Softw Syst Model (2009) 8:85–116
DOI 10.1007/s10270-007-0064-x

REGULAR PAPER

A compositional semantics of UML-RSDS

K. Lano

Received: 3 June 2006 / Revised: 25 May 2007 / Accepted: 29 June 2007 / Published online: 3 August 2007

Abstract This paper provides a semantics for the UML-RSDS (Reactive System Development Support) subset of UML, using the real-time action logic (RAL) formalism. We show how this semantics can be used to resolve some ambiguities and omissions in UML semantics, and to support reasoning about specifications using the B formal method and tools. We use 'semantic profiles' to provide precise semantics for different semantic variation points of UML. We also show how RAL can be used to give a semantics to notations for real-time specification in UML. Unlike other approaches to UML semantics, which concentrate on the class diagram notation, our semantic representation has behaviour as a central element, and can be used to define semantics for use cases, state machines and interactions, in addition to class diagrams.

Keywords UML semantics · UML-RSDS · Model transformations

1 Introduction

UML [42] is a large and complex notation, in which many aspects of the semantics remain incomplete or imprecise. Specific problems include:

1. Lack of distinction between concepts of precondition and guard in state machine semantics.
2. Lack of semantic consistency properties for individual models and between models of the same system [20].
3. The definition of behavioural compatibility.
4. Lack of consistent interpretation of concepts [40].

The upgrade of UML to UML 2.0 rationalised the metamodel structure of UML, but introduced further semantic complexities by enlarging the UML notation, for example to include Petri-net style models.

We solve some of these problems by using the following semantics approach:

1. Use a semantic model which is very general and supports treatment of large parts of UML, and of extensions of UML, for real-time and hybrid systems.
2. Use structured theories to decompose the semantics of a model into subtheories for individual classes and objects, so that instance-level reasoning can be carried out more efficiently.

We show how a complete semantics can be given to the UML-RSDS [30] subset of the UML 2 language, and we handle semantic variation in UML by defining 'semantic profiles' which incorporate these variant semantics.

UML-RSDS includes the following models:

1. Class diagrams.
2. State machine diagrams.
3. OCL [43] constraints.

UML-RSDS constraints include:

1. Local constraints of a class, referring only to data features of that class or its ancestors. Local constraints are the class invariant, and the pre and postconditions of

Communicated by Prof. Jean-Marc Jezequel.

K. Lano (✉)
Department of Computer Science, King's College London,
WC2R 2LS London, UK
e-mail: kevin.lano@kcl.ac.uk

operations of the class, which can also refer to parameters of their operation.
2. Inter-class constraints, attached to classes, associations or sets of associations. These may refer to data features of the connected classes.
3. State machine constraints: guards and state invariants, which may refer to data features of the class or its ancestors.

UML contains a number of notations which refer to time, such as time-based triggers in state machines, and the notation for interactions [42, Sect. 14]. It is also intended to extend UML-RSDS to include additional constraints to support the specification of concurrent and real-time aspects of a system, to support some of the features of the UML profile for real-time [41]. These facilities include:

1. Specification of durations of operation executions, and delay in a requested operation being executed.
2. Specification of periodic behaviour.
3. Specification of operation semantics as sequential, guarded or concurrent.
4. Specification of priority policies for request handling, such as "first come first served".

Therefore the semantics for UML-RSDS must support representation of time and properties of execution instances, at a detailed level.

A large number of relevant formalisms exist, including Real Time Logic (RTL) [3,25], Temporal Logic of Actions (TLA) [28], Duration Calculus [13] and real-time temporal logics [44,45]. We will use a simple but highly expressive formalism, RAL [29], based on RTL.

RAL directly supports the assignment of times to method initiations and terminations, and also contains an embedding of linear temporal logic, by interpreting "next time" as "next action invocation time". RAL is an extension of modal logics such as the object calculus of [17], and therefore can be used as a semantics for the B notation [1], which enables us to prove the correctness of a translation from UML-RSDS to B. RAL has been used to give a semantics to the real-time object-oriented language VDM^{++} [29]. The semantics described here is also used as the basis of the UML-RSDS tools [31].

In Appendix A we define precisely the metamodels and notation of UML-RSDS as a subset of UML 2. In Appendicies B and C the RAL formalism for UML-RSDS semantics is defined. Section 2 defines the semantics of UML-RSDS class diagrams. Section 3 defines the semantics of UML-RSDS state machine diagrams. Section 4 describes application of the semantics to verification of model transformations and semantic analysis of models. In Sect. 5 we consider extension to other notations of UML 2, and in Sect. 6 give comparisons to related work. Appendix D describes the UML-RSDS tools.

2 Semantics of class diagrams

The semantics of a class diagram model M is constructed in a modular fashion [5] from *instance theories* $\mathcal{I}_C$ of typical instances of classes C of the model, and *class theories* Γ_C of these classes, and *subsystem theories* Γ_S of subsystems S of the model. These are composed together to define a theory Γ_M of the complete model.

In the following sections we show how these theories are constructed incrementally from the elements of a class diagram.

2.1 Types

A model may define enumerated types T as enumerations of enumeration literals $val_1, \ldots, val_n$ as in UML. These types are represented in a theory Γ_T with no action symbols, and with a type symbol T defined as the appropriate finite set:

$$(ETD): \quad T = \{val_1, \ldots, val_n\}.$$

The val_i are defined as distinct constants of Γ_T (attributes which are not in the write frame of any action).

Types $Integer$, $Real$ and $Boolean$ are interpreted by the corresponding mathematical data types $\mathbb{Z}$, $\mathbb{R}$ and $\mathbb{B} = \{TRUE, FALSE\}$. $String$ is interpreted as the type $\mathbb{S}$ of sequences of characters. $\mathbb{B}$ and $\mathbb{S}$ are disjoint from $\mathbb{R}$ and from any enumerated type. All enumerated types are also disjoint from $\mathbb{R}$, $\mathbb{B}$ and $\mathbb{S}$.

2.2 Data features

If class C declares attributes $att_1 : T_1, \ldots, att_n : T_n$, then $\mathcal{I}_C$ has corresponding attributes $att_1 : T_1', \ldots, att_n : T_n'$ where T' is the semantic interpretation of T.

If an enumerated type T is used in an attribute declaration, then $\mathcal{I}_C$ is defined to extend the theory Γ_T.

If there is an association from C to a class D (Fig. 1), then any role at the D association end is represented in $\mathcal{I}_C$ as an attribute

$$(RTD): \quad role : DT$$

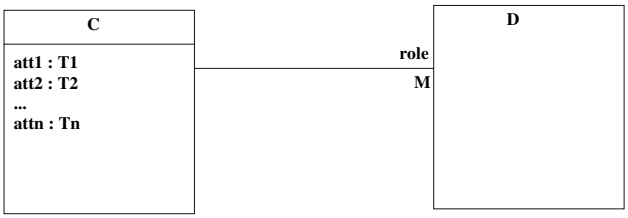


Fig. 1 Class definitions in UML-RSDS

Table 1 Representation of role multiplicities

Multiplicity M	Semantic type DT	Axiom
1	$@D$	
a..b	$\mathbb{F}(@D)$	$a \leq card(role) \wedge card(role) \leq b$
a	$\mathbb{F}(@D)$	$a = card(role)$
a..*	$\mathbb{F}(@D)$	$a \leq card(role)$
*	$\mathbb{F}(@D)$	

where DT is a type built from the type symbol $@D$ representing the type of possible instances of D. Table 1 shows the different cases of possible multiplicities M of $role$ and the corresponding DT type, and any additional axiom included in $\mathcal{I}_C$.

In the case that the $role$ association end is $\{ordered\}$, the sequence type $seq(@D)$ is used instead of $\mathbb{F}(@D)$.

In $\mathcal{I}_C$ we also include a Boolean attribute $exists_C : \mathbb{B}$ which indicates if $self$ currently exists as a valid object (i.e., if creation has occurred more recently than deletion).

Query operations $f(p_1 : PT_1, \ldots, p_n : PT_n) : RT$ of C are also represented as (constant) attributes

$$(FTD): \quad f(p_1 : PT_1', \ldots, p_n : PT_n') : RT'$$

of $\mathcal{I}_C$, where T' denotes the semantic representation of type T.

The definition of f is assumed to be given by a pre/post condition pair in which the $result$ parameter is used in the postcondition to denote the intended value of the query:

$$f(p : PT) : RT$$
$$\text{pre:} Pre_f$$
$$\text{post:} Post_f$$

This definition is semantically expressed by the axiom:

$$(FDef): \quad exists_C = TRUE \implies (\forall p : PT' \cdot Pre_f' \implies Post_f'[f(p)/result])$$

Local variables of update operations are also represented as instance theory attributes.

UML-RSDS attributes may have initial values defined in their declarations. These values are defined using pure literal values without any feature occurrences. The collection of all such initialisations $att_i = e_i$ are grouped together into a single new action $init_C$ defined as:

$$(InitDef): \quad init_C \supset att_1 := e_1' \parallel \ldots \parallel att_n := e_n' \parallel exists_C := TRUE$$

This has write frame $\{att_1, \ldots, att_n, exists_C\}$.

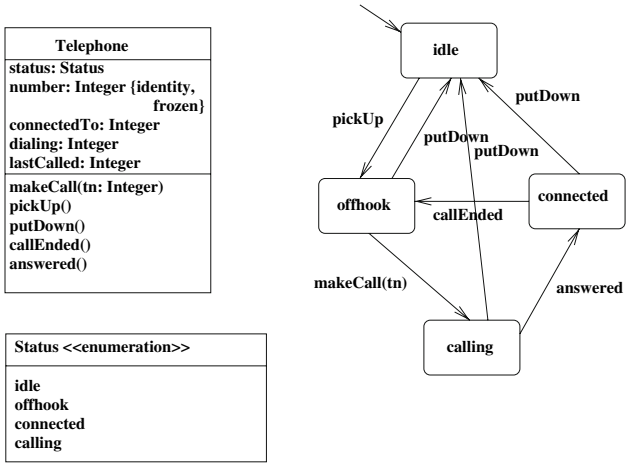


Fig. 2 Telephone class diagram and state machine

This action will in turn be invoked by the $create_C(a)$ action of the class theory (Sect. 2.7).

