Frontiers of Information Technology & Electronic Engineering
www.jzus.zju.edu.cn; engineering.cae.cn; www.springerlink.com
ISSN 2095-9184 (print); ISSN 2095-9230 (online)
E-mail: jzus@zju.edu.cn

Mechanized semantics and refinement of UML-Statecharts*

Feng SHENG, Liang DOU‡, Zong-yuan YANG

(*Department of Computer Science and Technology, East China Normal University, Shanghai 200241, China*)

E-mail: fsheng1990@163.com; ldou@cs.ecnu.edu.cn; yzyuan@cs.ecnu.edu.cn

Received Apr. 24, 2016; Revision accepted June 30, 2016; Crosschecked Nov. 29, 2017

Abstract: The Unified Modeling Language (UML) is an industry standard for modeling analysis and design. However, the semantics of UML is not precisely defined and the correctness of refinement relations cannot be verified. In this study, we use the theorem proof assistant Coq to formalize and mechanize the semantics of UML-Statecharts and the refinement relations between models. Based on the mechanized semantics, the desired properties of both the semantics and the refinement relations can be described and proven as predicates and lemmas. This approach provides a promising way to obtain certified fault-free modeling and refinement.

Key words: Unified Modeling Language (UML)-Statecharts; Coq; Refinement; Structured operational semantics
https://doi.org/10.1631/FITEE.1601196 **CLC number:** TP311.5

1 Introduction

Model-driven engineering (MDE) is a software engineering paradigm based on the specification of models of a system. In MDE, models are used as primary artifacts to drive engineering. The Unified Modeling Language (UML) has been developed as a standard object-oriented modeling notation in MDE and is well accepted in the industry. It offers different types of diagrams to describe different aspects of a system. UML-Statecharts, originally introduced by Harel *et al.* (1990), are used mainly to describe the dynamic behaviors of an object in its life cycle, including the sequences of the states of the object, the events that generate the transitions of the states, and the actions caused by the transitions. However, the syntax and semantics of notations are provided in terms of natural language descriptions, UML notations, and Object Constraint Language (OCL), which are not sufficient to express the semantics of UML notations precisely.

Moreover, the construction of MDE is driven by model refinement, starting with an initial design model, and a specification is often developed step-by-step. First, an abstract model is designed, and then the refinement takes place, introducing further details. Such details can either add new parts of the system or enhance its behaviors. This refinement step is repeatedly applied until an adequate level of description is obtained. As a result, the quality of the whole process strongly depends on the quality of refinement. UML cannot provide the formal specification of refinement in software design, which makes it difficult to formalize the refinement relations and verify their desired properties. Although formal methods and techniques that can promise the correctness of the desired properties in design models, are expensive, it is worth exploring how formal methods can be applied within MDE.

Coq (http://coq.inria.fr/) is an interactive theorem proof assistant for constructing programs and machine checkable proof based on the calculus of inductive constructions (CIC). CIC is a type theory

‡ Corresponding author

* Project supported by the National Natural Science Foundation of China (No. 61070226)

ORCID: Liang DOU, http://orcid.org/0000-0003-3044-3841

©Zhejiang University and Springer-Verlag GmbH Germany 2017

with dependent types, which allows us to write logical formulae of high order about objects of inductive and functional types and about potentially infinite structures of co-inductive types. The key to Coq is the Curry-Howard isomorphism, which connects types to logical propositions and well-founded functional programs to proof in constructive logic. Due to its very considerable expressive power and industrial-strength support, there has been much effort in using Coq to perform verification, such as CompCert (Leroy, 2015), JavaCard (Andronick *et al.*, 2003), and the four-color theorem (Gonthier, 2007).

In our previous work (Dou *et al.*, 2013), we demonstrated how to implement the mechanized semantics and refinement of UML sequence diagrams and perform formal verification in Coq. In this study, we use the Coq proof assistant to state specifications, create implementations, and build proofs for UML-Statecharts. The properties of refinement relations are described as lemmas using extra auxiliary functions, and the provability of lemmas indicates the correctness of the properties. The main contributions are listed as follows (source codes and the case study can be found at https://github.com/shengfeng/mechanized-semantics-of-statecharts):

1. The structured operational semantics (von der Beeck, 2002) of the UML-Statecharts is mechanized as object languages in Coq. The semantics includes entry and exit actions, a possibility to model inter-level transitions, specifying different history types, and a description of the non-termination situation. We redefine the potential configurations and extend the guard condition to make semantics more expressive.

2. We propose using Coq to formalize the refinement relations about UML-Statecharts, which can be mechanically proven to have the transitive, deterministic, and behavior-preserving properties. Transitivity implies that the correctness of multi-step refinement can be decomposed to the correctness of one-step refinement. The refinement relations should be deterministic and not change the behavior of the models.

2 Background

In this study, we focus on inductive data types, functions, and inductive predicates. One way to define types in Coq is using inductive definitions. The natural numbers are defined as follows:

```
Inductive nat : Set :=
| O : nat
| S : nat -> nat.
```

The inductive type 'nat' denotes a natural number by two constructors. First, the 'O' constructor states that 'O' is a natural number which we will consider as zero. The 'S' constructor states that, given any number n, 'S n' is also a natural number, which is the successor of n. We can use the same approach to define inductive predicates. For example, we can define an inductive predicate that tells whether a number is even:

```
Inductive even : nat -> Prop :=
| Ezero : even O
| Esucc : forall n, even n -> even (S (S n)).
```

The type of even is 'nat $\rightarrow$ Prop'. This is the type of total functions from natural numbers to logical propositions. Coq also provides a keyword 'Definition' to define new types as follows:

```
Definition pair_nat : Set := nat * nat.
Definition list_nat : Set := list nat.
```

The type 'pair_nat' defines the type of pairs of natural numbers. Using built-in Coq notations, the value of 'pair_nat' can be written as (1, 3). The type 'list_nat' defines the type of a list of natural numbers. In Coq, the empty list is denoted as 'nil' while the nonempty list can be connected by the '::' notation. For example, the list '1, 3, 5' can be written as '1 :: 3 :: 5 :: nil'.

