Semantics Formalisation – From Event-B Contexts to Theories

Thai Son Hoang[1][0000−0003−4095−0732], Laurent Voisin[2][0000−0002−2426−0101], Karla Vanessa Morris Wright[3][0000−0002−0146−3176], Colin Snook[1][0000−0002−0210−0983], and Michael Butler[1][0000−0003−4642−5373]

[1] ECS, University of Southampton, Southampton SO17 1BJ, United Kingdom
{t.s.hoang,cfs,m.j.butler}@soton.ac.uk
[2] Systerel, 1115 rue René Descartes, 13100 Aix-en-Provence, France
laurent.voisin@systerel.fr
[3] Sandia National Laboratories, 7011 East Avenue Livermore, California 94550, USA
knmorri@sandia.gov

Abstract. The Event-B modelling language has been used to formalise the semantics of other modelling languages such as Time Mobility (TiMo) or State Chart XML (SCXML). Typically, the syntactical elements of the languages are captured as Event-B contexts while the semantical elements are formalised in Event-B machines. An alternative for capturing a modelling language's semantics is to use the Theory plug-in to build datatypes capturing the syntactical elements of the language and operators to represent the various semantical aspects of the language. This paper draws on our experience to compare the two approaches in both modelling and reasoning features.

1 Introduction

Previously, Event-B [**?**] has been used to formalise the semantics of modelling languages such as Time Mobility (TiMo) [**?**] or State Chart XML (SCXML) [**?**]. Essentially, the semantics of the languages are captured as discrete transition systems represented by the Event-B models. An advantage of this approach is that the generic properties of the semantics can be captured as invariants of the Event-B models while the syntactical constraints are expressed as the axioms, to ensure the correctness of the semantics. Recent work on the Theory plug-in for Rodin [**?**] enabled the formalisation of the Event-B method within the EB4EB framework [**?**].

Our motivation for this paper is to explore the use of the Theory plugin for capturing the semantics of other modelling languages. In particular, we want to compare the pros and cons of the two modelling styles, using Event-B models and the Theory plugin. We will use the SCXML as the example of the language to be modelled, in particular, focusing on the untriggered state machine fragment.

The structure of the paper is as follows. Section 2 gives some background information about Event-B, the Theory plugin, and the formalisation of SCXML

semantics using Event-B standard constructs, i.e., contexts and machines. Section 3 gives some comparison in formalising of the SCXML semantics using the Theory plugin and using Event-B standard constructs. Section 4 gives a summary of the paper.

2 Background

In this section, we briefly review the Event-B modelling method, the Theory plugin, and the formalisation of SCXML using Event-B models.

Event-B is a formal modelling method for system development [**?**]. An Event-B model contains two types of components: *contexts* and *machines*. Contexts represent the static part of an Event-B model and can contains carrier sets (types), constants and axioms constraining them. Machines capture the dynamic part of an Event-B model as transition systems where the states are represented by variables and the transitions are expressed as guarded events. An important feature of a machine is invariants which are safety properties that must be satisfied in all reachable states. Proof obligations are generated to ensure that the invariants are indeed established and maintained by the Event-B machines. To cope with system complexity, contexts can be extended by further contexts (adding more carrier sets, constants, or axioms), and machines can be refined. Consistent refinement in Event-B guarantees that safety properties (e.g, invariants) are maintained through the refinement process.

The Theory plugin for Rodin [**?**] enables developers to define new polymorphic data types and operators upon those data types. These additional modelling concepts (datatypes and operators) might be defined axiomatically or directly (including inductive definitions). Not only restricted in the modelling capability, the Theory plugin also offers the developers the opportunity of extending the reasoning capacity by writing automatic/interactive inference rules or rewrite rules.