A $terminate_C$ action destroys the object:

$$(TermDef): \quad terminate_C \supset exists_C := FALSE$$

its write frame is $\{exists_C\}$.

A simple example of an instance theory could be that for a telephone (Fig. 2).

The $Telephone$ class has constraints:

```
dialing /= 0  =>  status = calling
status /= calling  =>  dialing = 0
status /= connected  => connectedTo = 0
connectedTo /= 0  => status = connected
```

The constraint that $lastCalled$ represents the last number dialed can only be directly expressed using RAL temporal operators. The operation $makeCall(tn : Integer)$ has the specification

```
makeCall(tn: Integer)
  pre: status = offhook
  post: dialing = tn
```

The data of the telephone class is represented by corresponding attributes of $\mathcal{I}_{Telephone}$:

$$status : Status$$
$$number : \mathbb{Z}$$
$$connectedTo : \mathbb{Z}$$
$$lastCalled : \mathbb{Z}$$
$$dialing : \mathbb{Z}$$

2.3 Operations

For each modifiable attribute $att : T$ of a class C there is assumed to be an operation $setatt(attx : T)$ which has the effect $post : att = attx$. It has write frame including att, but may need to modify other attributes in addition (to maintain invariants). *frozen* attributes do not have such operations. Likewise, for each modifiable rolename $role$ on the opposite end of an association incident to C, there is a $setrole$ operation, and $addrole$, $removerole$ operations if $role$ is not of multiplicity 1 ($removerole$ is omitted if $role$ is addOnly). These operations all have a standard definition, for example:

$addrole(rolex : D)$

pre: $(role \cup \{rolex\}).size \leq b$

post: $role = role@pre \cup \{rolex\}$

in the case of an unordered role of maximum cardinality b. All of these operations are represented as corresponding actions of $\mathcal{I}_C$.

User-defined update operations of C are also represented by actions of $\mathcal{I}_C$, with the same arity and set of input parameters. The declaration

$m(x_1 : X_1, \ldots, x_n : X_n)$

pre: Pre_m

post: $Post_m$

yields an action symbol $m(X'_1, \ldots, X'_n)$ of $\mathcal{I}_C$ and the axiom:

$$(OpD):$$
$$\forall i : \mathbb{N}_1; x_1 : X'_1; \ldots; x_n : X'_n \cdot$$
$$(exists_C = TRUE \wedge Pre'_m) \circledcirc \uparrow (m(x_1, \ldots, x_n), i) \implies$$
$$Post'_m[att \circledast \uparrow (m(x_1, \ldots, x_n), i)/att@pre] \circledcirc \downarrow$$
$$(m(x_1, \ldots, x_n), i)$$

In other words, if the precondition holds at commencement of an execution of $m(x_1, \ldots, x_n)$, then the postcondition holds at termination, with each $att@pre$ expression interpreted as the value of att at commencement.

This is the usual concept of precondition, in which no properties of the execution of the operation can be deduced if it is executed outside its precondition. This definition is used in languages such as B and Eiffel [38]. UML also assumes this definition as a default: "The behaviour of an invocation of an operation when a precondition is not satisfied is a semantic variation point" [42, p. 101] and p. 519 "…corresponds semantically to a precondition violation, for which no predefined behaviour is defined in UML".

However in [43] the semantics of a precondition as a permission guard is stated: "The precondition must evaluate to true whenever the operation starts executing." [43, Sect. 12.7]. Instead, we consider that it may be possible for an operation to be executed when its precondition fails, but that then no guarantee can be made about its behaviour (an implementation may throw an exception, for example). A separate proof obligation requires that callers ensure the precondition is true at the point where they make the call.

The *write frame* of an operation is the set of modifiable (non-frozen) attributes or roles att which it may change. This is calculated as the set of those attributes which:

1. Occur in prestate form $att@pre$ in $Post_m$, or
2. occur in a *writable modality* in $Post_m$, that is, in a subformula $att = exp$, $exp : att$, $exp /: att$ (except for addOnly roles att), $exp <: att$ where exp does not involve att except in the form $att@pre$.

in parameters of an operation cannot be modified in its postcondition.

If an update operation is defined by procedural code using the metamodel of Fig. 14:

$m(x : X)$

pre Pre_m then $Code_m$

then this is also represented by an action symbol $m(X')$ of $\mathcal{I}_C$ with write frame calculated from the form of $Code_m$ as for composite actions in Sect. C.4.

$Code_m$ can be interpreted as an RAL composite action $Code'_m$ in $\mathcal{I}_C$, and the effect of m expressed by the axiom

$$(OpP):$$
$$\forall x : X' \cdot m(x) \supset (pre\ exists_C = true \wedge Pre'_m\ then\ Code'_m)$$

In other words, if $m(x)$ is invoked when the object exists and the precondition holds, we are guaranteed to get the behaviour specified by $Code_m$.

If $Code_m$ itself involves operation calls $a.n(e)$ for $a : D$, or a collection a of D objects, these are interpreted in $Code'_m$ as actions $invoke_n(a', e')$ with empty write frames. Likewise a creation invocation $new_D(a)$ is interpreted as the action $create\_invoke_D(a')$ with empty write frame. These operations have no effect on the local state but will be linked with the behaviours they invoke in subsystem theories (Sect. 2.8).

Inconsistency between operation postconditions and class invariants can be detected by internal consistency checking using B [9]. Although the semantics can represent update operations defined in a self or mutually recursive manner, these cannot be translated to B for semantic analysis, since B does not permit such operations. However query operations can be defined recursively—these are translated as recursive functions (constant data) in B.

For the telephone specification, there are action symbols for the set operations $setstatus(statusx : Status)$, $setconnectedTo(connectedTox : Integer)$, etc., and for the user-defined operations $makeCall(tn : Integer)$, $pickup()$, etc.

Table 2 Semantic mapping for primitive literals

OCL term e	Semantics e'
number n	n
$true$	$TRUE$
$false$	$FALSE$
String "t"	Sequence denoted "t" consisting of characters of t in left to right order.
val from enumeration T	Representation of $T :: val$

$makeCall$ has the semantics:

$$\forall i : \mathbb{N}_1; n : \mathbb{Z} \cdot$$
$$(exists_{Telephone} = TRUE \ \wedge \ status = offhook) \circledcirc \uparrow$$
$$(makeCall(n), i) \implies (dialing = n) \circledcirc \downarrow (makeCall(n), i)$$

Additional functionality of this operation will be derived from the state machine of the $Telephone$ class in Sect. 3.

2.4 Expression semantics

This section defines the mathematical interpretation of OCL in our semantics. It also closely corresponds to the translation of OCL into B.

For primitive literal expressions e—numbers, strings, Booleans, and elements of enumerations, the semantic denotation e' of e directly corresponds to e (Table 2).

If v and w are two distinct enumeration literals (of the same or different enumerations) then their semantic denotations satisfy $v' \neq w'$. If v and w are syntactically the same, but belong to two distinct enumerations, then $v' \neq w'$. Otherwise $v' = w'$. The UML superstructure leaves this semantic aspect undefined [42, Sect. 7.3.16].

Also, for any enumeration literal v, $v' \notin \mathbb{R}$, $v' \notin \mathbb{B}$, $v' \notin \mathbb{S}$ and $v' \notin @C$ for any class C.

Collections (sets and sequences) in UML-RSDS are restricted to consist only of elements of a single type, as in [43]. This type can either be a numeric type, the Boolean type, the string type, a particular enumeration, or a particular class. Apart from elements of subclasses of a common superclass, mixtures of elements of different types are not allowed.

Collection literal expressions have a direct interpretation: a set literal $\{e_1, \ldots, e_n\}$ is interpreted by the mathematical set $\{e'_1, \ldots, e'_n\}$. This has type $\mathbb{F}(T)$ where T is the semantic representation of the common type of the elements of the set. A sequence literal $Sequence\{e_1, \ldots, e_n\}$ is interpreted by the mathematical sequence s of length n, which has $s(1) = e'_1, \ldots, s(n) = e'_n$. This is also written as $[e'_1, \ldots, e'_n]$. The typing of s is a sequence $1..n \rightarrow T$ where T is the semantic representation of the common type of the elements.

Table 3 Semantic mapping for collection operations

OCL term e	Condition	Semantics e'
$s.size$	Collection s	Cardinality $card(s')$
$x : s$	Set s	$x' \in s'$
$x \ / : s$	Set s	$x' \notin s'$
$x : s$	Sequence s	$x' \in ran(s')$
$x \ / : s$	Sequence s	$x' \notin ran(s')$
$s.asSet$	s set	s'
$s.asSet$	s sequence	$ran(s')$
$s \ <: \ t$	Sets s and t	$s' \subseteq t'$
$s \ <: \ t$	Sequence s or t	$(s.asSet \ <: \ t.asSet)'$
$s \ / <: \ t$	Sets s and t	$\neg(s' \subseteq t')$
$s \ / <: \ t$	Sequence s or t	$(s.asSet \ / <: \ t.asSet)'$
$s.sum$	Set s	Sum of elements of s'
$s.sum$	Sequence s, $card(s') = n$	$s'(1) + \cdots + s'(n)$

An identifier var denoting an attribute or role name of a class C is represented by the corresponding RAL attribute var in $\mathcal{I}_C$.

