45    exiting(trf) ∩ r = {s} ∧  // Either deterministically exiting r or (
       nondeterministically exiting r and r has one state)
46    r ≠ {s}              // r has more than one state (hence deterministically exiting
47   ⇒
48   enabling(trf) ∩ r = {s}  // s is the single enabling state in the region
49
50  @entering_type: entering ∈ transformations → ℙ(states)
51
52  @entering_at_most_one_in_a_region:
53   ∀trf, s, r ·
54    trf ∈ transformations ∧
55    s ∈ entering(trf) ∧
```

```
    r ∈ regions ∧
    s ∈ r
   ⇒
    entering(trf) ∩ r = {s}

// If a trf enters a container of a region r
// the trf must enter some states in r
 @entering−contained_region:
  ∀trf, r ·
   trf ∈ transformations ∧
   r ∈ regions ∧
   container[r] ⊆ entering(trf)
  ⇒
   entering(trf) ∩ r ≠ ∅

/*
 * For all transformation trf and a region r,
 *  if trf is entering r but not the container of r
 *  then the container of r is not an exiting state of trf
 */
@entering−stay_within_state:
 ∀trf, r, c ·
   trf ∈ transformations ∧
  r ∈ regions ∧
  container[r] = {c} ∧
  entering(trf) ∩ r ≠ ∅ ∧
  entering(trf) ∩ container[r] = ∅
 ⇒
  enabling(trf) ∩ cl(container)∼[{c}] ≠ ∅ ∧ c ∉ exiting(trf)

 @entering−you_must_go_somewhere:
 ∀trf, r ·
   trf ∈ transformations ∧
  r ∈ regions ∧
  container[r] ∩ exiting(trf) = ∅ ∧
  exiting(trf) ∩ r ≠ ∅
 ⇒
  entering(trf) ∩ r ≠ ∅  // You must go somewhere within r

@enter−through−parent−or−within−region:
 ∀trf, r ·
  trf ∈ transformations ∧
  r ∈ regions ∧
  entering(trf) ∩ r ≠ ∅ ∧
  exiting(trf) ∩ r = ∅
 ⇒
  entering(trf) ∩ container[r] ≠ ∅
```

```
104  @root−not−entering: ∀trf · trf ∈ transformations ⇒ root ∉ entering(trf)
105
106 end
```

Semantical Elements The semantical element of an untriggered statechart is represented mainly by an Event-B machine modelled how the active states is transformed during the execution of the statechart.

```
1  /*
2   * This context declare the initialisation of the active states.
3   */
4  context utstc_ctx
5  extends utstc_ctx2_transformation
6  constants
7   init_active
8  axioms
9   @typeof−init_active: init_active ⊆ states
10  @container_init_active:
11   ∀ s · s ∈ init_active \ {root} ⇒ container(s) ∈ init_active
12  @content_init_active:
13   ∀ s · s ∈ ran(container) ∧ s ∈ init_active ⇒ container∼[{s}] ∩ init_active ≠ ∅
14  @init_active−region−unique: ∀r, s · r ∈ regions ∧ s ∈ r ∩ init_active ⇒ r ∩
        init_active ⊆ {s}
15  @init_active−region−parallel: ∀region1, region2 · region1 ∈ regions ∧ region2 ∈
        regions ∧
16   container[region1] = container[region2] ∧ region1 ∩ init_active = ∅ ⇒ region2 ∩
        init_active = ∅
17 end
```

```
1  /*
2   * This machine captures the semantics of the statemachine as trace of active
        states.
3   */
4  machine utstc
5  sees utstc_ctx utstc_c0_4_tree_prop
6  variables
7   active // The set of active states
8
9  invariants
10  // @active is a set of states
11  @typeof−active: active ⊆ states
12
13  // If a non−root state is active then its container is also active
14  @container_active:
15   ∀ s · s ∈ active \ {root} ⇒ container(s) ∈ active
```