Here, we use the keyword 'Lemma' or 'Theorem' to signify proofs, though this is an alias for 'Definition'.

```
Lemma zero_is_even : Prop := even O.
Theorem plus_ex : Prop := forall n m, n + m = m + n.
```

For the first lemma, it is easy to prove that this lemma uses the tactic 'apply'. For the second, a tactic 'induction' divides the goal into two subgoals. This is the basic process for verifying lemmas in Coq. More information about Coq can be found in the Coq website (https://coq.inria.fr/distrib/current/stdlib/).

3 Formalizing structured operational semantics

In this section, we present the main concepts and definitions for UML-Statecharts that we will use in this study.

3.1 Abstract syntax

Let N, $\mathcal{T}$, $\mathcal{E}$, and A be countable sets of the names of states, transitions, events, and actions, respectively. We denote the events and actions by a, b, c..., and α, β, γ... indicate the sequences. For a set M, we define M^* as the set of finite sequences over M. G is a set of guard conditions that describe the operations of logic and relation. Then the set UML-SC is inductively defined to be the least set satisfying the following three types of states, where $n \in N$ and en, ex $\in A^*$. For simplicity, $s_0, s_1, ..., s_k$ can be abbreviated as $s_{0...k}$.

1. Basic-state

Denote $s = [n, (\text{en}, \text{ex})]$ a basic-state, where $\text{name}(s) = n$, $\text{type}(s) = \text{basic}$, and 'en' and 'ex' are sequences of the entry and exit actions, respectively.

2. Or-state

If $s_0, s_1, ..., s_k$ are states for $k > 0$, $\rho = \{0, 1, ..., k\}$, $l \in \rho$, HT is the history types in {none, deep, shallow}, $\wp(N) = 2^N$, and $T := \mathcal{T} \times \rho \times \wp(N) \times \mathcal{E} \times G \times A^* \times \wp(N) \times \rho \times \text{HT}$, then $s = [n, (s_{0...k}), l, T, (\text{en}, \text{ex})]$ is an or-state with $\text{type}(s) = \text{or}$. Note that $s_0, s_1, ..., s_k$ are the substates of s, l is the index of current active substate, and T is a set of transitions. For each transition $\text{tr} = (\text{tn}, i, \text{sr}, e, g, \alpha, \text{td}, j, \text{ht}) \in T$, we have functions of transition 'tr' as follows:

$$\text{name(tr)=tn, sou(tr)=}s_i\text{, souRes(tr)=sr,}$$
$$\text{ev(tr)=}e\text{, guard(tr)=}g\text{, actSeq(tr)=}\alpha\text{,}$$
$$\text{tarDet(tr)=td, tar(tr)=}s_j\text{, historyType(tr)=ht,}$$

where 'tn' is the name of transition 'tr', i and j are the indexes of source state and target state of 'tr', respectively, e and α are the triggers (input events) and actions of 'tr', respectively, and g is a guard condition. When the guard condition is satisfied, the transition will generate the set of actions α. Here, 'sr' and 'td' are the source restriction and target determinator of 'tr', respectively, which provide a possibility to perform an inter-level transition. Transition 'tr' is an inter-level transition, if its source restriction 'sr' or its target determinator 'td' differs from the empty set. Symbol 'ht' is the history type of 'tr'.

3. And-state

If $s_0, s_1, ..., s_k$ are states for $k > 0$, then $s = [n, (s_{0...k}), (\text{en, ex})]$ is an and-state with $\text{type}(s) = \text{and}$. Here, $s_0, s_1, ..., s_k$ are the parallel substates of s with all of them being active substates.

We give the Coq notations for these three types states as follows:

```
Definition seqact := list action.
...
Inductive history : Set :=
  | none : history
  | deep : history
  | shallow : history.
...
Definition trans :=
   tname * nat * set sname * event * guard *
   seqact * set sname * nat * history.
...
Definition entryexit := seqact * seqact.
...
Inductive sc : Type :=
| basic_sc : sname -> entryexit -> sc
| or_sc : sname -> list sc -> nat ->
    set trans -> entryexit -> sc
| and_sc : sname -> set sc -> entryexit -> sc.
```

This 'seqact' represents a list of 'actions'. The inductive type 'history' defines a set of history mechanisms, in which 'none', 'deep', and 'shallow' are three different types. The 'trans' indicates transitions in the or-state, 'entryexit' is a pair of entry and exit actions, and inductive type 'sc' denotes the three types of states.

3.2 Semantic auxiliary functions

In this subsection, we will introduce some semantic auxiliary functions for defining the structured operational semantics (SOS), including configurations, the set of all possible configurations, history mechanism, the entry and exit actions, and computing the next state. Because of the limited space, here we introduce only the configurations. Table 1 lists the main auxiliary functions that we will use in this study. The configuration of a state is used to denote the current active substate, which is a set of states. Function conf: SC $\rightarrow \wp(N)$, which is inductively defined along the structure of states, computes the current configuration of a given state s:

$$\text{conf}([n, (\text{en}, \text{ex})]) = \{n\},$$
$$\text{conf}([n, (s_{0...k}), l, T, (\text{en}, \text{ex})]) = \{n\} \cup \text{conf}(s_l),$$

Table 1 Semantic auxiliary functions in UML-Statecharts

Name	Purpose
conf: SC $\rightarrow \wp(N)$	'configuration' of a state is the set of the names of all current active substates.
conf_all: SC $\rightarrow (\wp(\wp(N)))$	'all configuration' of a state is the set of all potential configurations, which can be complete or incomplete.
default: SC $\rightarrow$ SC	'default' allows reentry of an or-state, such that the same substate becomes the current active substate as it has been the case.
entry: SC $\rightarrow \wp(A^*)$	'entry' is the set of all sequences of entry actions.
exit: SC $\rightarrow \wp(A^*)$	'exit' is the set of all sequences of exit actions.
next: HT $\times N \times$ SC $\rightarrow$ SC	If a transition is executed, 'next' is used to compute the next state according to its history mechanism and target determinator.

$$\mathrm{conf}([n,(s_{0\ldots k}),(\mathrm{en},\mathrm{ex})])=\{n\}\cup\bigcup_{i=0}^{k}\mathrm{conf}(s_i).$$

Obviously, the configuration of the basic-state is itself, the configuration of the or-state is the union of itself and the configuration of the set of its active substate, and the configuration of the and-state is the union of itself and all the configurations of its substates.