Numeric operators such as *, +, /, − are represented as corresponding function symbols of arity 2 on $\mathbb{R}$. The definitions of [43] are used, likewise for abs, $floor$, $>$, $<$, $<=$, $>=$, div and mod. The logical operators are interpreted by the corresponding semantic operators: & by $\wedge$, or by $\vee$, not by $\neg$ and $=>$ by $\implies$.

max and min apply to non-empty sets of elements (numerics or strings) comparable by $\leq$. For a non-empty set s, $(s.max)'$ is the unique element x of s' such that

$$y \in s' \implies y \leq x$$

Likewise $(s.min)'$ is the unique element z of s' such that

$$y \in s' \implies z \leq y$$

$size$, $=$ and $+$ ($concat$) are defined on strings as in [43]. Equality of strings means that they have the same characters in the same order (as in sequence equality). Likewise the Boolean operators or, &, not and $=>$ are defined according to the usual truth tables on $\mathbb{B}$.

On collections the operators :, / :, <:, / <: and $size$ are given the usual definitions of membership, non-membership, subset, negated subset and cardinality. Table 3 shows some examples of interpretations of collection operators.

The operators $\cup$, $\cap$ and $\frown$ are defined in terms of their mathematical counterparts (Table 4). $\cup$ on sequences is defined to be the same as $\frown$, as in [43].

Sequence-specific operations are defined in Table 5. $front$ and $tail$ of empty sequences are also empty. $first, last, tail$ and $front$ are defined on strings in the same manner, except that $(s.first)'$ is $[s'(1)]$ and $(s.last)'$ is $[s'(card(s'))]$ for a string s. $self$ in a select expression refers to the identity of

Table 4 Semantic mapping for collection operations

OCL term e	Condition	Semantics e'
$s \cup t$	s and t sets	$s' \cup t'$
$s \cap t$	s and t sets	$s' \cap t'$
$s - t$	s and t sets	$s' - t'$
$s \frown t$	s and t sequences	$s' \frown t'$

the objects of the first argument, which are being selected from.

Table 6 shows the semantics of navigation expressions on single objects.

Table 7 shows the semantics of navigation expressions which start from sets of objects.

In Table 7 $conc(seqs)$ is distributed concatenation of the sequences in $seqs$. $C.role$, $role$ not static, is treated as evaluation of $role$ over the object set $\overline{C}$. Navigations involving query operations are treated in a similar way to those with attributes or roles.

Select expressions evaluate to sets or sequences depending on the collection they operate over (Table 8). Their first argument must denote a finite set or sequence. $contract(m)$ turns a map $m : 1..n \nrightarrow T$ into a sequence sq by removing gaps in the index set, maintaining the same order of elements. For example $contract(\{2 \mapsto a, 3 \mapsto b, 7 \mapsto c\})$ is $[a, b, c]$.

The notation $a.P$ denotes the class version of P with a substituted into each new parameter slot, for example $a.(att > 10)$ is $att(a) > 10$. $a.self$ is a.

In a few cases, operators are overloaded. Table 9 details these cases. In cases where $s + t$ is used with one argument being a string and the other not, say t, then t is converted to a string before concatenation of values.

In cases of $x = y$ where x is a collection, and y is not, then y is promoted to a collection of the same kind as x, and likewise in the reverse case.

A set of useful algebraic laws relate OCL expressions, for example, if x and y are sets of objects of a class C, and f is a C attribute, P and Q constraints involving only features of C:

$$
\begin{aligned}
(x \cup y).f &= x.f \ \cup \ y.f \\
(x \cup y)|(P) &= x|(P) \ \cup \ y|(P) \\
x|(true) &= x \\
x|(false) &= \{\} \\
x|(P)|(Q) &= x|(P \ \& \ Q) \\
x|(P \ \& \ Q) &= x|(P) \ \cap \ x|(Q) \\
x|(P \ or \ Q) &= x|(P) \ \cup \ x|(Q)
\end{aligned}
$$

These can be proved using the semantics given above.

We can prove all of the axioms of the OCL standard library [43, Sect. 11] which relate to our subset of OCL, with the following differences:

Section 11.7.1 of [43]: We define the semantics of $size$ using the mathematical operator $card$, not using the computational approach ($iterate$) given in [43]. Likewise for $count$, which could be defined as

$$card(\{i | i \in dom(sq) \wedge sq(i) = x\})$$

for the count of x in a sequence sq.

Section 11.7.5 of [43]: at is not given a precise semantics in OCL, only the precondition

$$pre: \ i \geq 1 \ \& \ i \leq s.size$$

is given for $s \rightarrow at(i)$. In contrast in our semantics it is precisely defined as function application $s'(i')$.

Our definition of $-$ on sequences corresponds to an iteration of the OCL $excluding$ operation.

2.5 Invariants

The following are proof obligations for consistency of a class, which a developer must ensure, for example by specifying that additional actions execute when the initialisation takes place, or when some update operation takes place.

Table 5 Semantic mapping for sequence operations

OCL term e	Condition	Semantics e'
$s = t$	s and t sequences	$s' = t'$ as maps
$s.first$	Non-empty sequence s	$s'(1)$
$s.last$	Non-empty sequence s	$s'(card(s'))$
$s.front$	Non-empty sequence or string s	Subsequence $[s'(1), \ldots, s'(card(s') - 1)]$ of s'
$s.tail$	Non-empty sequence or string s	Subsequence $[s'(2), \ldots, s'(card(s'))]$ of s'
$sq - cl$	Sequence sq, collection cl	$(sq \vert (self \ / : cl))'$
$s.sort$	Sequence s	Reordering of s' such that elements are in non-descending $<$ order
$s.reverse$	Sequence s, $n = card(s')$	$\{i \mapsto s'(n - i + 1) \vert i \in dom(s')\}$

Table 6 Semantic mapping for navigation expressions

OCL term e	Condition	Semantics e'
$obj.att$	Attribute att	$att(obj')$
$obj.role$	1-Multiplicity role	$role(obj')$
$obj.role$	Unordered collection-valued role	Set $role(obj')$
$obj.role$	Ordered collection-valued role	Sequence $role(obj')$

The invariant Inv_C of a class must be established by the initialisation:

$$(InitInv): \quad [init_C]Inv'_C$$

The invariant Inv_C must hold at the initiation and termination of every update operation:

$$(PInv): \quad \Box_S(exists_C = TRUE \implies Inv'_C) \wedge \forall x_1 : X_1 \cdot [\alpha_1(x_1)](exists_C = TRUE \implies Inv'_C) \wedge \ldots \wedge \forall x_n : X_n \cdot [\alpha_n(x_n)](exists_C = TRUE \implies Inv'_C)$$

where $S = \{\alpha_1, \ldots, \alpha_n\}$ is the set of actions representing the update operations of C, which have corresponding parameters $x_1 : X_1$, etc.

Because of the frame axioms, these two requirements ensure that the semantics of an invariant in UML [43, Sect. 12] are valid: "The invariant must be true for each instance of the classifier at any moment in time. Only when the instance is executing an operation, this does not need to evaluate to true" ([43, Sect. 7.3.3] incorrectly leaves out the qualification).

The UML-RSDS tools permit a specifier to leave the definition of operations incomplete, and instead to specify, implicitly, their behaviour in some cases by defining suitable invariant constraints – in this sense constraints are the primary specification mechanism in UML-RSDS, and operation definitions are secondary, or derived [30].

From the constraints it is possible to deduce what actions must co-execute with a given updating action in order that the invariants are preserved. If the operation op has basic code $Code_{op}$, then in order that this should establish an invariant I, the weakest precondition $[Code'_{op}]I'$ must be true when $Code_{op}$ activates. This is ensured if the additional code Add_{op} has the property $[Add'_{op}][Code'_{op}]I'$.

For example, consider a class with a set-valued role r and an integer attribute x with the constraint I:

$$x = r.size$$

The operation $setr(rx)$ with effect $r := rx$ requires additional code such that $[Add'_{op}][r := rx]I'$, that is: $[Add'_{op}](x = card(rx))$.

The UML-RSDS tool automatically derives such Add_{op} code from the condition $[Code'_{op}]I'$, using definitions of the *update form* of this condition [30]. If an update form does not exist for $[Code'_{op}]I'$, then this is added as an additional precondition of op.