```
16
17  // If a state is active then all of its ancestor is active
18  theorem @ancestor_active:
19   ∀ s · s ∈ active ⇒ cl(container)[{s}] ⊆ active
20
21  // If a container is active then there must be an active sub−state
22  @content_active:
23   ∀ s · s ∈ ran(container) ∧ s ∈ active ⇒ container∼[{s}] ∩ active ≠ ∅
24
25  // At most one active in the same region
26  @active−region−unique: ∀r, s · r ∈ regions ∧ s ∈ r ∩ active ⇒ r ∩ active ⊆ {s}
27  theorem @active−region: ∀r, s · r ∈ regions ∧ s ∈ r ∩ active ⇒ r ∩ active = {s}
28
29  // All parallel regions are inactive (hence active) at the same time
30  @active−region−parallel: ∀region1, region2 · region1 ∈ regions ∧ region2 ∈ regions
        ∧
31   container[region1] = container[region2] ∧ region1 ∩ active = ∅ ⇒ region2 ∩
        active = ∅
32
33  // This is a consequence of @active−region−parallel
34  theorem @active−region−parallel−equiv: ∀region1, region2 · region1 ∈ regions ∧
        region2 ∈ regions ∧
35   container[region1] = container[region2] ⇒ (region1 ∩ active = ∅ ⇔ region2 ∩
        active = ∅)
36
37  // If a container of a region is active then the region must be active
38  // Use in transformation/active−region−parallel/INV
39  theorem @region_active:
40   ∀ r · r ∈ regions ∧ container[r] ⊆ active ⇒ r ∩ active ≠ ∅
41
42 events
43  event INITIALISATION
44  then
45   @update−active: active := init_active
46  end
47
48  event transformation
49  any
50   trf // the transformation to be executed
51  where
52   @typeof−trf: trf ∈ transformations
53   @active−enabling: enabling(trf) ⊆ active
54   // This theorem is intended to be use in proving
55   // transformation/active−region−unique/INV
56   theorem @active−no−exit−no−enter:
57    ∀s · s ∈ states \ {root} ∧ s ∈ active ∧ s ∉ exiting(trf) ⇒
58     (∀r · r ∈ regions ∧ s ∈ r ⇒ r ∩ entering(trf) = ∅)
59  then
```

```
60    @update−active: active := (active \ exiting(trf)) ∪ entering(trf)
61   end
62  end
```

2.4 An Example of Untriggered Statecharts

An example of untriggered statecharts can be created by "instatitate" the context accordingly. The following contexts corresponding to the Turnstile example described in the accompanying paper.

```
1  context utstc_ctx0_turnstile
2  extends utstc_ctx0
3  constants
4   OFF
5   ON
6   // GATE
7   BLOCKED
8   UNBLOCKED
9   TIMEOUT
10  // CARD_READER
11  READY
12  READING
13  ACCEPT
14 axioms
15  @def−STATE: partition(STATE, {root}, {OFF}, {ON}, {BLOCKED},
16   {UNBLOCKED}, {TIMEOUT}, {READY}, {READING}, {ACCEPT}
17  )
18  @def−states: states = STATE
19  @def−container: container = {OFF ↦ root, ON ↦ root, BLOCKED ↦ ON,
        UNBLOCKED ↦ ON,
20   TIMEOUT ↦ ON, READY ↦ ON, READING ↦ ON, ACCEPT ↦ ON
21  }
22  @def−leaves: leaves = {BLOCKED, UNBLOCKED, TIMEOUT, READY,
        READING, ACCEPT, OFF}
23 end
```

```
1  context utstc_ctx1_turnstile
2  extends utstc_ctx0_turnstile utstc_ctx1_region
3  axioms
4   @regions−def: regions = { {ON, OFF},
5    {BLOCKED, UNBLOCKED, TIMEOUT}, // GATE
6    {READY, READING, ACCEPT} // CARD_READER region
7   }
```