3.3 Semantic definitions

First, we define an auxiliary semantics that deals with only processing single input events. A labeled transition system (LTS) is taken as the semantic domain to deal with processing a single input event.

The auxiliary semantics $[\![s]\!]_{\mathrm{aux}}$ of a state s is given by the labeled transition system (SC, $L, \rightarrow, s_0) \in$ LTS, where SC is the set of states, $L = E \times G \times A^* \times \{\mathrm{true}, \mathrm{false}\}$ is the set of labels, $\rightarrow \subseteq$ SC $\times L \times$ SC is the transition relations, and s_0 is the initial state. For the sake of simplicity, we use $s \xrightarrow[\alpha]{e[g]}_f s'$ instead of $(s,(e,g,\alpha,f),s') \in \rightarrow$, and $s \overset{e[g]}{\nrightarrow}_f$ instead of $\nexists s'\ \alpha, s \xrightarrow[\alpha]{e[g]}_f s'$, where s and s' are the source and target states of these (semantic) transition rules, respectively. For an input event e, if the guard condition g is satisfied, the state s may perform a transition with output α and flag f to state s'. Intuitively, stuttering flag f states whether a semantic transition is performed, where $f=$ true denotes as a positive flag that at least one transition is taken (non-stuttering transition), while $f=$ false denotes as a negative flag without taking any transition (stuttering transition), just 'consumed' input event e. SOS rules $\rightarrow$ are defined in Fig. 1.

The explanation of the SOS rules is as follows:

1. basic: a basic state may perform a stuttering transition with an arbitrary input event e, and generate an empty action, a negative flag, and an identical state if the guard condition g is satisfied; i.e., the input event is just consumed.

2. or1: if tr $=(\mathrm{tn}, l, \mathrm{sr}, e, g, \alpha, \mathrm{td}, m, \mathrm{ht})$ is a transition of an or-state s, the source restriction 'sr' of 'tr' is the configuration of the current active substate s_l, and the input event e cannot trigger a transition whose priority is higher than 'tr', s will perform a non-stuttering transition with input event e and guard condition g.

3. or2: if the current active substate of an or-state can perform a non-stuttering transition with a 'positive flag', then the or-state may perform a non-stuttering transition with the same label.

4. or3: if the current active substate of an or-state can perform a stuttering transition with a 'negative flag', and the or-state cannot conduct a non-stuttering transition with a 'positive flag' with input event e and guard condition g, then the or-state may perform a stuttering transition with a 'negative flag'.

5. and: if each substate s_j of and-state s can perform a transition with input event e, output α_j, and flag f_j, then and-state s can perform a transition with the same input event e, the output is α_j in an arbitrary order, and the flag $\vee_{j=0}^{k} f_j$ is given by the logical disjunction of all flag f_j.

In our work, the proposition 'priority' is defined to determine whether an event can trigger a transition of substates:

```
Inductive priority : event -> sc -> Prop :=
| p_or : forall l lsc lt n tn i s e g a ee sr td h,
   set_In (tn, l, sr, e, g, a, td, i, h) lt ->
   (forall st, set_In st sr ->
   set_In st (conf (nth l lsc s))) ->
```

$$\text{basic}\ \frac{\text{true}}{[n,\ _]\ \xrightarrow[<>]{e[g]}_{\text{false}}\ [n,\ _]},$$

$$\text{or1}\ \frac{(\text{tn}, l, \text{sr}, e, g, \alpha, \text{td}, m, \text{ht}) \in T, \text{sr} = \text{conf}(s_l), s_l \overset{e[g]}{\nrightarrow}_{\text{true}}}{[n, (s_{0\ldots k}), l, T,\ _]\ \xrightarrow[\text{ex::}\alpha\text{::en}]{e[g]}_{\text{true}}\ [n, (s_{0\ldots k})_{[s_m/\text{next}(h,N,s_m)]}, m, T,\ _]} \left(\begin{array}{c}\text{ex} \in \text{exit}(s_l),\\ \text{en} \in \text{entry}(\text{next}(\text{ht}, \text{td}, s_m))\end{array}\right),$$

$$\text{or2}\ \frac{s_l\ \xrightarrow[\alpha]{e[g]}_{\text{true}}\ s_l'}{[n, (s_{0\ldots k}), l, T,\ _]\ \xrightarrow[\alpha]{e[g]}_{\text{true}}\ [n, (s_{0\ldots k})_{[s_l/s_l']}, l, T,\ _]},$$

$$\text{or3}\ \frac{s_l\ \xrightarrow[<>]{e[g]}_{\text{false}}\ s_l, [n, (s_{0\ldots k}), l, T] \overset{e[g]}{\nrightarrow}_{\text{true}}}{[n, (s_{0\ldots k}), l, T,\ _]\ \xrightarrow[<>]{e[g]}_{\text{false}}\ [n, (s_{0\ldots k}), l, T,\ _]},$$

$$\text{and}\ \frac{\forall j \in \{0, 1, ..., k\},\ s_j\ \xrightarrow[\alpha_j]{e[g]}_{f_j}\ s_j'}{[n, (s_{0\ldots k}),\ _]\ \xrightarrow[\alpha]{e[g]}_{\vee_{j=0}^{k} f_j}\ [n, (s'_{0\ldots k}),\ _]} \left(\begin{array}{c}\alpha \in \{\alpha_{b(0)} :: \ldots :: \alpha_{b(k)} |\\ \exists \text{bijection}\ b : \{0, 1, ..., k\} \to \{0, 1, ..., k\}\}\end{array}\right).$$

Fig. 1 Structured operational semantic rules

```
   priority e (or_sc n lsc l lt ee)
| p_and : forall n lsc e ee,
   (exists s, set_In s lsc -> priority e s) ->
   priority e (and_sc n lsc ee).
```