In the case of the constraint I, Add_{op} can be derived as:

$$setx(rx.size)$$

In general it will be the case that Add_{op} cannot update either variable read in $Code_{op}$: the variable v changed by $Code_{op}$ and the parameter value vx used to modify v, so $Add'_{op}; Code'_{op}$ and $Add'_{op} \| Code'_{op}$ are equivalent with

Table 7 Semantics of navigation expressions on collections

OCL term e	Condition	Semantics e'
$objs.att$	$objs$ unordered	$\{att(obj) \mid obj \in objs'\}$
	$objs$ ordered	$\{i \mapsto att(objs'(i)) \mid i \in dom(objs')\}$
$objs.role$	$objs$ unordered	$\{role(obj) \mid obj \in objs'\}$
$role$ 1-multiplicity	$objs$ ordered	$\{i \mapsto role(objs'(i)) \mid i \in dom(objs')\}$
$objs.role$	$objs$ unordered and	$\bigcup(\{role(obj) \mid obj \in objs'\})$
$role$ not 1-multiplicity	$role$ unordered	
$objs.role$	$objs$ unordered and	$\bigcup(\{ran(role(obj)) \mid obj \in objs'\})$
$role$ not 1-multiplicity	$role$ ordered	
$objs.role$	$objs$ ordered and	$\bigcup(\{role(objs'(i)) \mid i \in dom(objs')\})$
$role$ not 1-multiplicity	$role$ unordered	
$objs.role$	$objs$ and $role$ ordered	$conc(\{i \mapsto role(objs'(i)) \mid i \in dom(objs')\})$
$role$ not 1-multiplicity		
$C.att$	att static	att
$C.att$	att instance scope	$\{att(x) \mid x \in \overline{C}\}$
$C.size$		$card(\overline{C})$

Table 8 Semantic mapping for select expressions

OCL term e	Condition	Semantics e'
$objs\|P$	Set $objs$	$\{x \mid x \in objs' \wedge x.P'\}$
$objs\|P$	Sequence $objs$	$contract(\{i \mapsto x \mid (i \mapsto x) \in objs' \wedge x.P'\})$

Table 9 Semantic mapping for overloaded expressions

OCL term e	Condition	Semantics e'
$x + y$	x and y numbers	$x' + y'$ (numeric addition)
$s + t$	s or t a string	$s' \frown t'$ (string concatenation)
$x - y$	x and y numbers	$x' - y'$ (numeric subtraction)
$x - y$	x or y collections	$x' - y'$ (collection subtraction)
$x = y$	x set-valued, y not	$x' = \{y'\}$
$x = y$	x sequence-valued, y not	$x' = [y']$
$x = y$	y set-valued, x not	$\{x'\} = y'$
$x = y$	y sequence-valued, x not	$[x'] = y'$
$x = y$	Other cases	$x' = y'$
$br[ind]$	br sequence	$br'(ind')$
$br[ind]$	br string	$[br'(ind')]$

regard to their weakest preconditions. In Java we use the ; form of combination to implement the constraints, and in B the || form.

For the telephone system, the invariants are expected to be true at each operation start and termination time, for an existing object. In particular the invariant

$$dialing /= 0 \implies status = calling$$

places a requirement for additional behaviour on $makeCall(n)$ which we can derive as:

$$if\ n \neq 0\ then\ status := calling$$

This should co-execute with the existing behaviour.

2.6 Inheritance

If class C inherits from class D, then $\mathcal{I}_C$ incorporates $\mathcal{I}_D$. In addition there are axioms linking the creation, destruction and existence of objects of C and D:

$$(OIAx):$$
$$exists_C = TRUE \implies exists_D = TRUE$$
$$init_C \supset init_D$$
$$terminate_D \supset terminate_C$$

These correspond to the relationship $\overline{C} \subseteq \overline{D}$ in the class theory, and express the semantics "each instance of the specific classifier is also an indirect instance of the general classifier." [42, p. 67].

This model of inheritance also has the consequence that "any constraint applying to instances of the general classifier also applies to instances of the specific classifier" [42, p. 50]. It is not clear however if this last statement should apply to pre and postconditions of operations of the classifiers, we discuss this aspect further in Sect. 3.4 below.

Despite the transitive nature of this concept of generalisation, it is not the case that UML generalisation is transitive: it is possible for

$$a = g1.general$$
$$b = g1.specific$$
$$b = g2.general$$
$$c = g2.specific$$

for classes a, b, c and generalizations $g1$, $g2$, without there existing any generalization between a and c.

Similar issues apply to interface realization, which also uses the 'satisfaction of all constraints' condition when comparing a classifier to an interface it is implementing [42, p. 85].

There are two points in UML 2 where relaxation of the 'all constraints of the general class should be satisfied in the specific' condition appears:

1. Attributes may have default values for their initialisation [42, p. 46]. If these defaults differ in a subclass and superclass, then the semantics of the classes will also differ. In our semantics any initialisation performed in the superclass cannot be varied by the subclass. There are no defaults.
2. Likewise, static features are permitted to change their values in subclasses, in [42]. This can be modelled by considering such redefinition as the declaration of a new static feature, with a name qualified by the name of the redefining class, and semantically unrelated to the feature it replaces.

We use the 'one object' view of specialisation [21]: even though an object may be classified by many classifiers (related in an inheritance hierarchy), it is represented as the same semantic element in each. Its identity cannot change. It is however possible for an object to move from one subclass of a class to another subclass (dynamic classification), by the occurrence of $create$ and $kill$ actions of these subclasses on the object.

The axioms $OIAx$ and OpD together imply that if an operation is defined both in the superclass and subclass, then both sets of pre/post specifications apply when it is used on an object of the subclass ($exists_C$ and $exists_D$ both true). A semantic variation in which the subclass postcondition is

allowed to override and contradict the superclass postcondition could be considered, since this is common practice in OO programming (cf, redefinition of $bodyCondition$ on p. 101 of [42]). However the subclass would not then be substitutable with regard to the superclass.

2.7 Class theory of C

In the class theory Γ_C of a class C we define the (finite) set $\overline{C}$ of existing objects of C as an attribute of type $\mathbb{F}(@C)$ where $@C$ is the type of all possible instances of C. Initialisation of a C object is carried out at object creation, likewise termination takes place at object destruction:

(CI):

$$\forall a : @C \cdot create_C(a) \supset init_C(a)$$
$$\forall a : @C \cdot kill_C(a) \supset terminate_C(a)$$

The constant $self(@C) : @C$ is defined as a constant attribute (i.e., an attribute which is not in the write frame of any action). $self$ is the identity function:

$(SelfD)$:

$$\forall a : @C \cdot self(a) = a$$

$exists_C$ expresses that an object exists:

$(ExistsD)$:

$$\forall a : @C \cdot (exists_C(a) = TRUE) \equiv (a \in \overline{C})$$

If an attribute att of C is stereotyped as an $\{identity\}$ attribute, then the axiom

$(IdenD)$:

$$\forall a1, a2 : \overline{C} \cdot att(a1) = att(a2) \implies a1 = a2$$

is included in Γ_C. Likewise if there is a group of attributes which together form a compound primary key (a single identity constraint is attached to all elements of the group).

In particular the $number$ attribute of $Telephone$ is an identity attribute, so satisfies the axiom:

$$\forall a1, a2 : \overline{Telephone} \cdot number(a1) = number(a2) \implies a1 = a2$$

2.8 Subsystem theories

If class C uses class D as a supplier, i.e., there is an association directed from C to D, then Γ_D and Γ_C are combined together into the theory Γ_S of the subsystem of D and C together with their linking association, and we connect the actions denoting calls with the actual invoked operations:

(RSC):

$$\forall a : @D \cdot invoke_n(a, e) \supset a.n(e)$$

Table 10 Additional axioms for associations

Association	Additional axioms
$A_*\text{–}^r_*B$	$\forall a : \overline{A} \cdot r(a) \in \mathbb{F}(\overline{B})$
$A_{0..1}\text{–}^r_*B$	$\forall a : \overline{A} \cdot r(a) \in \mathbb{F}(\overline{B})$
	$\forall a1, a2 : \overline{A} \cdot r(a1) \cap r(a2) \neq \{\} \implies a1 = a2$
$A_1\text{–}^r_*B$	$\forall a : \overline{A} \cdot r(a) \in \mathbb{F}(\overline{B})$
	$\forall a1, a2 : \overline{A} \cdot r(a1) \cap r(a2) \neq \{\} \implies a1 = a2$
	$\forall b : \overline{B} \cdot \exists a : \overline{A} \cdot b \in r(a)$
$A_*\text{–}^r_1B$	$\forall a : \overline{A} \cdot r(a) \in \overline{B}$
$A_{0..1}\text{–}^r_1B$	$\forall a : \overline{A} \cdot r(a) \in \overline{B}$
	$\forall a1, a2 : \overline{A} \cdot r(a1) = r(a2) \implies a1 = a2$
$A_1\text{–}^r_1B$	$\forall a : \overline{A} \cdot r(a) \in \overline{B}$
	$\forall a1, a2 : \overline{A} \cdot r(a1) = r(a2) \implies a1 = a2$
	$\forall b : \overline{B} \cdot \exists a : \overline{A} \cdot b = r(a)$

Table 11 Axioms for association constraints

Association	Additional axiom
$A_*\text{–}^r_*B$	$\forall a : \overline{A}; b : \overline{B} \cdot b \in r(a) \implies a.(b.Inv)$
$A_{0..1}\text{–}^r_*B$	The same
$A_1\text{–}^r_*B$	The same
$A_*\text{–}^r_1B$	$\forall a : \overline{A} \cdot a.(r(a).Inv)$
$A_{0..1}\text{–}^r_1B$	The same
$A_1\text{–}^r_1B$	The same

and

(RCC):

$$\forall a : @D \cdot create\_invoke_D(a) \supset create_D(a)$$

These model synchronous invocations with no communication delays between client and supplier. The properties could be generalised to deal with distributed systems, for example by asserting:

$$\forall i : \mathbb{N}_1 \cdot \exists j : \mathbb{N}_1 \cdot \uparrow(invoke_n(a, e), i) = \leftarrow(n(a, e), j)$$

and that $invoke_n(a, e)$ terminates when a result message is received from a for this request.

When $invoke_n(objs, e)$ is used with a set $objs$ of objects, it is interpreted as a concurrent invocation of each of the individual object operations:

$(MRSC)$:

$$invoke_n(objs, e) \supset \|_{a:objs} n(a, e)$$

Additional axioms may be required in Γ_S to define the properties of the association from C to D, depending on its multiplicity at the C end (Table 10).