```
8 end
```

```
1  context utstc_ctx2_turnstile
2  extends utstc_ctx2_transformation utstc_ctx1_turnstile
3  constants
4   ON_2_OFF
5   OFF_2_ON
6   BLOCKED_2_UNBLOCKED
7   UNBLOCKED_2_TIMEOUT
8   TIMEOUT_2_BLOCKED
9   READY_2_READING
10  READING_2_READY
11  READING_2_ACCEPT
12  ACCEPT_2_READY
13 axioms
14  @def—transformations: partition(transformations,
15   {ON_2_OFF},
16   {OFF_2_ON},
17   {BLOCKED_2_UNBLOCKED},
18   {UNBLOCKED_2_TIMEOUT},
19   {TIMEOUT_2_BLOCKED},
20   {READY_2_READING},
21   {READING_2_READY},
22   {READING_2_ACCEPT},
23   {ACCEPT_2_READY}
24  )
25
26  @def—enabling: enabling = {
27   ON_2_OFF ↦ {ON},
28   OFF_2_ON ↦ {OFF},
29   BLOCKED_2_UNBLOCKED ↦ {BLOCKED},
30   UNBLOCKED_2_TIMEOUT ↦ {UNBLOCKED},
31   TIMEOUT_2_BLOCKED ↦ {TIMEOUT},
32   READY_2_READING ↦ {READY},
33   READING_2_READY ↦ {READING},
34   READING_2_ACCEPT ↦ {READING},
35   ACCEPT_2_READY ↦ {ACCEPT}
36  }
37
38  @def—exiting: exiting = {
39   ON_2_OFF ↦ {ON, BLOCKED, UNBLOCKED, TIMEOUT, READY,
        READING, ACCEPT},
40   OFF_2_ON ↦ {OFF},
41   BLOCKED_2_UNBLOCKED ↦ {BLOCKED},
42   UNBLOCKED_2_TIMEOUT ↦ {UNBLOCKED},
43   TIMEOUT_2_BLOCKED ↦ {TIMEOUT},
```

```
  READY_2_READING ↦ {READY},
  READING_2_READY ↦ {READING},
  READING_2_ACCEPT ↦ {READING},
  ACCEPT_2_READY ↦ {ACCEPT}
 }

 @def−entering: entering = {
  ON_2_OFF ↦ {OFF},
  OFF_2_ON ↦ {ON, BLOCKED, READY},
  BLOCKED_2_UNBLOCKED ↦ {UNBLOCKED},
  UNBLOCKED_2_TIMEOUT ↦ {TIMEOUT},
  TIMEOUT_2_BLOCKED ↦ {BLOCKED},
  READY_2_READING ↦ {READING},
  READING_2_READY ↦ {READY},
  READING_2_ACCEPT ↦ {ACCEPT},
  ACCEPT_2_READY ↦ {READY}
 }

end
```

3 Run-to-Completion Schedule

3.1 Triggers and Queues

We formalise the triggers and queues using three context as follows.

```
/*
 * This context declares the set of triggers
 * @author cfsnook
 */
context r2c_c0_0_triggers
sets
 TRIGGERS // The set of triggers
constants
 InternalTriggerType ExternalTriggerType
 nullTrigger
axioms
 @triggerKinds: partition (TRIGGERS, InternalTriggerType, ExternalTriggerType, {
     nullTrigger})
 theorem @nte: nullTrigger ∉ ExternalTriggerType
 theorem @nti: nullTrigger ∉ InternalTriggerType
end
```