We define the inductive predicate ‘sred’ to describe the operational semantics with the single input event, in which each constructor of the inductive predicate ‘sred’ denotes a semantic rule in Fig. 1.

```
Inductive sred (st : state) : sc -> label -> sc -> Prop :=
| or1 : forall e g a n lsc l lt i tn s ee en ex tr sr td h s’,
 beval st g = true ->
 set_In (tn, l, sr, e, g, a, td, i, h) lt ->
 (forall sta, set_In sta sr ->
  set_In sta (conf (nth l lsc s))) ->
  ~ priority e (nth l lsc s) ->
  exit (nth l lsc s) ex  ->
  entry (nth i lsc s) en  ->
  tr = ex ++ a ++ en ->
  s’ = subst_or (or_sc n lsc i lt ee) (nth i lsc s)
  (next h td (nth i lsc s)) ->
  sred st (or_sc n lsc l lt ee) (e, g, tr, true) s’
  ...
with reconstruct_action (st : state) : event -> list sc ->
    list (list string) -> list sc -> bool -> Prop :=
| r_action_true : forall lsc ltr lsc’ e g,
 (exists sj, exists a, exists sj’,
  set_In sj lsc /\ sred st sj (e, g, a, true) sj’) ->
  (forall sj sj’, set_In sj lsc ->
  subst_and_r lsc sj sj’ lsc’ ->
  (exists tr’, exists f,
  set_In tr’ ltr /\ sred st sj (e, g, tr’, f) sj’)) ->
  (forall tr’ sj’, set_In tr’ ltr ->
  (exists sj, exists f, subst_and_r lsc sj sj’ lsc’ ->
  set_In sj lsc /\ sred st sj (e, g, tr’, f) sj’)) ->
  reconstruct_action st e lsc ltr lsc’ true
  ...
```

3.4 Complete semantics

To specify the UML-Statecharts in a more complete way, we describe the complete semantics with a sequence of input events based on the auxiliary semantics. The complete semantics can be divided into two parts: the input event is consumed and removed from the sequence, and the output action is added to the tail of actions to be used in the following steps.

The Kripke structure is applied for the semantic domain for dealing with the sequences of input events, as it is appropriate for modeling the output of one step serving as the input of the next steps. The complete semantics $[\![s]\!]$ of a state $s \in \text{SC}$ is given by the Kripke structure $(S, \text{st}, \Longrightarrow)$, where $S = \text{SC} \times E^*$ is the set of Kripke states K, $\text{st} = (s, t_0) \in S$ is the start state of K with $t_0 \in E^*$, and $\Longrightarrow \subseteq S \times S$ is the transition relation of K.

For the sake of simplicity, we use $(s, t) \Longrightarrow (s', t')$ instead of $(s, t, s', t') \in \Longrightarrow$. The following rules define the complete semantics using the auxiliary semantics:

$$\text{self}\ (s, t) \Longrightarrow (s, t),$$

$$\text{trans}\ \frac{s \xrightarrow[\alpha \quad f]{e[g]} s'}{(s,t) \Longrightarrow (s',t')}, \text{where} \begin{pmatrix} \exists e = \text{head}(t) \\ t' = \text{tail}(t) :: \alpha \end{pmatrix}.$$

The complete semantics of the UML-Statecharts is given by the inductive predicate 'sstar' based on 'sred':

```
Inductive sstar (st : state) :
   sc -> trace -> sc -> trace -> Prop :=
| sstar_self : forall s t, sstar st s t s t
| sstar_trans :
   forall s t s' t' s'' t'' a b e g,
   sstar st s t s' t' ->
   sred st s' (hd e t', g, a, b) s'' ->
   t'' = (tl t') ++ a -> sstar st s t s'' t''.
```

3.5 Desired properties

In this subsection, we will present some proofs of the desired properties of the UML-Statecharts using the theorem proof assistant Coq.

1. Determinacy: for an arbitrary state s_1, the next state of s_1 after performing a transition is deterministic.

2. Transitivity: for arbitrary states s_1, s_2, and s_3, if $(s,t) \Longrightarrow (s_2,t_2)$, $s_2 \xrightarrow[\alpha 2 \quad f2]{e2[g2]} s_3$, and $e_2 =$ (head true t_2) are all satisfied, then $(s,t) \Longrightarrow (s_3,t_3)$ can be derived, where $t_3 =$ (tail t_2) $+ \alpha_2$.

3. Reflexivity: for an arbitrary state 'st', st$\xrightarrow[\alpha \quad f]{e[g]}$ st should satisfy the reflexivity, where (e,g,α,f) is empty.

```
Theorem deterministic: forall st s1 s2 s3 l,
   sred st s1 l s2 -> sred st s1 l s3 -> s2 = s3.

Theorem transitive : forall st s t s' t' s'' t'' a b e g,
   sstar st s t s' t' ->
   sred st s' (hd e t', g, a, b) s'' ->
   t'' = (tl t') ++ a -> sstar st s t s'' t''.

Theorem reflexivity : forall st s, sred st s nil s.
```

4 Refinement relations

In general, we can refine a UML-Statechart by adding new parallel or sequential states, adding a new transition between two states, adding actions within the sequence of entry and exit, adding actions in transitions, etc. The incremental refinement in software modeling must be behavior-preserving.

4.1 Definitions

If s is a substring of t, we denote as $s \curvearrowright t$, described by the inductive predicate 'sub_seqact' in Coq:

```
Inductive sub_seqact : list action -> list action -> Prop :=
| subnil : forall l, sub_seqact nil l
| subcons1 :
   forall l1 l2 x, sub_seqact l1 l2 ->
   sub_seqact l1 (x :: l2)
| subcons2 :
   forall l1 l2 x, sub_seqact l1 l2 ->
   sub_seqact (x :: l1)(x :: l2).
```

If transition 'tr' is a well-formed transition of the list of states $s_{0\ldots k}$, we define it wellformed_tran($\{s_{0\ldots k}\}$, tr), which satisfies the following rules:

- sou(tr) $\in \{s_{0\ldots k}\}$,
- tar(tr) $\in \{s_{0\ldots k}\}$,
- (souRes(tr) $= \varnothing$)$\vee$ (souRes(tr) $\in$ conf_all(sou(tr))),
- (tarDet(tr) $= \varnothing$)$\vee$ (tarDet(tr) $\in$ conf_all(tar(tr))).