If a constraint Inv is attached to the association between a class A and a supplier class B, then an axiom expressing its meaning is included in Γ_S, depending on the form of association (Table 11). Inv may involve features of both A

Table 12 Axioms for bidirectional associations

Association	Additional axiom
A^{ar}_{*}–$^{br}_{*}$B	$\forall a : \overline{A}; b : \overline{B} \cdot a \in ar(b) \equiv b \in br(a)$
A^{ar}_{1}–$^{br}_{*}$B	$\forall a : \overline{A}; b : \overline{B} \cdot a = ar(b) \equiv b \in br(a)$
A^{ar}_{*}–$^{br}_{1}$B	$\forall a : \overline{A}; b : \overline{B} \cdot a \in ar(b) \equiv b = br(a)$
A^{ar}_{1}–$^{br}_{1}$B	$\forall a : \overline{A}; b : \overline{B} \cdot a = ar(b) \equiv b = br(a)$

and B. The notation $a.(b.Inv)$ means b is substituted into all new $@B$ parameter slots in the class version of Inv, and a into all new $@A$ parameter slots.

In the case of bidirectional associations, there are properties relating the two directions (Table 12). The case of 0..1 multiplicity at an association end produces the same axioms as for $*$.

The semantics of other forms of association between classes can also be expressed in this semantics, by transforming them into simpler constructs. Association classes are modelled as a class plus associations (Fig. 3). The new axiom

$$(AssocClass): \quad \forall r1, r2 : \overline{A\_B} \cdot a(r1) = a(r2) \wedge b(r1) = b(r2) \implies r1 = r2$$

holds in such a subsystem.

Qualified associations are also modelled by introducing a new intermediate class (Fig. 4). Here, $br[xval]$ is interpreted by $(cr|x = xval).br1$ in the new model.

Likewise we can formalise the two key properties of composite aggregations [42, p. 38]. The deletion propagation property of a composition (Fig. 5) is expressed by:

$$\forall w : \overline{W} \cdot kill_W(w) \supset \|_{p \in pr(w)} kill_P(p)$$

The property that there are no object-level loops in the extent of a composite that is a self-association (on a class W) can be expressed as:

$$\forall w : \overline{W} \cdot w \notin \bigcup_{n:\mathbb{N}_1} pr^n(w)$$

In the client/supplier construction, if D and C are the same class (the case of a self association), Γ_S is simply Γ_C extended with the additional axioms for any self-associations on C.

A similar theory composition is used in the case that C inherits from D, i.e., there is $g : Generalization$ with $g.general = D$ and $g.specific = C$. We include the axioms

$$(InheritD): \quad \overline{C} \subseteq \overline{D}$$

in Γ_S, and identify $@C$ and $@D$. This ensures that attributes and operations of D can also be applied to elements of $@C$. Notice that if $a \in \overline{C}, a.m(e)$ is required to obey the behaviour of both the C and D definitions of m. In addition, due to the frame axioms, operations of the subclass can only modify data of the superclass by invoking (co-executing with) update operations of the superclass which have that data in their frames. This condition ensures the subtyping principle of Liskov [37].

In the general case of several inheritance relationships, all classes concerned are represented in a single subsystem theory, only the classes C without superclasses are represented by a type $@C$. When forming the theory of a system involving both inheritance and associations, the inheritance construction should be applied to form a composed subsystem theory before the clientship construction.

Other cases of subsystems arise if there are constraints attached to sets of associations. In this case the collection of connected classes forms a subsystem, and the association constraint is expressed semantically in this subsystem.

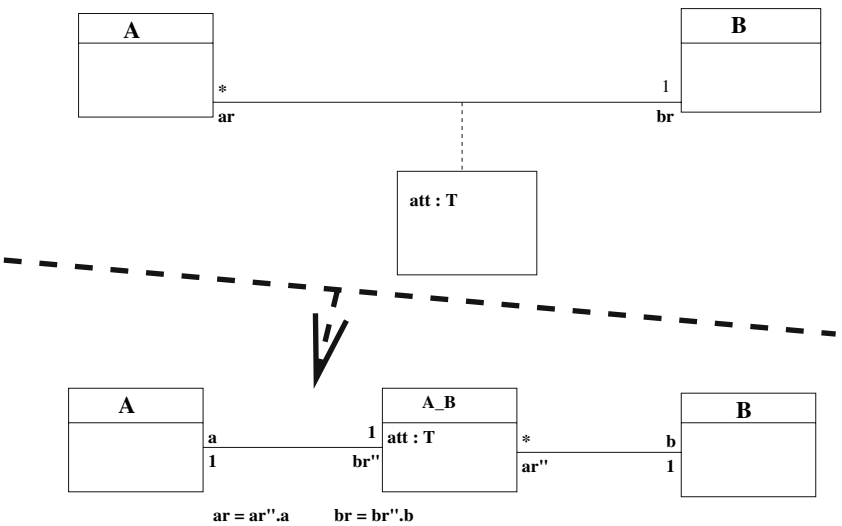


Fig. 3 Transformation of association classes to associations

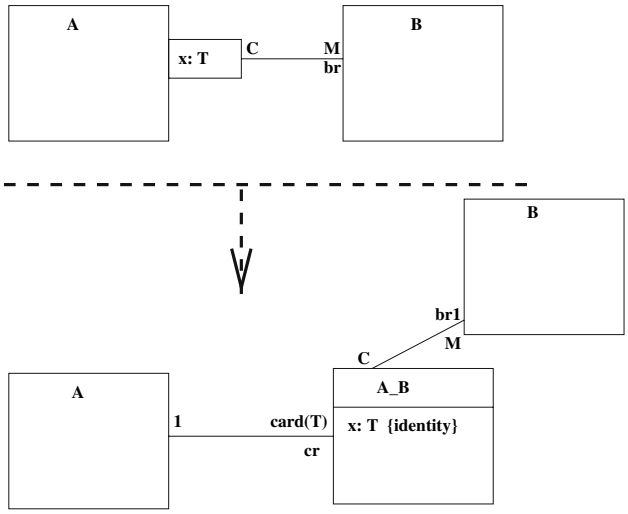


Fig. 4 Removing a qualified association

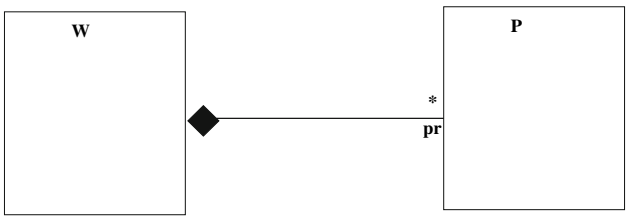


Fig. 5 Composite aggregation association

In addition, if a class A is a superclass of classes $A_1, \ldots, A_n$, then these classes should all be represented in a single subsystem whose theory extends Γ_A and each Γ_{A_i}. If A is $\{abstract\}$ then the axiom

$$(AbsD): \quad \overline{A} = \overline{A_1} \cup \cdots \cup \overline{A_n}$$

is added to this subsystem theory.

3 State machine semantics

The semantics of protocol and behaviour state machines for a class C are incorporated into the instance and class theories of C. This enables semantic checks of the consistency of the state machine models compared to the class diagram model.

3.1 Protocol state machines

In the case of a protocol state machine SC of C (Fig. 6):

1. The set of states is represented as a new enumerated type $State_{SC}$, and a new attribute c_state of this type is added to $\mathcal{I}_C$. The axiom

$$c\_state \in State_{SC}$$

holds. Local attributes of the state machine are represented as attributes of $\mathcal{I}_C$.
2. We specify the initialisation $c\_state := initial_{SC}$ of this attribute to the initial state of SC. This initialisation is invoked by $init_C$ ("When an instance of a behavioured classifier is created, its classifier behaviour is invoked", [42, p. 420]).
3. Each transition tr from a state src to a state trg, triggered by $m(x)$, with guard G and postcondition $Post$, is represented as an additional pre/post specification of m [42, p. 521]:

$$(c\_state = src \ \wedge \ G')$$

is added as an additional disjunct of the precondition of $m(x)$, and

$$(c\_state = src \ \wedge \ G')@pre \implies (c\_state = trg \ \wedge \ Post')$$

as an additional conjunct of the postcondition. Axiom (OpD) applies with these extended conditions. The write frame of m is extended to include c_state and any attributes or roles in a writable modality in $Post$.
Only operations with at least one transition in the state machine have c_state in their write frame—other operations are assumed not to change the state [42, p. 521].
4. State invariants Inv_s have the semantics:

$$(StateInv): \quad exists_C = TRUE \wedge c\_state = s \implies Inv_s'$$

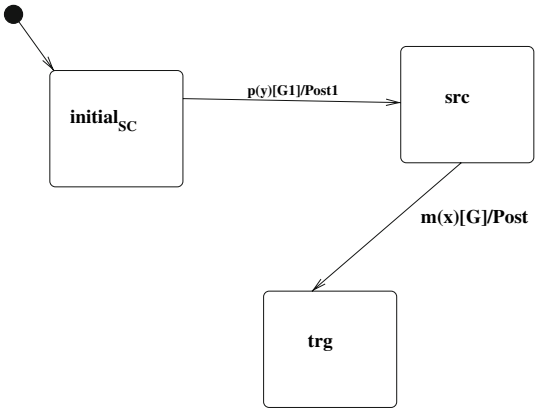


Fig. 6 Protocol state machine

An alternative modelling approach would be to use actions to represent individual transitions [35]. However this approach would not correspond so closely to formalisation in B, since an operation of a B module cannot invoke an operation of the same module.

In the UML documents there is an apparent inconsistency regarding the time at which guards are evaluated: "the guard is evaluated before the transition is triggered" [42, p. 556] and "the [guard] expression is evaluated at the moment the transition attached to the guard is attempted" [43, Sect. 12.11]. The latter corresponds correctly with the equivalence of transition guards and preconditions (for protocol state machines), and with our semantic interpretation. In practical implementation, the guard may be evaluated before the transition is selected and starts executing—however its truth value should not change over this interval.