A Formalisation of SCXML in Event-B is presented in [**?**]. SCXML [**?**] describes UML-style statemachines with run-to-completion semantics. SCXML diagrams provide a compact representation for modelling hierarchy, concurrency and communication in systems design. In [**?**], we develop a formalisation of the semantics of SCXML by separately modelling statemachines (untriggered statecharts) and the run-to-completion semantics (triggered mechanism), and combine them together using the inclusion mechanism [**?**]. Figure 1 summarises the formalisation as in [**?**]. Here, the purple rectangles correspond to Event-B contexts and the light-blue ovals represent Event-B machines. The labelled arrows capture the relationship between the different constructs including context extension (**extends**), machines see contexts (**sees**), and machine inclusion (**includes**). The two grey boxes indicate the two separate models of the statemachines and of the run-to-completion semantics.

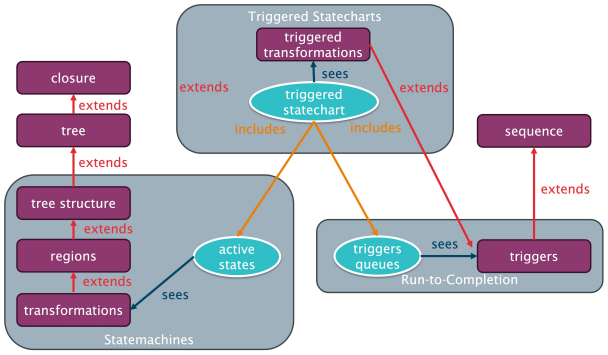

Fig. 1. Formalisation of SCXML using Event-B contexts and machines

- *Statemachines.* The model of the statemachines relies on the mathematical definition of irreflexive transitive closure (in context closure) and of a tree-shape structure (in context tree). The syntactical elements of statemachines are captured in three separate contexts (each one extending the other in order) named tree_structure, regions, and transformations. Machine active_states essentially specifies the semantics of the statemachines, captured as the set of active states.
- *Run-to-completion.* The model of the run-to-completion semantics relies on the notion of sequences (context sequences). The syntactic elements of the run-to-completion are captured in context triggers and the semantics are capture in machine triggers_queues. More specifically, the dynamic state of the run-to-completion represented by the external and internal queues modelled as sequences of triggers.
- *Triggered Statecharts.* A triggered statechart is a combination of the untriggered statechart and the run-to-completion semantic. In particular, machine triggered_statechart includes both machine active_states and triggers_queues.

3 Formalisation using Contexts/Machines vs Theories

In this section, we present an attempt to formalise the semantics of statemachines using theories, in comparison with the contexts/machines as in [**?**]. Figure 2 show our strategy for developing a semantics of statecharts using theories.

3.1 Formalisation of `closure`

In [**?**], the (irreflexive) transitive closure is formalised as a constant with an axiom defining its value. Various theorems capturing the properties of closure are derived from the axiom. Here STATE is a carrier set defining the set of states in the statemachines.

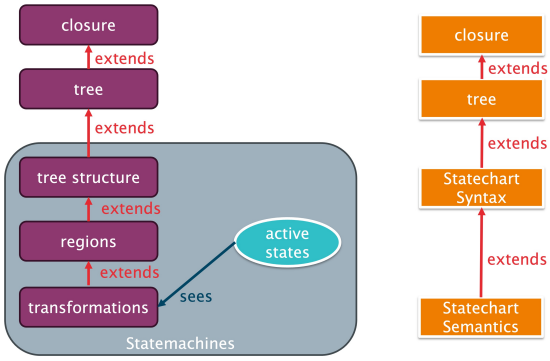

Fig. 2. Formalisation of statemachines: contexts/machines vs theories

```
constants closure
axioms
 @def−closure: closure = (λ r · r ∈ STATE ↔ STATE | inter({p | r ⊆ p ∧ p;p ⊆ p}))
 theorem @typeof−closure: closure ∈ (STATE ↔ STATE) → (STATE ↔ STATE)
 theorem @closure_strengthen: ∀ r· r ⊆ closure(r)
 theorem @closure_transitivity: ∀ r·closure(r);closure(r) ⊆ closure(r)
 theorem @closure_minimal: ∀ r· (∀ p· r⊆p ∧ p;p⊆p ⇒ closure(r)⊆p)
```