```
/*
 * This context declares queues of triggers
```

```
3  * @author cfsnook – Initial model
4  * @author htson – Added sequence definitions
5  */
6  context r2c_c0_1_queues
7  extends r2c_c0_0_triggers
8
9  constants
10  Seq    // Sequences of triggers
11  Seq_empty  // Empty sequence
12  Seq_content // Content of a sequence
13  Seq_length  // length of a sequence
14  Seq_append  // Append a trigger to a sequence
15  Seq_concat  // Concatenation a sequence to another sequence
16  Seq_head  // Head of sequences
17  Seq_tail  // Tail of sequences
18  Seq_keep  // Keep certain element from set in a sequence
19  InternalQueueType // The internal queues of triggers
20  ExternalQueueType // The external queues of triggers
21 axioms
22  // Definition of Seq
23  @Seq–def: Seq = (λT · T ⊆ TRIGGERS | {n, s · n ∈ ℕ ∧ s ∈ 0 .. n − 1 → T | s})
24  theorem @Seq–type: Seq ∈ ℙ(TRIGGERS) → ℙ(ℕ ⇸ TRIGGERS)
25  theorem @Seq–finite: ∀s · s ∈ Seq(TRIGGERS) ⇒ finite(s)
26  // Definition of Seq_empty
27  @Seq_empty–def: Seq_empty = ∅ ⦂ ℙ(ℤ × TRIGGERS)
28  theorem @Seq_empty–type: ∀T·T ⊆TRIGGERS ⇒ Seq_empty ∈ Seq(T)
29  theorem @Seq–card: ∀s, m · s ∈ Seq(TRIGGERS) ∧ s ≠ Seq_empty ∧ dom(s) = 0
        ..m ⇒ card(s) = m+1
30  theorem @Seq–subset: ∀s, S · S ⊆ TRIGGERS ∧ s ∈ Seq(S) ⇒ s ∈ Seq(TRIGGERS
        )
31
32  // Definition of Seq_content
33  @Seq_content–def: Seq_content = (λs · s ∈ Seq(TRIGGERS) | ran(s))
34  theorem @Seq_content–type: Seq_content ∈ Seq(TRIGGERS) → ℙ(TRIGGERS
        )
35
36  // Definition of Seq_length
37  @Seq_length–def: Seq_length = (λs · s ∈ Seq(TRIGGERS) | card(s))
38  theorem @Seq_length–type: Seq_length ∈ Seq(TRIGGERS) → ℕ
39  theorem @Seq_length–card: ∀n, s · n ∈ ℕ ∧ s ∈ 0 .. n − 1 → TRIGGERS ⇒ card(s
        ) = n
40
41  // Definition of Seq_append
42  @Seq_append–def: Seq_append = (λ s ↦ t · s ∈ Seq(TRIGGERS) ∧ t ∈
        TRIGGERS |
43   { i ↦ v | i ∈ 0 .. Seq_length(s) ∧
44    (i < Seq_length(s) ⇒ v = s(i)) ∧
45    (i = Seq_length(s) ⇒ v = t)
```

```
 }
)
theorem @Seq_append−type: Seq_append ∈ Seq(TRIGGERS) × TRIGGERS →
    Seq(TRIGGERS)
theorem @Seq_append−pres: ∀T,s,t·T ⊆ TRIGGERS ∧ s ∈ Seq(T) ∧ t ∈ T ⇒
    Seq_append(s ↦ t) ∈ Seq(T)
// Definition of Seq_concat
@Seq_concat−def: Seq_concat = (λ s1 ↦ s2 · s1 ∈ Seq(TRIGGERS) ∧ s2 ∈ Seq(
    TRIGGERS) |
 { i ↦ v | i ∈ 0 .. Seq_length(s1) + Seq_length(s2) − 1 ∧
  (i < Seq_length(s1) ⇒ v = s1(i)) ∧
  (i ≥ Seq_length(s1) ⇒ v = s2(i − Seq_length(s1)))
 }
)
theorem @Seq_concat−type: Seq_concat ∈ Seq(TRIGGERS) × Seq(TRIGGERS)
    → Seq(TRIGGERS)
theorem @Seq_concat−pres: ∀T,s1,s2·T ⊆ TRIGGERS ∧ s1 ∈ Seq(T) ∧ s2 ∈ Seq
    (T) ⇒ Seq_concat(s1↦s2) ∈ Seq(T)
theorem @Seq_concat−empty: ∀T, s · T ⊆ TRIGGERS ∧ s ∈ Seq(T) ⇒
    Seq_concat(s ↦ ∅) = s