For the telephone specification, the behaviour of most of its operations are derived from the protocol state machine. For example, $putDown$ has precondition

$$status = connected \; or \; status = calling \; or \; status = offhook$$

and postcondition

$$(status = connected \; or \; status = calling \; or \; status = offhook)@pre \implies status = idle$$

Further actions need to be added to this operation in order that this operation maintains the invariants, as described in Sect. 4.2.

3.2 Behavioural state machines

In the case of a behavioural state machine SC of C, transitions have an action which executes when the transition is taken, instead of a postcondition. The transition actions $acts$ are sequences

$$obj_1.op_1(e_1); \ldots; obj_n.op_n(e_n)$$

of operation calls on supplier objects, sets of supplier objects, or on the $self$ object. They are special cases of statements according to the meta-model of Fig. 14, and have a direct interpretation as composite actions $acts'$ in RAL:

$$obj_1'.op_1(e_1'); \ldots; obj_n'.op_n(e_n')$$

where the obj_i' and e_j' are the interpretations of these expressions in RAL.

In addition to state invariants, there may be entry actions to states.

The axiomatic representation of a behavioural state machine is:

1. The set of states is represented as a new enumerated type $State_{SC}$.
2. A new attribute c_state of this type is added to $\mathcal{I}_C$, together with the initialisation $c\_state := initial_{SC}$ of this attribute to the initial state of SC. An entry action $entry_{initial_{SC}}$ co-executes with this update, if specified. Local attributes of the state machine are represented as attributes of $\mathcal{I}_C$.
3. The transitions t_i, $i : 1..k$, from states src_i to states trg_i, triggered by $m(x)$, with guard G_i and actions $acts_i$, are represented as an additional operational specification $Code_m$ of m:

$$\begin{array}{l}(BSCOpP): \\ \quad \alpha(x) \supset if \; (exists_C = TRUE \wedge c\_state = src_1 \wedge G_1') \\ \quad then \; acts_1'; c\_state := trg_1 \\ \quad else \; if \; \ldots . \\ \quad else \; if \; (exists_C = TRUE \wedge c\_state = src_k \wedge G_k') \\ \quad then \; acts_k'; c\_state := trg_k\end{array}$$

   where α represents m, and any entry action of trg_i is included at the end of the $acts_i$ sequence.
   If there is already an existing procedural definition D_m of m in the class C, the complete definition of m is $D_m'; Code_m$ ([42, p. 422] we assume that an existing pre/post specification should however always refer to the entire span of execution of m).
4. The axioms $(StateInv)$.

The semantics defined here corresponds to the usual 'run to completion' semantics of UML state machines: a transition only completes execution when all of its generated actions do so [42, p. 546].

A behaviour state machine SC attached to an operation op defines an explicit algorithm for op. It is formalised as follows:

1. The set of states is represented as a new enumerated type $State_{SC}$.
2. A new attribute op_state of this type is added to $\mathcal{I}_C$ as a local variable of op, together with the initialisation $op\_state := initial_{SC}$ of this attribute to the initial state of SC. Local attributes of the state machine are represented as local variables of op.
3. The state machine yields the operational definition

$$\begin{array}{l}(BSCOpM): \\ \quad op(p) \; \supset \; pre \; exists_C = TRUE \; then \; Code_{op}\end{array}$$

   where $Code_{op}$ is:

$entry'_{initial_{SC}}$;
$op\_state := initial_{SC}$;
while $op\_state \neq terms_1 \wedge \ldots \wedge op\_state \neq terms_m$
do
  if $op\_state = src_1 \wedge G'_1$
  then
    act'_1; $entry'_{trg_1}$; $op\_state := trg_1$;
  else if ...
  else if $op\_state = src_k \wedge G'_k$
  then
    act'_k; $entry'_{trg_k}$; $op_state := trg_k$;

where the $terms_i$ are all the terminal (final) states of SC (i.e., states with no outgoing transitions), and the transitions of SC are $src_1 \rightarrow_{[G_1]/act_1} trg_1$ upto $src_k \rightarrow_{[G_k]/act_k} trg_k$.
Entry actions of a state must complete before the state machine is considered to properly enter the state ("before commencing a run-to-completion step, a state machine is in a stable state configuration with all entry ... activities completed" [42, p. 546]). An entry action will often be used to ensure that the state invariant holds.

4. The loop invariant of the above *while* loop is:

$$(op\_state = s_1 \implies Inv'_{s_1}) \wedge \ldots \wedge (op\_state = s_n \implies Inv'_{s_n})$$

where s_1 to s_n are all the states of SC.
This expresses that the local data of the particular execution instance of op is in a consistent state, satisfying a particular state invariant, when no transition or entry action is occurring.

3.3 Semantic profiles for state machine semantics

The UML semantics for protocol state machines does not specify whether transition guards are preconditions (sufficient conditions for valid execution of the actions of the transition and entering the target state) or are permission guards (necessary conditions for the transition to take place). In addition the meaning of an omitted transition for an operation is left open: it may mean that execution of the operation in that case is not permitted, is undefined in its effect, or has no effect.

Our semantics assumes only the minimal properties given in [42]:

1. If a logical case is missing for the transitions triggered by an operation, leaving a particular state, then the state machine gives no information about the effect of executing the operation in that state, and such an execution may not be possible [42, p. 519].
2. Operations which do not appear on the state machine are assumed to be state-insensitive in their behaviour, and not to modify the state [42, p. 521].

To express the concept of a guard as permission for an operation to execute, additional specification notation is needed. A clause $guard : G$ could be added as a new form of constraint to an operation $op(x : X)$. The clause would have the semantics

$$(OpG): \quad \forall x : X \cdot \forall i : \mathbb{N}_1 \cdot G' \circledcirc \uparrow (op(x), i)$$

Alternatively, we could define that the disjunction G of guards of the transitions leaving a particular state s and triggered by op, constitute a permission guard for op on that state (similar to the semantics of op being deferred in that state):

$$(PreAsGuard): \quad \forall x : X \cdot \forall i : \mathbb{N}_1 \cdot (c\_state = s \implies G') \circledcirc \uparrow (op(x), i)$$

The third alternative is that the operation is a 'skip' for these missing cases, as in behaviour state machines [42, p. 546]:

$$(SkipCase): \quad \forall x : X \cdot \forall i : \mathbb{N}_1 \cdot (c\_state = s \wedge \neg G') \circledcirc \uparrow (op(x), i) \implies (c\_state = s) \circledcirc \downarrow (op(x), i)$$

op is permitted to take place if G fails in state s, but then it does not change the state.

These three alternatives form three 'semantic profiles' which are alternative extensions of the basic semantics. Each constitutes some possible extra notations, and additional axioms. Developers should indicate which semantics they are adopting, and record this together with the models.

3.4 Generalisation of state machines and behavioural compatibility

Behavioural compatibility is the requirement that the specified behaviour of a superclass object should not be violated by a subclass object. In the UML 2 documents there are various partly conflicting statements about generalisation and behaviour:

1. "An operation may be redefined in a specialization of the featured classifier. This redefinition may specialize the types of the owned parameters, add new preconditions or postconditions ... or otherwise refine the specification of the operation." [42, p. 101]. The same page also asserts that *all* the preconditions of an operation

can be assumed by any of its implementations, and that operation behaviour is unspecified if *a precondition* fails to hold. This suggests a semantics in which preconditions are logically combined by "&", not "or". Section 11 of [43] contains a similar statement. But then adding preconditions (and restricting input parameter types) makes an operation *less defined* in a subclass than in the superclass – contrary to what is necessary for semantic subtyping. Despite the claim on p. 101 of [42] that both covariant and contravariant specialisation are semantic variation points, the previous statements imply covariant specialisation.
The $bodyCondition$ of an operation (for query operations) can be arbitrarily redefined in redefinitions of the operation, and so can violate superclass semantics.

2. If a generalization g has $g.isSubstitutable = true$, then the traces of $g.general$ are a subset of the traces of $g.specific$ [42, p. 67].
This contradicts the notion that preconditions can be strengthened in subclasses, if a guarded interpretation of preconditions is taken. In this interpretation, strengthening postconditions (to remove non-determinism) may also eliminate traces, even if preconditions are not changed.
For example, an operation $op1$ with postcondition $x = 0$ *or* $x = 1$ could be refined to have postcondition $x = 1$. But if operation $op2$ has precondition (guard) $x = 0$ the trace $op1; op2$ is no longer possible.
3. Various transformations on state machines are proposed on p. 548 of [42]: splitting a source state, adding states to a composite state, etc.
These transformations aim to preserve the capability of clients to invoke particular sequences of operations on the state machine, however the behaviour of these same sequences of operations may be unexpected to the client who only knows the superclass specification.
In particular, transitions should only have their target state replaced by a more specific state (corresponding to a strengthened postcondition) rather than an arbitrary state (as in [42]), and it is valid for transitions to be entirely replaced by a set of more specific transitions for the same trigger.

We propose the following solutions to these issues:

1. There should be explicit semantic profiles for different forms of generalisation, such as contravariance of operation input types (and weakening of preconditions), or invariance of input types, as used in Java.
A consistency rule constraining the $bodyCondition$ to be consistent with the operation postconditions should be defined.
2. Precise criteria for valid state machine specialisation transformations should be defined, based on an underlying semantics for these models. The $refinement$, $adequacy$ and $locality$ conditions are such criteria, as given below. They can be used to establish behavioural compatibility for the base semantics of protocol state machines described above.