We use inductive predicate 'wellformed_tran' to describe well-formed transitions:

```
Inductive wellformed_tran : list sc -> trans -> Prop :=
| wellformed : forall lsc t a,
   set_In (name (nth (sou t) lsc a)) (names lsc) ->
   set_In (name (nth (tar t) lsc a)) (names lsc) ->
   ((souRes t = nil) \/
   set_In (souRes t) (conf_all (nth (sou t) lsc a))) ->
   ((tarDet t = nil ) \/
   set_In (tarDet t) (conf_all (nth (tar t) lsc a))) ->
   wellformed_tran lsc t.
```

Note that 'name' and 'names' indicate an acquired name of the states and its set in the list of states, respectively. 'SouRes' and 'tarDet' denote the source restriction and the target determinator, respectively.

Now we can define the one-step refinement relations of the UML-Statecharts.

Definition 1 Assuming $s_1, s_2 \in$ SC, if s_2 and s_1 satisfy the one-step refinement rules in Fig. 2, then s_2 is a one-step refinement of s_1, noting $s_1 \rightsquigarrow s_2$.

Some explanations of the one-step refinement relations are as follows:

1. and/or: and/or-add1 refines a basic-state to an and/or-state; and/or-add2 adds a new substate to an and/or-state; and/or-subst means that if s_1' is a one-step refinement of s_1, then we can refine an and/or-state by replacing s_1 with s_1'.

2. trans: trans-add adds transition 'tr' to T if it is a well-formed transition for a list of states $s_{0\ldots k}$;

$$\text{and}-\text{add1}\ \frac{\text{type}(s_0)...\text{type}(s_k)=\text{basic},\ \forall i\neq j,\ \text{name}(s_i)\neq\text{name}(s_j),\ \forall i,\ n\neq\text{name}(s_i)}{[n,_]\rightsquigarrow[n,(s_{0...k}),_]},$$

$$\text{and}-\text{add2}\ \frac{\text{type}(s')=\text{basic},\ \forall i,\ \text{name}(s_i)\neq\text{name}(s'),\ \text{name}(s')\neq n}{[n,(s_{0...k}),_]\rightsquigarrow[n,(s_{0...k},s'),_]},$$

$$\text{and}-\text{subst}\ \frac{s_i\rightsquigarrow s_i',\ \forall j=0,...,i-1,i+1,...,k,\ \text{name}(s_i')\neq\text{name}(s_j)}{[n,(s_{0...k}),_]\rightsquigarrow[n,(s_{0...k})_{[s_i/s_i']},_]},$$

$$\text{or}-\text{add1}\ \frac{\text{type}(s')=\text{basic},\ \text{name}(s')\neq n}{[n,_]\rightsquigarrow[n,s',0,\emptyset,_]},$$

$$\text{or}-\text{add2}\ \frac{\text{type}(s')=\text{basic},\ \forall i,\ \text{name}(s_i)\neq\text{name}(s'),\ \text{name}(s')\neq n}{[n,(s_{0...k}),l,T,_]\rightsquigarrow[n,(s_{0...k},s'),l,T,_]},$$

$$\text{or}-\text{subst}\ \frac{s_i\rightsquigarrow s_i',\ \forall j=0,...,i-1,i+1,...,k,\ \text{name}(s_i')\neq\text{name}(s_j)}{[n,(s_{0...k}),l,T,_]\rightsquigarrow[n,(s_{0...k})_{[s_i/s_i']},l,T,_]},$$

$$\text{trans}-\text{add}\ \frac{\text{tr}\in T,\ \text{wellformed_tran}(\{s_{0...k}\},\text{tr})}{[n,(s_{0...k}),l,T,_]\rightsquigarrow[n,(s_{0...k}),l,T\cup\{\text{tr}\},_]},$$

$$\text{trans}-\text{imp}\ \frac{g'\models g,\ \text{tr}=(\text{tn},i,\text{sr},e,g,\alpha,\text{td},j,\text{ht})\in T,\ \text{tr}'\in T}{[n,(s_{0...k}),l,T,_]\rightsquigarrow[n,(s_{0...k}),l,T_{[\text{tr}/\text{tr}']},_]},$$
$$\text{where}\quad \text{tr}'=(\text{tn},i,\text{sr},e,g',\alpha,\text{td},j,\text{ht}),$$

$$\text{act}-\text{add}\ \frac{\alpha\curvearrowright\alpha',\ \text{tr}=(\text{tn},i,\text{sr},e,g,\alpha,\text{td},j,\text{ht})\in T,\ \text{tr}'\in T}{[n,(s_{0...k}),l,T,_]\rightsquigarrow[n,(s_{0...k}),l,T_{[\text{tr}/\text{tr}']},_]},$$
$$\text{where}\quad \text{tr}'=(\text{tn},i,\text{sr},e,g,\alpha',\text{td},j,\text{ht}),$$

$$\text{en}-\text{add1}\ \frac{\text{en}\curvearrowright\text{en}'}{[n,(\text{en},\text{ex})]\rightsquigarrow[n,(\text{en}',\text{ex})]},$$

$$\text{en}-\text{add2}\ \frac{\text{en}\curvearrowright\text{en}'}{[n,(s_{0...k}),l,T,(\text{en},\text{ex})]\rightsquigarrow[n,(s_{0...k}),l,T,(\text{en}',\text{ex})]},$$

$$\text{en}-\text{add3}\ \frac{\text{en}\curvearrowright\text{en}'}{[n,(s_{0...k}),(\text{en},\text{ex})]\rightsquigarrow[n,(s_{0...k}),(\text{en}',\text{ex})]},$$

$$\text{ex}-\text{add1}\ \frac{\text{ex}\curvearrowright\text{ex}'}{[n,(\text{en},\text{ex})]\rightsquigarrow[n,(\text{en},\text{ex}')]},$$

$$\text{ex}-\text{add2}\ \frac{\text{ex}\curvearrowright\text{ex}'}{[n,(s_{0...k}),l,T,(\text{en},\text{ex})]\rightsquigarrow[n,(s_{0...k}),l,T,(\text{en},\text{ex}')]},$$

$$\text{ex}-\text{add3}\ \frac{\text{ex}\curvearrowright\text{ex}'}{[n,(s_{0...k}),(\text{en},\text{ex})]\rightsquigarrow[n,(s_{0...k}),(\text{en},\text{ex}')]}.$$

Fig. 2 One-step refinement relations

trans-imp indicates that if g' can imply to g, notation $g' \models g$, transition $\text{tr} = (\text{tn}, i, \text{sr}, e, g, \alpha, \text{td}, j, \text{ht})$ can be refined as $\text{tr}' = (\text{tn}, i, \text{sr}, e, g', \alpha, \text{td}, j, \text{ht})$.