Using the Theory plug-in, closure is defined as an operator in a theory for type parameter S. This operator is polymorphic with respect to this type parameter and hence can be utilised in different contexts (compared with the constant defined in the context for a specific STATE set).

```
THEORY
  closure
TYPE PARAMETERS
  S
OPERATORS
  •closure  :  closure  EXPRESSION  PREFIX
  direct definition
   closure≙ (λ r · r ∈ S ↔ S | inter({p | r ⊆ p ∧ p;p ⊆ p}))
THEOREMS
  typeof-closure   :  closure ∈ (S ↔ S) → (S ↔ S)
  closure_strengthen   :  ∀r · r ∈ S ↔ S ⇒ r ⊆ closure(r)
  closure_transitivity   :  ∀r · r ∈ S ↔ S ⇒ closure(r);closure(r) ⊆ closure(r)
  closure_minimal   :  ∀r · r ∈ S ↔ S ⇒ (∀p · p ∈ S ↔ S ∧ r ⊆ p ∧ p;p ⊆ p ⇒ closure(r) ⊆ p)
```

3.2 Formalisation of `tree`

Using context, the definition of trees is defined as a constant Tree with its value defined using set comprehension and utilising transitive closure. In this definition, Sts represents the set of nodes in a tree, rt represents the root of the tree, and prn is the parent relationship of the tree.

```
axioms @def−Tree: Tree = {Sts ↦ rt ↦ prn |
  Sts ⊆ STATE ∧ rt ∈ Sts ∧ prn ∈ Sts \ {rt} → Sts ∧ (∀ n · n ∈ Sts \ {rt} ⇒ rt ∈ cl(prn)[{n}])}
```

Following the EB4EB framework [**?**], we formalise tree as a datatype with a well-definedness operator. The datatype is polymorphic with a type parameter NODE and operator TreeWD stating similar conditions in the axiom @def−Tree.

```
THEORY
  Tree
IMPORTS THEORY PROJECTS
  [Closure]
  THEORIES
     closure
TYPE PARAMETERS
  NODE
DATATYPES
  TREE(NODE)  ≙
   ▶ Cons_TREE(States:ℙ(NODE), Root:NODE, Parent:ℙ(NODE × NODE))
OPERATORS
  •TreeWD  :  TreeWD(tr : TREE(NODE))  PREDICATE  PREFIX
  direct definition
   TreeWD(tr : TREE(NODE)) ≙ Root(tr) ∈ States(tr) ∧
   Parent(tr) ∈ States(tr) \ {Root(tr)} ⟶ States(tr)
   ∧ (∀ n · n ∈ States(tr) \ {Root(tr)} ⇒ Root(tr) ∈ closure(Parent(tr))[{n}])
```

3.3 Formalisation of the Statechart Syntactical Elements

The syntactical elements of the statecharts are captured in three contexts to introduce the different aspects gradually: (1) tree-shape structure (2) parallel regions, and (3) transformations (an abstraction of transitions between states, including enabling, entering, and exiting states for each transformation). We will not present the details of the formalisation here, but refer the readers to [?]. Using the Theory plugin, we define the STATECHART datatype as in Figure 3.

