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 on the statemachines (part of SCXML) to compare the two approaches in terms of modelling efforts.

Keywords: Statecharts, SCXML, Event-B, Theory plugin, Semantics formalisation

1 Introduction

Previously, Event-B [1] has been used to formalise the semantics of modelling languages such as Time Mobility (TiMo) [4] or State Chart XML (SCXML) [9]. 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 [6] enabled the formalisation of the Event-B method within the EB4EB framework [7].

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.

In this short paper, we focus on the modelling efforts to capture the semantics of SCXML. More in-depth comparison including the proving efforts will be our future work.

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 [1]. An Event-B model contains two types of components: *contexts* and *machines*. Contexts represent the static part of an Event-B model and can contain 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 [3] 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 to the modelling capability, the Theory plugin also offers 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 [9]. SCXML [2] 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 [9], 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 [5].

Our formalisation of Statemachines using Event-B contexts and machines [9], 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. This can be seen on the left-hand side of Figure 1.

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 [9]. Figure 1 show our strategy for developing a semantics of statecharts using theories. Here inter represents the intersection.

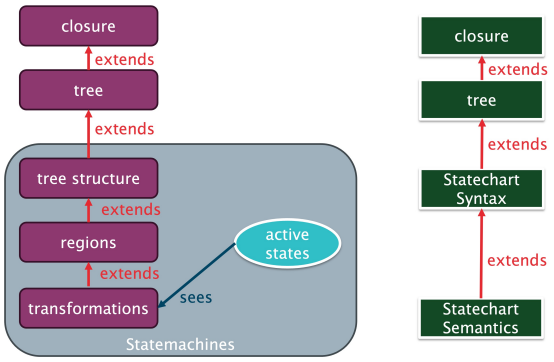

Fig. 1. Formalisation of statemachines: contexts/machines vs theories

3.1 Formalisation of `closure`

In [9], 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.

```
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 contexts, the definition of trees is given 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, prn is the parent relationship of the tree, and cl is the same previously defined closure operator.

```
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 [7], 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 [9]. Using the Theory plugin, we define the STATECHART datatype as in Figure 2. Notice that we decide to define the STATECHART datatype all at once rather

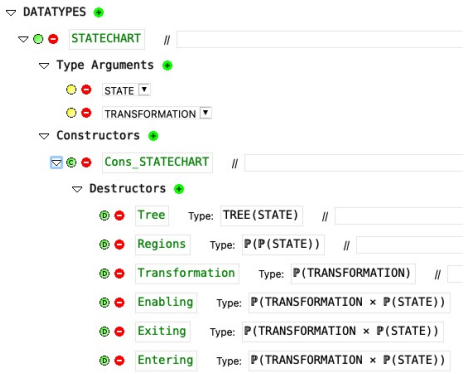

Fig. 2. Statechart datatype

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 results in operators of the form shown to define the well-definedness for Regions of a statechart. In order to get to the states of a statechart st, one needs 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 [9], 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 due to space reasons). 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))
```

The semantics of the statechart is captured as the following direct definition of the transform operator.