3. act: act-add means that if α is a substring of α', $\text{tr} = (\text{tn}, i, \text{sr}, e, g, \alpha, \text{td}, j, \text{ht})$, and $\text{tr}' = (\text{tn}, i, \text{sr}, e, g, \alpha', \text{td}, j, \text{ht})$, then we can refine an or-state by replacing 'tr' with 'tr′'.

4. en/ex: en/ex-add1, en/ex-add2, and en/ex-add3 mean that if en′/ex′ is the refinement of en/ex, then we can refine en/ex with en′/ex′ in a basic-state, an or-state, and an and-state respectively.

According to the refinement relations we propose, we define the one-step refinement relations with the inductive predicate 'refineone' in Coq:

```
Inductive refineone (st : state) : sc -> sc -> Prop :=
| and_add1 : forall lsc ee n, all_basic lsc ->
   refineone st (basic_sc n ee) (and_sc n lsc ee)
   ...
| or_add1 : forall n s' ee, all_basic (s' :: nil) ->
   refineone st (basic_sc n ee)
   (or_sc n (s' :: nil) 0 nil (nil,nil))
   ...
| trans_add : forall n lsc l lt ee t',
   wellformed_tran lsc t' ->
   refineone st (or_sc n lsc l lt ee)
   (or_sc n lsc l (set_add trans_dec t' lt) ee)
   ...
| act_add : forall tn s1 sr e a a' g td s2 h lt ee lsc n l,
   set_In (tn, s1, sr, e, g, a, td, s2, h) lt ->
   sub_seqact a a' ->
   refineone st (or_sc n lsc l lt ee)
   (or_sc n lsc l
   (set_add trans_dec (tn, s1, sr, e, g, a', td, s2, h)
  (remove trans_dec (tn, s1, sr, e, g, a, td, s2, h) lt)) ee)
   ...
```

Definition 2 Assuming $s_1, s_2 \in \text{SC}$, if s_2 and s_1 both satisfy the following rules, s_2 is the multi-step refinement of s_1, noting $s_1 \hookrightarrow s_2$:

$$\text{one}\ \frac{s_1 \rightsquigarrow s_2}{s_1 \hookrightarrow s_2},$$

$$\text{reflex}\ \frac{s \in \text{SC}}{s \hookrightarrow s},$$

$$\text{tran}\ \frac{s \in \text{SC}, s_1 \hookrightarrow s, s \hookrightarrow s_2}{s_1 \hookrightarrow s_2}.$$

The inductive type 'refine' indicates the definition of the multi-step refinement relations:

```
Inductive refine(st : state) : sc -> sc -> Prop :=
| one : forall sc1 sc2,
   refineone st sc1 sc2 -> refine st sc1 sc2
| reflex : forall sc, refine st sc sc
| tran : forall sc1 sc0 sc2,
   refine st sc1 sc0 -> refine st sc0 sc2 ->
   refine st sc1 sc2.
```

4.2 Properties

In this subsection, we will present some invariant properties of the refinement relations. The correctness of the refinement relations are verified in Coq. For the sake of simplicity, we give only three theorems. All proofs are omitted because of limited space.

First of all, the one-step refinement relations preserve the elements in 'conf'.

Theorem 1 (conf_preserve) If $s_1 \rightsquigarrow s_2$, then $\text{conf}(s_1) \subseteq \text{conf}(s_2)$.

Then the one-step refinement relations must maintain the semantic transitions in Section 3. Note that the priority of the inner transition is higher than that of the outer transition. If a new transition with the same input event is added, the model must keep the priority of the semantic transitions.

Theorem 2 (behavior_pre) $\forall s_1, s_1', s_2 \in \text{SC}$, $\forall e \in E$, if $(s_1 \xrightarrow[\alpha]{e[g]}{}_{\text{true}}\ s_1') \wedge (s_1 \rightsquigarrow s_2)$, then $\exists s_2', (s_2 \xrightarrow[\alpha]{e[g]}{}_{\text{true}}\ s_2') \wedge (s_1' \rightsquigarrow s_2')$.

Finally, the reflexivity and transitivity should be satisfied in the refinement relations.

Theorem 3 (refine_ref and refine_trans) The refinement relations are reflexive and transitive.

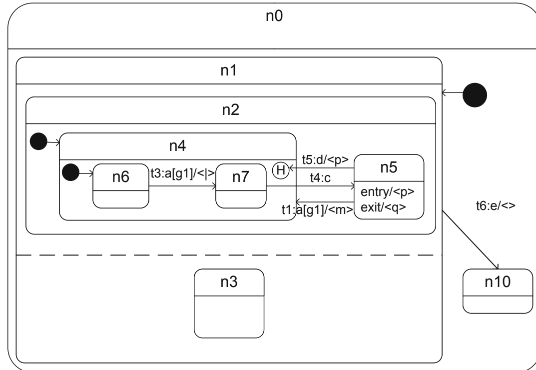
```
Theorem conf_preserve:
   forall st s1 s2 n, refineone st s1 s2 ->
   (set_In n (conf s1) -> set_In n (conf s2)).

Theorem behavior_pre: forall st s1 s1' s2 e g a,
   sred st s1 (e, g, a, true) s1' ->
   refineone st s1 s2 ->
   (exists s2', sred st s2 (e, g, a, true) s2' /\
    refineone st s1' s2').

Theorem refine_refl :
   forall st : state, reflexive _ (refine st).
Theorem refine_trans :
   forall st : state, transitive _ (refine st).
```