// Definition of Seq_head
@Seq_head−def: Seq_head = (λs · s ∈ Seq(TRIGGERS) ∧ s ≠ ∅ | s(0))
theorem @Seq_head−type: Seq_head ∈ Seq(TRIGGERS) \ {∅} → TRIGGERS

@Seq_tail−def: Seq_tail = (λs · s ∈ Seq(TRIGGERS) ∧ s ≠ ∅ |
 {i ↦ v | i ∈ 0 .. Seq_length(s) − 2 ∧ v = s(i+1)}
)
theorem @Seq_tail−type: Seq_tail ∈ Seq(TRIGGERS) \ {∅} → Seq(TRIGGERS)
theorem @Seq_tail−pres: ∀T,s·T ⊆ TRIGGERS ∧ s ∈ Seq(T) ∧ s ≠ ∅ ⇒ Seq_tail
    (s) ∈ Seq(T)

// Axiomatic definition of Seq_keep
@Seq_keep−type: Seq_keep ∈ Seq(TRIGGERS) × ℙ(TRIGGERS) → Seq(
    TRIGGERS)
@Seq_keep−empty: ∀T·T⊆TRIGGERS⇒Seq_keep(∅↦T) = ∅
@Seq_keep−individual: ∀s, T · s ∈ Seq(TRIGGERS) ⇒
        Seq_keep(s ↦ T) ∈ Seq(TRIGGERS)
@Seq_keep−content: ∀s, T· s ∈ Seq(TRIGGERS) ⇒
        Seq_content(Seq_keep(s ↦ T)) ⊆ T
@Seq_keep−concat: ∀s1, s2, T · s1 ∈ Seq(TRIGGERS) ∧ s2 ∈ Seq(TRIGGERS) ⇒
        Seq_keep(Seq_concat(s1 ↦ s2) ↦ T) = Seq_concat(Seq_keep(s1 ↦ T)
    ↦ Seq_keep(s2 ↦ T))

@Seq_keep−notEmpty1: ∀s,T,t · s ∈ Seq(TRIGGERS) ∧ t∈Seq_content(s) ∧ t∈
    T ⇒ t ∈ Seq_content(Seq_keep(s↦T))

```

```
@Seq_keep_append_d1: ∀t,T,s·T ⊆ TRIGGERS ∧ t ∈ T ∧ s ∈ Seq(TRIGGERS)
    ⇒
        Seq_keep(Seq_append(s↦t)↦T) = Seq_append(Seq_keep(s↦T)↦t)
@Seq_keep_append_d2: ∀t,T,s·T ⊆ TRIGGERS ∧ t ∉ T ∧ s ∈ Seq(TRIGGERS)
    ⇒
        Seq_keep(s↦T) = Seq_keep(Seq_append(s↦t)↦T)
@Seq_keep_head:   ∀T,s·T ⊆ TRIGGERS ∧ s ∈ Seq(TRIGGERS) ∧ s ≠
    Seq_empty ∧ Seq_head(s) ∈ T ⇒
        Seq_head(s) = Seq_head(Seq_keep(s↦T))

@Seq_keep_tail_d1:   ∀T,s·T ⊆ TRIGGERS ∧ s ∈ Seq(TRIGGERS) ∧ s ≠
    Seq_empty ∧ Seq_head(s) ∉ T ⇒
        Seq_keep(s↦T) = Seq_keep(Seq_tail(s)↦T)
@Seq_keep_tail_d2:   ∀T,s·T ⊆ TRIGGERS ∧ s ∈ Seq(TRIGGERS) ∧ s ≠
    Seq_empty ∧ Seq_head(s) ∈ T ⇒
        Seq_tail(Seq_keep(s↦T)) = Seq_keep(Seq_tail(s)↦T)
@Seq_keep_empty:   ∀s,T·s ∈ Seq(TRIGGERS) ∧ Seq_content(s) ∩ T = ∅
      ⇒ Seq_keep(s ↦ T) = ∅