To ensure semantic behavioural compatibility, we can define three syntactic conditions on the protocol state machine C of a subclass CC of a class AC and the protocol state machine A of AC [33,34]:

1. *Refinement*: For every state s of C, there is a state $\sigma(s)$ of A, and for every transition tr of C triggered by an operation of AC there is a transition $\sigma(tr)$ of A such that:
   (a) $\sigma(s)$ is initial in A if s is initial in C.
   (b) $\sigma(tr) : \sigma(s) \rightarrow \sigma(t)$ in A if $tr : s \rightarrow t$ in C.
   (c) tr and $\sigma(tr)$ have the same trigger.
   (d) $Post_{tr} \implies Post_{\sigma(tr)}$
   This means that any behaviour of C must satisfy the specifications of behaviour of A. σ is termed an *abstraction morphism*.
2. *Adequacy*: For each state s of A there is at least one state s' of C such that $\sigma(s') = s$. The disjunction of the state invariants of all such s' is equivalent to the state invariant of s. If s is initial in A, so is one of the s' in C.
   For each transition $tr : s \rightarrow t$ in A there are transitions $tr' : s' \rightarrow t'$ of C such that $\sigma(tr') = tr$, for every state s' such that $\sigma(s') = s$. The disjunction of guards of the tr' with source a particular such s' should be equivalent to the guard of tr.
   This means that behaviour defined in the superclass must also be defined in the subclass.
3. *Locality*: If a new operation op of CC has a transition $tr : cs \rightarrow ct$ in C for which $\sigma(cs) \neq \sigma(ct)$, or which modifies any data feature of AC, then op must (in terms of its effect on the data of AC and state of A) be expressible as a procedural combination of operations of AC.

These conditions can be formally deduced from the requirement that the semantics of CC together with C, as an RAL theory, is a theory extension of the theory of AC and A [34]. Refinement and adequacy also correspond to the usual definition of refinement in state-based system specification [1]. The locality condition is a consequence of the frame axiom of RAL, and corresponds to Liskov's composition requirement for new operations introduced in subclasses [37], it ensures that clients of the subclass do not see unexpected behaviour—new transitions and pathways between states not present in the superclass specification.

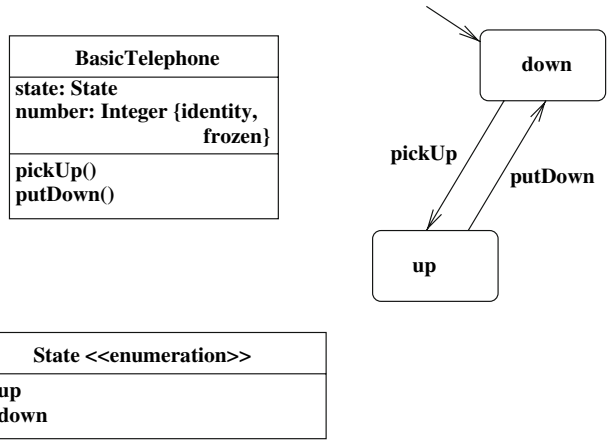


Fig. 7 Basic Telephone

These conditions also apply for the skip semantic interpretation of state machine semantics.

When comparing two behaviour state machines for behavioural compatibility, the condition 1(d) above is replaced by the requirement that

$$acts_{tr} \supset acts_{\sigma(tr)}$$

where the action lists include any entry actions of their target states.

As an example of behavioural compatibility using these rules, the $BasicTelephone$ of Fig. 7 is a valid generalisation of $Telephone$, via the abstraction mapping of $\sigma(idle) = down$, and $\sigma(s) = up$ for the other states of $Telephone$, and with the consequent mapping of transitions.

Using the above conditions, we can verify that many generalisation transformations on state machines are semantically correct:

1. A postcondition of a transition can be strengthened.
2. A transition can be split into several cases from the same source state, with guards logically partitioning the original guard, and with targets equal to the original target.
3. A new region can be added to a concurrent composite state, provided no transition in this region has a trigger in common with the existing composite state. It is valid to refer to the state of this region in the transition guards of other regions—provided that all possible states of the new region are alternatives in the guards of the refined transitions from any particular state (Fig. 8).

In case 3, σ discards the new region.

The adequacy, refinement and locality conditions do ensure that the theory of a specialised protocol state machine C of a subclass CC of a class AC satisfies all the axioms of a generalised protocol state machine A of AC, under the interpretation of c_state_A as $\sigma(c\_state_C)$:

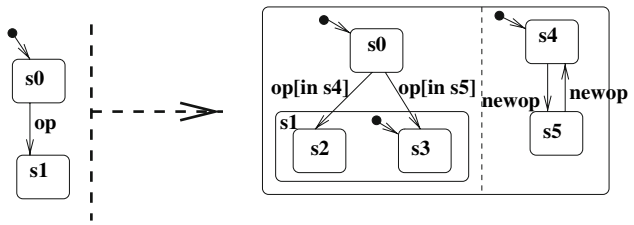


Fig. 8 Adding a region

1. We need to prove that the interpretation

   $$\sigma(c\_state_C) \in State_A$$

   of the state machine A axiom is provable, which is clear from the typing of σ.
2. The interpretation of the initialisation axiom is

   $$init_{CC} \supset \sigma(c\_state_C) := initial_A$$

   But the initialisation of c_state_C is to the initial state of C, and $\sigma(initial_C) = initial_A$, so this also holds.
3. We need to show that the pre-post specifications of operations in A can be proved from those of C.
   It is sufficient to consider abstract transitions one-by-one, the pre/post behaviour for each of these has the form

   $$(c\_state_A = s \wedge G) \circledcirc \uparrow(op, i) \implies ((c\_state_A = s \wedge G) \circledcirc \uparrow(op, i) \implies (c\_state_A = t \wedge Post) \circledcirc \downarrow(op, i))$$

   for a transition $tr : s \rightarrow_{[G]/Post} t$ triggered by op.
   This simplifies to:

   $$(c\_state_A = s \wedge G) \circledcirc \uparrow(op, i) \implies (c\_state_A = t \wedge Post) \circledcirc \downarrow(op, i)$$

   The interpretation of this axiom should be provable from corresponding axioms of C. Due to adequacy we know there are concrete states $s_1, \ldots, s_n$ with $\sigma(s_j) = s$ each j. Also, for each of these, there are transitions $tr_{j,1}, \ldots, tr_{j,n_j}$ which abstract to tr, and whose guards partition G. This means that the interpretation

   $$\sigma(c\_state_C) = s \wedge G$$

   of the abstract precondition due to tr is equivalent to the disjunction of the concrete preconditions

   $$c\_state_C = s_j \wedge G_{j,k}$$

of all the $tr_{j,k}$ which abstract to tr. Thus the interpretation of the abstract precondition implies the concrete precondition. The concrete postconditions of the $tr_{j,k}$ have the form:

$$c\_state_C = t_{j,k} \wedge Post_{j,k}$$

But $\sigma(t_{j,k}) = t$ for such poststates, and $Post_{j,k} \implies Post$, so the result follows.
Informally, we have shown that each possible behaviour of the system defined by C satisfies the specification of A.

4. For state invariants we have required that

$$Inv_{s_j} \implies Inv_s$$

for each s_j in the states of C with $\sigma(s_j) = s$, so this axiom also follows.

Locality ensures that the frame axioms of AC are true in interpreted form in CC.

The UML-RSDS tool automatically tries to construct an abstraction morphism σ from a subclass state machine C to a superclass state machine A. If such a mapping exists it will then check it for the adequacy condition.

Other problems with state machine semantics are identified in [15], in particular for history states, transition priorities and entry/exit points. Alternative semantic profiles may need to be defined to accommodate the different interpretations of these elements.

4 Application of the semantics

In this section we show how the semantics for UML-RSDS given above can be used to support extensions of UML to specify real-time system properties, and to support verification and refinement.

4.1 UML extensions for concurrency and real-time

UML contains some notations for referring to detailed communication and real-time behaviour, such as the $OclMessage$ type [43, Sect. 7.7]. However these mechanisms are not complete. For example whilst $OclMessage$ represents messages sent from an object, there is no specification facility within OCL to represent or express properties of messages *received* by an object, or the temporal and synchronisation properties of the operations invoked by these messages.

In Appendix B.2 we give definitions for real-time and concurrency notations:

1. The times $\leftarrow(op(p), i)$, $\rightarrow(op(p), i)$, $\uparrow(op(p), i)$, $\downarrow(op(p), i)$ of sending, request arrival, activation and termination of an action $op(p)$.
2. Corresponding counters $\#snd(op(p))$, $\#req(op(p))$, $\#act(op(p))$ and $\#fin(op(p))$.
3. Expressions derived from these such as $delay(op(p), i)$ and $\#active(op(p))$.

We can therefore extend OCL with such expressions, for the update operations op of each class. This allows the expression of UML semantics such as *sequential* for an operation $op(p : PT)$, disallowing multiple concurrent executions of the operation:

$$\forall p : PT \cdot \#active(op(p)) \leq 1$$
$$\forall p : PT \cdot \#active(op(p)) > 0 \implies (\forall q : PT \cdot q \neq p \implies \#active(op(q)) = 0)$$

This becomes an axiom of the instance theory of the class that owns op.

Protocols such as readers-writers can be expressed, and implemented using design patterns such as synchroniser objects [24].

Periodic behaviour can be specified by $periodic(op, \delta, \epsilon, T)$:

$$\forall i : \mathbb{N}_1 \cdot \uparrow(op, i) \leq T + \delta * i + \epsilon \wedge \uparrow(op, i) \geq T + \delta * i$$

meaning that op executes every δ time units after an initial time T, with a maximum lag time of ϵ.

Different policies for ordering the input event pool of an object can be specified:

1. "First come first served":

$$\forall i, j : \mathbb{N}_1 \cdot \rightarrow(op, i) \leq \rightarrow(op, j) \implies \uparrow(op, i) \leq \uparrow(op, j)$$

2. "Shortest job first", for some function f of the parameter data:

$$\forall i, j : \mathbb{N}_1 \cdot f(x) < f(y) \wedge \rightarrow(op(x), i) < \uparrow(op(y), j) \implies \uparrow(op(x), i) \leq \uparrow(op(y), j)$$

Other semantic variations on input pool processing [42, p. 420] can be defined in a similar manner.