5 Case study

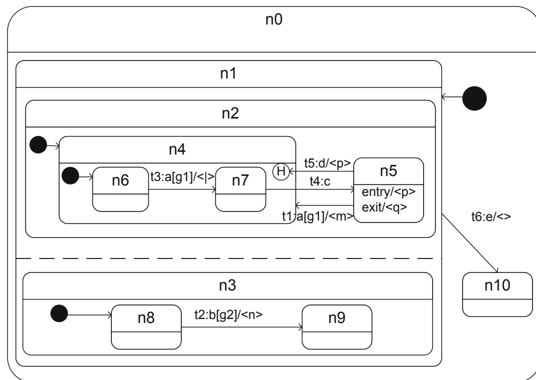
In this section, we will show how to describe and refine a UML-Statechart using Coq in Fig. 3. Only a part of codes are given because of space limitation. According to the abstract syntax, the syntax in Fig. 3a is as follows:

```
Definition t1: trans :=
  ("t1", 0, nil, "a", g1, ("m" :: nil), nil, 1, none).
Definition t3: trans :=
  ("t3", 0, nil, "a", g1, ("l" :: nil), nil, 1, none).
Definition t4: trans :=
  ("t4", 0, ("n4" :: "n7" :: nil), "c", BTrue, nil, nil, 1, none).
Definition t5: trans :=
  ("t5", 1, nil, "d", BTrue, ("p" :: nil), nil, 0, shallow).
Definition t6: trans :=
  ("t6", 0, nil, "e", BTrue, nil, nil, 1, none).
Definition s3 : sc := basic "n3" (nil, nil).
Definition s5 : sc := basic_sc "n5" ("p" :: nil, "q" :: nil).
Definition s6 : sc := basic_sc "n6" (nil, nil).
Definition s7 : sc := basic_sc "n7" (nil, nil).
Definition s10 : sc := basic_sc "n10" (nil, "s" :: nil).
Definition s4 : sc :=
  or_sc "n4" (s6 :: s7 :: nil) 0 (t3 :: nil) (nil, nil).
Definition s2 : sc :=
  or_sc "n2" (s4 :: s5 :: nil) 0 (t1 :: t4 :: t5 :: nil)(nil, nil).
Definition s1 : sc := and_sc "n1" (s2 :: s3 :: nil) (nil, nil).
Definition s0 : sc :=
  or_sc "n0" (s1 :: s10 :: nil) 0 (t6 :: nil) (nil, nil).
```

Note that s_1 is an and-state, s_0, s_2, and s_4 are or-states, and others are basic-states. The current active state is the first substate in the substate list in default. State s_5 has entry p and exit q actions. Transition t_4 is an inter-level transition where souRes$(t_4) = \{n_4, n_7\}$; i.e., the set of the source restriction is not empty, in which s_7 is the real transition source state. The other transitions are inner-level transitions as 'sr' and 'td' are both empty. Transition t_5 is a transition with a shallow history mechanism, which means that, when the transition performs, the target state will recover the active state as it has been active last time.



(a)



(b)

Fig. 3 An example of UML-Statechart: (b) is the refinement of (a)

We can compute all the configurations of state s_2 using the auxiliary function 'conf_all':

```
Eval compute in conf_all s2.
```

The result is

```
("n2" :: nil) :: ("n2" :: "n5" :: nil) ::
("n2" :: "n4" :: nil) :: ("n2" :: "n4" :: "n6" :: nil) ::
("n2" :: "n4" :: "n7" :: nil) :: nil.
```

Then we will perform refinement in Fig. 3a to Fig. 3b by processing a list of refinement relations, including changing a basic-state n_3 to an or-state containing n_8, adding a substate n_9 in n_3 and a transition t_2 between n_8 and n_9, and adding an exit action for n_9. Additional syntax in Fig. 3b is given as follows:

```
Definition s8 : sc := basic_sc "n8" (nil, "r" :: nil).
Definition s9 : sc := basic_sc "n9" (nil, nil).
Definition s3_2 : sc :=
  or_sc "n3" (s8 :: s9 :: nil) 0 (t2 :: nil) (nil, nil).
Definition t2: trans :=
  ("t2", 0, nil, "b", g2, ("n" :: nil), nil, 1, none).
```

To demonstrate that Fig. 3b is the refinement of Fig. 3a, we need to prove only the following theorems, where refine_s0_0_to_s0_1, refine_s0_1_to_s0_2, refine_s0_2_to_s0_3, and refine_s0_3_to_s0 are auxiliary theorems. The key part of proving process is the rules of the one-step refinement such as or_add1, trans_add, and ex_add1.

```
Lemma refine_s0_0_to_s0: forall st, refine st s0_0 s0.
Proof.
 intro st.
 apply tran with (sc0:=s0_1).
 apply refine_s0_0_to_s0_1.
 apply tran with (sc0:=s0_2).
 apply refine_s0_1_to_s0_2.
 apply tran with (sc0:=s0_3).
 apply refine_s0_2_to_s0_3.
 apply refine_s0_3_to_s0.
Qed.
```

The process of structured operational semantics can be described and verified in Coq. Beginning with the initial state in Fig. 3b, if the events are $< a, c, b, d, e >$, the UML-Statechart will perform a list of transitions with target state n_{10}. The theorem 'tranexmp' presents the mentioned execution

processes with the inductive predicate ‘sstar’. We can also prove that some transitions cannot occur. For instance, when guard condition g_1 is satisfied, the input event can trigger state n_6 to state n_7 or n_5. According to the low-first priority in UML, the target state should be n_7, while the transition from n_6 to n_5 should not occur as theorem ‘t1false’.

```
Theorem tranexmp: forall st,
   sstar st s6 ("a"::"c"::"b"::"d"::"e"::nil) s10 nil.

Theorem  t1flase : ~ sstar s6 s5.
```

If the proof fails for two particular models, then the refinement relations do not hold between the models. This process can help designers correct the system design.

6 Related work

In this section, we discuss the related work dealing with the formal semantics and refinement relations of UML-Statecharts.

Latella *et al.* (1999) proposed a formal transition from the UML-Statecharts to the PROMELA language in model checker SPIN. They presented the correctness verification of the transition and proved the security of the system in SPIN. Simons (2000) defined the denotational semantics of the UML-Statecharts with set theory. They used the terms of set theory to describe the states, events, and guard conditions of an object, providing the hierarchy of the models to support verification and testing. Börger *et al.* (2000) used abstract state machine to denote the formal semantics of UML-Statecharts. Broy *et al.* (2007) proposed that the formal semantics of the UML-Statecharts is attached to the state transition system. A state transition system consists of state space and state transition functions. Jürjens (2005) presented the formal semantics of UML-Statecharts using UML machines and UML machine systems, trying to establish the standard of the executable UML. Liu *et al.* (2013) presented a formal semantics of complete UML state machines with communications with LTS as its semantic domain.