@intQ: InternalQueueType = Seq(InternalTriggerType)
@extQ: ExternalQueueType = Seq(ExternalTriggerType)

end
```

```
/*
 * This context declares the type set for the dequeued triggers variable
 * @author cfsnook
 */
context r2c_c0_2_dequeue
extends r2c_c0_1_queues
constants
 DeQueueType // dequeued triggers type
axioms
 @deQ_1:   DeQueueType = {∅} ∪ {t · t ∈ TRIGGERS\{nullTrigger} | {t} }
 theorem @deQ_2:   ∅ ∈ DeQueueType
end
```

3.2 Formalisation of Run-to-Completion Schedule

A context and a machine are used to specify the run-to-completion steps.

Syntatical Elements

```
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
 @def_StepTrigger: StepTrigger ∈ Steps ⇸ Triggers
 @def_StepRaised: StepRaised ∈ Steps → Seq(InternalTriggers)
end
```

Semantical Elements

```
/*
 * This machine models the semantics for a SCXML model.
 *
 * @author cfs
 */
machine r2c
sees r2c_ctx

variables
 internal_queue // The queue of raised internal triggers
 external_queue // The queue of raised external triggers
 dequeue_trigger // The current dequeue trigger or none
 completed // Flag indicating that a run has been completed

invariants
 @internal_queue:      internal_queue ∈ Seq(InternalTriggers)
 @external_queue:      external_queue ∈ Seq(ExternalTriggers)
 @dequeue_trigger:     dequeue_trigger ∈ DeQueueType
 @dequeue_triggerwd:   dequeue_trigger ⊆ InternalTriggers ∪ ExternalTriggers
 @typeof_completed:    completed ∈ BOOL

 @firingTriggered:     dequeue_trigger ≠ ∅ ⇒ completed=FALSE  //just a sanity
     check
 theorem @readyToDequeue:  completed=TRUE ⇒ dequeue_trigger = ∅  //just
     a sanity check

events

 event INITIALISATION
```

```
28 then
29  @internal_queue_init:  internal_queue := Seq_empty
30  @external_queue_init:   external_queue := Seq_empty
31  @dequeue_trigger_init:   dequeue_trigger := ∅
32  @completed_init:    completed := FALSE
33 end
34
35 event raiseExternalTrigger
36 any trigger where
37  @trigger:       trigger ∈ ExternalTriggers
38 then
39  @external_queue_set:   external_queue := Seq_append(external_queue ↦
        trigger)
40 end
41
42 event dequeueExternalTrigger
43 any trigger where
44  @completed:      completed = TRUE
45  theorem @dequeue_trigger: dequeue_trigger = ∅ //sanity check
46  @external_queue−non_empty: external_queue ≠ ∅
47  @trigger:       trigger = Seq_head(external_queue)
48  @internal_queue:    internal_queue = ∅
49 then
50  @dequeue_trigger_set:   dequeue_trigger := {trigger}
51  @external_queue_set:   external_queue := Seq_tail(external_queue)
52  @completed_set:    completed := FALSE
53 end
54
55 event dequeueInternalTrigger
56 any trigger where
57  @completed:      completed = TRUE
58  theorem @dequeue_trigger: dequeue_trigger = ∅ //sanity check
59  @internal_queue−non_empty: internal_queue ≠ ∅
60  @trigger:       trigger = Seq_head(internal_queue)
61 then
62  @dequeue_trigger_set:   dequeue_trigger := {trigger}
63  @external_queue_set:   internal_queue := Seq_tail(internal_queue)
64  @completed_set:    completed := FALSE
65 end
66
67 event futureDequeue
68 when
69  @completed: completed = TRUE
70  theorem @dequeue_trigger: dequeue_trigger = ∅ //sanity check
71 then
72  @set_incompleted: completed := FALSE
73 end
74
```