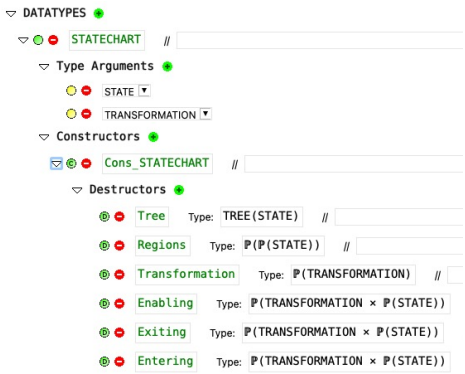

Fig. 3. Statechart datatype

Notice that we decide to define the STATECHART datatype all at once rather than gradually introducing its aspects. Datatypes in the Theory plugin are closed and hence cannot be extended. An example is the use of the Tree datatype for part of the STATECHART datatype resulting in a "nesting" effect. This result in the following operators to define the well-definedness for Regions of a statechart. In order to get to the states of a statechart st, one need to use States(Tree(st)) due to this nesting effect.

```
•RegionsWD  :  RegionsWD(st : STATECHART(STATE, TRANSFORMATION))  PREDICATE  PREFIX
direct definition
 RegionsWD(st : STATECHART(STATE, TRANSFORMATION)) ≙ Regions(st) ⊆ ℙ(States(Tree(st)))
 ∧ (∀r1, r2 · r1 ∈ Regions(st) ∧ r2 ∈ Regions(st) ∧ r1 ≠ r2 ⇒ r1 ∩ r2 = ∅)
 ∧ union(Regions(st)) = States(Tree(st)) \ {Root(Tree(st))}
 ∧ (∀ r · r ∈ Regions(st) ⇒ (∃p · p ∈ States(Tree(st)) ∧ Parent(Tree(st))[r] = {p}))
```

3.4 Formalisation of the Statechart Semantical Elements

In [**?**], the semantical elements of the statecharts are captured in a machine with a variable representing the active states of the statechart. Invariants capture properties, such as there is always an active state and if a child state is active then its parent state must be active.

Using the Theory plugin, we define a datatype ACTIVE_STATECHART for this purpose (we omit some details for spacing reason). This datatype wraps the STATECHART datatype together with the active states. The well-definedness operator ActiveStatechartWD captures the properties that we want to impose on the statechart semantics.

```
DATATYPES
  ACTIVE_STATECHART(STATE, TRANSFORMATION)  ≙
   ▶ Cons_ACTIVE_STATECHART(Statechart:STATECHART(STATE, TRANSFORMATION), Active:ℙ(STATE))
OPERATORS
  •ActiveStatechartWD  :  ActiveStatechartWD(a_sc : ACTIVE_STATECHART(STATE, TRANSFORMATION))  PREDICATE  PREFIX
  direct definition
   ActiveStatechartWD(a_sc : ACTIVE_STATECHART(STATE, TRANSFORMATION)) ≙ Active(a_sc) ⊆ States(Tree(Statechart(a_sc)))
   ∧ Active(a_sc) ≠ ∅
   ∧ (∀ s · s ∈ Active(a_sc) \ {Root(Tree(Statechart(a_sc)))} ⇒ Parent(Tree(Statechart(a_sc)))(s) ∈ Active(a_sc))
```

We define the transformation operator and state the following theorem (to be proved). The theorem says that any transformation of an active statechart preserves the well-definedness of the active statechart.

```
                ∀a_sc, tr · a_sc ∈ ACTIVE_STATECHART(STATE, TRANSFORMATION) ∧
thm1    :          ActiveStatechartWD(a_sc) ∧ tr ∈ Transformation(Statechart(a_sc)) ⇒
                   ActiveStatechartWD(transform(a_sc, tr))
```

4 Summary

This short paper provides some insights comparing the two modelling styles for formalising semantics of modelling languages: using Event-B contexts/machines vs using the Theory plugin's theories. In both approaches, the syntactical constraints on the models can be represented, either as axioms in the context or as the definition of the well-definedness operators. While contexts and context extensions are a natural way to introduce the syntactical elements of a modelling language gradually, essentially they implicitly represent a single model (e.g., a single statechart). Using the Theory plugin, a datatype and the corresponding well-definedness operator represent all valid models (e.g., all well-defined statecharts). The explicit representation of models as objects from a datatype allows us to write theorems in first-order logic about these well-defined models. We predicted that using Theory will help with stating and reasoning about model relationships such as refinement.