Much work has dealt with the refinement relations of Harel statechart (Klein *et al.*, 1997; Scholz, 2001; Prehofer, 2013). UML-B (Snook and Butler, 2008) is a graphical front end for Event-B. UML-B is similar to UML but is essentially a new notation based on a separate meta-model. It also provides the tools to support drawing tools and a translator to generate Event-B models. Sun *et al.* (2004) discussed a co-algebraic description of UML-Statecharts, directly derived from their operational semantics. In particular, such an approach induces suitable notions of equivalence and (behavioral) refinement for statecharts. Lano and Clark (2008) presented an axiomatic semantics for UML 2.0 behavior state machines, and gave transformation rules to establish refinements of behavior state machines, together with proofs of the semantic validity of these rules, based on a unified semantics of UML 2.0. Said *et al.* (2009) introduced the concept of class and state machine refinement to support refining class and state machine in UML-B. Hallerstede and Snook (2011) defined two kinds of refinements for a state machine: node refinement and edge refinement.

In this study, we provide a formalization of the semantics of the UML-Statecharts and the refinement relations. The advantage of our approach is an intuitive and graph-based representation of the structured operational semantics and the refinement relations. We also prove that the one-step refinements are behavior-preserving, and multi-step refinements are reflexitive and transitive. All these studies strongly ensure the correctness of modeling and refinements of UML-Statecharts.

7 Conclusions and future work

The refinement relations of UML-Statecharts are an important task in MDE. In this paper, we have proposed the use of the theorem proof assistant Coq to mechanize the structured operational semantics of UML-Statecharts, as well as the refinement relations. Then we have verified the desired properties or the correctness of refinement relations based on the formal semantics. We have developed a tool which can automatically transform models to inductive definitions in Coq (http://www.kermeta.org/). This work provides a way to possibly obtain certified fault-free modeling and refinements.

Our future work will be directed toward the extension of the semantics and refinement relations considering the initial state, final state, and data dependencies during transitions. Besides, we will use integrated proof tactics as UML library to reduce proof code size and improve automation.

References

Andronick, J., Chetali, B., Ly, O., 2003. Using Coq to verify Java Card^{TM} applet isolation properties. Proc. Int. Conf. on Theorem Proving in Higher Order Logics, p.335-351. https://doi.org/10.1007/10930755_22

Börger, E., Cavarra, A., Riccobene, E., 2000. Modeling the dynamics of UML state machines. Proc. Int. Workshop on Abstract State Machines, p.223-241. https://doi.org/10.1007/3-540-44518-8_13

Broy, M., Cengarle, M., Rumpe, B., *et al.*, 2007. Towards a System Model for UML: the Structural Data Model. http://rzbl04.biblio.etc.tu-bs.de:8080/docportal/servlets/MCRFileNodeServlet/DocPortal_derivate_00003898/Document_00018887.pdf

Dou, L., Lu, L., Yang, Z., *et al.*, 2013. Towards mechanized semantics of UML sequence diagrams and refinement relation. Proc. 24th IASTED Int. Conf. on Modelling and Simulation, p.262-269. https://doi.org/10.2316/P.2013.802-021

Gonthier, G., 2007. The four colour theorem: engineering of a formal proof. Proc. 8th Asian Symp. on Computer Mathematics, p.333. https://doi.org/10.1007/978-3-540-87827-8_28

Hallerstede, S., Snook, C., 2011. Refining nodes and edges of state machines. Proc. Int. Conf. on Formal Engineering Methods, p.569-584. https://doi.org/10.1007/978-3-642-24559-6_38

Harel, D., Lachover, H., Naamad, A., *et al.*, 1990. STATEMATE: a working environment for the development of complex reactive systems. *IEEE Trans. Softw. Eng.*, **16**(4):403-414. https://doi.org/10.1109/32.54292

Jürjens, J., 2005. Secure Systems Development with UML. Springer-Verlag Berlin Heidelberg, Germany. https://doi.org/10.1007/b137706

Klein, C., Prehofer, C., Rumpe, B., 1997. Feature specification and refinement with state transition diagrams. Proc. 4th IEEE Workshop on Feature Interactions in Telecommunications Networks and Distributed Systems, p.284-297.

Lano, K., Clark, D., 2008. Semantics and refinement of behavior state machines. Proc. 10th Int. Conf. on Enterprise Information Systems, p.42-49.

Latella, D., Majzik, I., Massink, M., 1999. Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model-checker. *Form. Aspec. Comput.*, **11**(6):637-664. https://doi.org/10.1007/s001659970003

Leroy, X., 2015. The CompCert C verified compiler: documentation and user's manual. *Inria*, **16**(5):563-576.

Liu, S., Liu, Y., André, E., *et al.*, 2013. A formal semantics for complete UML state machines with communications. Proc. Int. Conf. on Integrated Formal Methods, p.331-346. https://doi.org/10.1007/978-3-642-38613-8_23

Prehofer, C., 2013. Behavioral refinement and compatibility of statechart extensions. *Electron. Notes Theor. Comput. Sci.*, **295**:65-78. https://doi.org/10.1016/j.entcs.2013.04.006

Said, M., Butler, M., Snook, C., 2009. Language and tool support for class and state machine refinement in UML-B. Proc. Int. Symp. on Formal Methods, p.579-595. https://doi.org/10.1007/978-3-642-05089-3_37

Scholz, P., 2001. Incremental design of statechart specifications. *Sci. Comput. Program.*, **40**(1):119-145. https://doi.org/10.1016/S0167-6423(00)00026-5

Simons, A., 2000. On the compositional properties of UML statechart diagrams. Proc. Rigorous Object-Oriented Methods Conf., p.1-12.

Snook, C., Butler, M., 2008. UML-B and Event-B: an integration of languages and tools. Proc. IASTED Int. Conf. on Software Engineering, p.336-341.

Sun, M., Zhang, N., Barbosa, L., 2004. On semantics and refinement of UML statecharts: a coalgebraic view. Proc. 2nd Int. Conf. on Software Engineering and Formal Methods, p.164-173. https://doi.org/10.1109/SEFM.2004.1347517

von der Beeck, M., 2002. A structured operational semantics for UML-statecharts. *Softw. Syst. Model.*, **1**(2):130-141. https://doi.org/10.1007/s10270-002-0012-8