```
Cons_ACTIVE_STATECHART(Statechart(a_sc),
 (Active(a_sc) \ Exiting(Statechart(a_sc))(tr)) ∪ Entering(Statechart(a_sc))(tr))
```

Given a well-defined active statechart a_sc and a transformation tr of that statechart, i.e., tr ∈ Transformations(Statechart(a_sc)), the transform operator construct a new active statechart by updating the active states of the statechart by removing tr's exiting states and then adding the entering states of tr.

Consistency of the statechart semantics can now be expressed as the following theorem (to be proved).

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

The theorem says that any transformation of an active statechart preserves the well-definedness of the active statechart. The proof of this theorem using the Theory plugin will be our future work.

4 Summary

This short paper provides some insights comparing the two modelling styles for formalising semantics of modelling languages: using Event-B contexts/machines (Approach 1) vs using the Theory plugin's theories (Approach 2). Using Approach 1, essentially the model corresponds to a single statemachine, whereas with Approach 2, statemachines are modelled as a datatype. Context extension is a natural way to develop the statemachine's syntactical elements gradually in Approach 1, however, attempts to do this using theory extensions with Approach 2 results in nested datatypes which are cumbersome to use (see Section 3). An alternative is to use type class extension for theories [8]. In both approaches, the syntactical constraints on the models can be represented. In Approach 1, these are captured as axioms in the context contraining the syntactical elements. Using Approach 2, the well-formedness conditions are encoded as well-definedness predicate operators. Semantical properties are also captured in both approaches: as machine invariants in Approach 1, and as theory theorems in Approach 2. In the future, reasoning about refinement using Approach 1 requires duplication of the models (representing the abstract and the concrete statemachines). On the other hand, the explicit representation of statemachines as objects from a datatype in Approach 2 allows us to write theorems in first-order logic about these well-defined objects. We expect that using Theory will help with stating and reasoning about refinement relationships. While we use the example of statemachines to compare Approaches 1 and 2, these comparisons are applicable to formalisation of other type of models, e.g., UML-B statemachines, SCXML statecharts, etc.

References

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

2. Barnett, J.: Introduction to SCXML, pp. 81–107. Springer International Publishing, Cham (2017), https://doi.org/10.1007/978-3-319-42816-1_5
3. Butler, M.J., Maamria, I.: Practical theory extension in Event-B. In: Liu, Z., Woodcock, J., Zhu, H. (eds.) Theories of Programming and Formal Methods - Essays Dedicated to Jifeng He on the Occasion of His 70th Birthday. Lecture Notes in Computer Science, vol. 8051, pp. 67–81. Springer (2013), https://doi.org/10.1007/978-3-642-39698-4_5
4. Ciobanu, G., Hoang, T.S., Stefanescu, A.: From TiMo to Event-B: Event-driven timed mobility. In: 2014 19th International Conference on Engineering of Complex Computer Systems, Tianjin, China, August 4-7, 2014. pp. 1–10. IEEE Computer Society (2014), https://doi.org/10.1109/ICECCS.2014.10
5. Hoang, T.S., Dghaym, D., Snook, C.F., Butler, M.J.: A composition mechanism for refinement-based methods. In: 22nd International Conference on Engineering of Complex Computer Systems, ICECCS 2017, Fukuoka, Japan, November 5-8, 2017. pp. 100–109. IEEE Computer Society (2017), https://doi.org/10.1109/ICECCS.2017.27
6. Hoang, T.S., Voisin, L., Salehi, A., Butler, M.J., Wilkinson, T., Beauger, N.: Theory plug-in for Rodin 3.x. CoRR **abs/1701.08625** (2017), http://arxiv.org/abs/1701.08625
7. Riviere, P., Singh, N.K., Ameur, Y.A., Dupont, G.: Formalising liveness properties in Event-B with the reflexive EB4EB framework. In: Rozier, K.Y., Chaudhuri, S. (eds.) NASA Formal Methods - 15th International Symposium, NFM 2023, Houston, TX, USA, May 16-18, 2023, Proceedings. Lecture Notes in Computer Science, vol. 13903, pp. 312–331. Springer (2023), https://doi.org/10.1007/978-3-031-33170-1_19
8. Snook, J., Butler, M.J., Hoang, T.S.: Developing A new language to construct algebraic hierarchies for event-b. In: Feng, X., Müller-Olm, M., Yang, Z. (eds.) Dependable Software Engineering. Theories, Tools, and Applications - 4th International Symposium, SETTA 2018, Beijing, China, September 4-6, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10998, pp. 135–141. Springer (2018). https://doi.org/10.1007/978-3-319-99933-3_9, https://doi.org/10.1007/978-3-319-99933-3_9
9. Wright, K.V.M., Hoang, T.S., Snook, C.F., Butler, M.J.: Formal language semantics for triggered enable statecharts with a run-to-completion scheduling. In: Ábrahám, E., Dubslaff, C., Tarifa, S.L.T. (eds.) Theoretical Aspects of Computing - ICTAC 2023 - 20th International Colloquium, Lima, Peru, December 4-8, 2023, Proceedings. Lecture Notes in Computer Science, vol. 14446, pp. 178–195. Springer (2023), https://doi.org/10.1007/978-3-031-47963-2_12