```
75
76  event triggeredStep
77  any Step
78  where
79   @step:       Step ∈ dom(StepTrigger)
80   @trigger_dequeued:   StepTrigger(Step) ∈ dequeue_trigger
81   theorem @completed:  completed = FALSE //sanity check
82   theorem @unique_trigger: dequeue_trigger = {StepTrigger(Step)}
83  then
84   @dequeue_trigger_set:  dequeue_trigger := dequeue_trigger \ {StepTrigger(
         Step)}
85   @internal_queue_set: internal_queue := Seq_concat(internal_queue ↦
         StepRaised(Step))   //raised triggers
86  end
87
88  event untriggeredStep
89  any Step
90  where
91   @step:        Step ∈ Steps \ dom(StepTrigger)
92   @dequeue_trigger:     dequeue_trigger = ∅
93   @completed:       completed = FALSE
94  then
95   @internal_queue_set:   internal_queue := Seq_concat(internal_queue ↦
         StepRaised(Step))   //raised triggers
96  end
97
98  event completion
99  where
100  @dequeue_trigger:     dequeue_trigger = ∅
101  @completed:       completed = FALSE
102 then
103  @completed_set:     completed := TRUE
104 end
105
106 end
```

4 Triggered Statecharts

The formalisation of triggered statecharts is done by composing the formalisation of untriggered statecharts and the run-to-completion mechanism as follows.

4.1 Syntactical Elements

```
1  context tstc_ctx
2  extends r2c_ctx utstc_ctx
3  constants
```

```
  transitions // transition: link between transformation and Steps (both triggered
      and untriggered)
  discardSteps // steps for discarding triggers
axioms
  // Bijection between transformation and non−discard steps
  @typeof−transitions: transitions ∈ transformations ⤖ Steps ∖ discardSteps
  // Bijection between discardSteps and triggers (For each trigger there is exactly
      one and only one discard step and vice−versa
  @discardSteps−Triggers: discardSteps ◁ StepTrigger ∈ discardSteps ⤖ Triggers
  // Discard steps cannot raise any triggers
  @discardSteps−Raised: StepRaised[discardSteps] = { ∅ }
end
```

4.2 Semantical Elements

```
machine tstc
sees tstc_ctx
includes r2c as r2c
includes utstc as utstc

events
 event INITIALISATION
 synchronises r2c.INITIALISATION
 synchronises utstc.INITIALISATION
 end

 event raiseExternalTrigger
 synchronises r2c.raiseExternalTrigger
 end

 event dequeueExternalTrigger
 synchronises r2c.dequeueExternalTrigger
 end

 event dequeueInternalTrigger
 synchronises r2c.dequeueInternalTrigger
 end


 event discardTriggered
 any trigger
 synchronises r2c.triggeredStep
 where
  // r2c_Step is a discard step
  @Step−discard: r2c_Step ∈ discardSteps
  // trigger is the discard trigger
  @trigger−def: trigger = StepTrigger(r2c_Step)
```

```
33
34   // for all transformation @trf that is triggered by @trigger is disabled,
35   // i.e., not all enabling states of @trf is active.
36   @no_enabling: ∀trf · transitions(trf) ∈ StepTrigger∼[{trigger}] ⇒ ¬enabling(trf)
         ⊆ utstc_active
37  end
38
39  event triggeredTransition
40  synchronises r2c.triggeredStep
41  synchronises utstc.transformation
42  where
43   @this_trigger: transitions(utstc_trf) = r2c_Step
44  end
45
46  event untriggeredTransition
47  synchronises r2c.untriggeredStep
48  synchronises utstc.transformation
49  where
50   @this_trigger: transitions(utstc_trf) = r2c_Step
51  end
52
53  event completion
54  synchronises r2c.completion
55  where
56   // This theorem is useful for discharging no_enabling/WD PO
57   theorem @COPY−typeof−transitions: transitions∈transformations ⤖ Steps \
         discardSteps
58
59   // For all untriggered step r2c_Step, the corresponding transformation,
60   // i.e., transitions∼(r2c_Step) is not enabled.
61   @no_enabling:
62    ∀ r2c_Step · r2c_Step ∈ Steps \ dom(StepTrigger)
63   ⇒
64    ¬ (enabling(transitions∼(r2c_Step)) ⊆ utstc_active)
65  end
66
67 end
```

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.