The UML concept of a time observation is represented by the attribute now in RAL. The use of *change expressions* and *time triggers* as triggers of state machine transition triggers can also be represented in our semantics.

A relative time trigger $after\ T$ on a transition away from a state s represents that the transition should be taken if the state has remained occupied for at least T time units continuously:

$$duration(c\_state = s) \circledast now \geq T$$

where for a predicate P and time t:

$$duration(P)\circledast t \;=\; max(\{0\} \cup \{x : TIME \mid \forall y : TIME \cdot t - x \leq y \wedge y \leq t \implies P \circledcirc y\})$$

Section 15.3.13 of [42] gives a slightly ambiguous wording for the definition of this trigger, but the above seems to be the intended semantics.

Development techniques for distributed and real-time systems used in VDM^{++} could be adopted for this extended UML [22,51].

4.2 Semantic analysis

A number of consistency and completeness properties of a UML-RSDS specification can be defined. Checks that these properties hold can be used to detect errors in formulation of a specification, and to ensure the consistency of the corresponding theories representing the semantics of the specification.

Some semantic rules for a class C are as follows, in general these can only be checked using a proof tool:

1. The class invariant must be satisfiable, i.e., there must exist at least one combination of attribute/role values of C in which Inv_C is true:

$$\mathcal{I}_C \vdash \exists v_1 : T_1; \ldots; v_n : T_n; rv_1 : DT_1; \ldots; rv_m : DT_m \cdot Inv_C'[v/att, rv/role]$$

   where the $att_i : T_i$ are the semantic representations of the attributes (including inherited attributes) of C, and the $role_j : DT_j$ represent the roles of C.
   This also confirms that the explicit invariant of C is consistent with superclass invariants, as these are all conjoined to form the complete class invariant.
2. The initialisation of a class establishes the invariant:

$$[init_C]Inv_C'$$

3. A consistency requirement for a state machine for C is that there are no two different transitions from the same state triggered by the same operation which have overlapping guards [42, p. 547]:

$$\mathcal{I}_C \vdash \forall x : X'; i : \mathbb{N}_1 \cdot (c\_state = s \wedge G_1' \implies \neg G_2') \circledcirc \uparrow (op(x), i)$$

   for the guards G_1 and G_2 of any two transitions for $op(x : X)$ from state s.
4. The state machine is *complete* if, for any operation op which has at least one transition in the state machine, for each state s from which there is a transition for op, the disjunction of the guards on the transitions for op from s is equivalent to $true$. In the case that an explicit permission guard is defined for op from s, the disjunction should be equivalent to this guard.
5. If an explicit algorithm is provided for an operation op by a behavioural state machine, then this algorithm $Code_{op}$ must satisfy the pre/post constraints given for the operation:

$$[pre\ exists_C = true \wedge Pre_{op}'\ then\ Code_{op}]Post_{op}'$$

   If $Code_{op}$ includes calls of other operations, then the preconditions of these operations should be true at the point of call.
6. The actions of each state machine transition should establish the invariant of the target state, if any:

$$Inv_s' \wedge G' \implies [op(x); acts'; entry_t']Inv_t'$$

   for a transition $s \rightarrow_{op(x)[G]/acts} t$ of a behaviour state machine for an object. For protocol state machines, the postcondition of a transition should be consistent with the invariant of its target state.
7. Definedness obligations: the invariant of a class should always be well-defined (not contain applications of functions to elements outside their domain, such as division by zero), and the precondition of an operation should ensure that the postcondition or code definition of this operation is well-defined, likewise for transition guards and transition actions.

A translation from UML-RSDS into the formal specification language B [1] is used to support this semantic analysis, using the proof tools provided for B in the BToolkit [9] or B4Free [4].

The properties 1, 2, 5 and 6 above can be checked as part of the standard proof obligations of internal consistency of a B component which represents the semantics of a UML class together with its protocol state machine. For a correct component, this proof is usually quite effective, the B tools are normally able to prove 90% of verification conditions of internal consistency for such components [7]. Failed proof can also be useful to identify semantic errors in a component (and the source UML specification, in our case).

The translation to B corresponds very closely to the RAL semantics. In particular, individual B modules are used to represent instance, class and subsystem theories within a model, allowing a modular approach to verification and a potential reduction in proof complexity. Attributes of a RAL theory become variables of the B module representing the theory, and actions of the theory become operations of the B module.

Table 13 Correspondence of semantics and B

Semantics element e	B element $e^{\#}$
$\mathbb{N}$	NAT
$\mathbb{Z}$	INT
$\mathbb{B}$	BOOL
$@C$	C_OBJ
$\overline{C}$	cs
$att(a : @C) : T$	$att : cs \rightarrow T^{\#}$
$att : T$	$att : T^{\#}$

Table 13 shows the correspondence between elements of the semantics of a UML-RSDS model M, and elements of the B specification module generated from M.

$T^{\#}$ represents the interpretation of T in B. There are no types of rational or real numbers in B, so $\mathbb{R}$ cannot be represented in B, and $TIME$ will be interpreted by $\mathbb{N}$. The semantics of OCL expressions in B is defined using the set-theoretic denotations given in Sect. 2.4. There are only small differences in the denotation of OCL expressions in RAL and in B: sum is calculated by a $SIGMA$ operator in B, and implicit promotion of elements to collections (Table 9) is not supported. On formulae without real numbers, modal or RTL operators, RAL reduces to the same classical first order logic as B. This means that:

$$S' \vdash_{RAL} \varphi' \equiv S^{\#} \vdash_B \varphi^{\#}$$

holds, for any set $S \cup \{\varphi\}$ of such sentences in UML-RSDS OCL, where in RAL we assert $TIME = \mathbb{N}$.

In the case of the telephone specification, the B module derived from the class diagram and state machine is as follows (corresponding to the instance theory $\mathcal{I}_{Telephone}$):

```
MACHINE Telephone SEES Int_TYPE
SETS Status = {idle, offhook, calling,
  connected}
VARIABLES status, number, dialing,
  connectedTo, lastCalled
INVARIANT
  status : Status & number: INT &
  dialing : INT & connectedTo : INT
  lastCalled : INT & (dialing /= 0  =>
  status = calling) &
  (status /= calling  =>  dialing = 0) &
  (status /= connected  =>
             connectedTo = 0) &
  (connectedTo /= 0  =>
                  status = connected)
INITIALISATION status := idle
OPERATIONS
  setstatus(statusx) =
  PRE statusx : Status
  THEN status := statusx ||
   IF statusx /= connected THEN
                 connectedTo := 0 END ||
   IF statusx /= calling THEN
                 dialing := 0 END
  END;

  pickUp =
  PRE status = idle
  THEN status := offhook
  END;

  putDown =
  PRE status = connected or status =
  calling or status = offhook
  THEN status := idle
  END;

  ... /* other operations of
                           Telephone */
END
```

The initialisation must establish the invariants Inv:

$$(status \neq calling \implies dialing = 0) \wedge$$
$$(status \neq connected \implies connectedTo = 0)$$

But initialisation only performs the action

$$status := idle$$

which gives the proof obligation $Inv[idle/status]$, ie:

$$dialing = 0 \wedge connectedTo = 0$$

This is not provable from the existing specification, which is therefore incomplete. We can satisfy this requirement by initialising $dialing$ and $connectedTo$ to 0.

Likewise, analysis of the operation $putDown$ identifies inconsistency between the behaviour derived from the state machine, and that required by the invariants: again the obligation

$$dialing = 0 \wedge connectedTo = 0$$

is not provable from the B version of the operation, and this operation must be extended to:

```
putDown =
  PRE status = connected or status =
  calling or status = offhook
  THEN status := idle || dialing := 0 ||
  connectedTo := 0
  END
```

Another case of incompleteness, identified by animation, is the failure to set $lastCalled$ correctly. This should be set on entry to the calling state.

4.3 Model transformations

Model transformations for UML can be classified in five categories:

1. Quality improvement transformations.
2. Elaborations.
3. Refinements.
4. Abstractions.
5. Design patterns.

Model transformations can be formally specified as operations at the UML-RSDS metamodel level (using the metamodels given in Appendix A). For example, the transformation "Replace inheritance by association" removes an element $g : Generalization$ and creates a new $a : Association$ and $r1, r2 : Property$ such that

$$a.memberEnd = Sequence\{r1, r2\}$$
$$r1.classifier = g.general$$
$$r2.classifier = g.specific$$

and $r1$ has 0..1 multiplicity and $r2$ has 1 multiplicity.

To prove that a transformation is semantically correct, the semantics of the original untransformed model is compared with that of the transformed model. It should be possible to construct a theory interpretation (as in Appendix C) of the semantics of the original model into the theory of the new model, so that all the properties of the original model remain true in the new model, in interpreted form. This property of a transformation is critical if it is to be used reliably to carry out application development.

All of the above categories of transformations, except abstractions, will normally be provable as theory interpretations of the theory of the source/starting model in the theory of the target model. For abstractions the theory interpretation will go in the opposite direction.

We give examples of a quality improvement transformation (factoring out entry actions) and a refinement transformation (removing inheritance by amalgamating classes).

4.3.1 Amalgamate subclasses into superclass

This transformation amalgamates all features of all subclasses of a class C into C itself, together with an additional flag attribute to indicate which class the current object really belongs to. It is one strategy for representing a class hierarchy in a relational database.

An example of the transformation is shown in Fig. 9.

Constraints of the subclasses must be re-expressed as constraints of the amalgamated class, using the flag attribute, as illustrated in Fig. 9.

This transformation is related to the Collapsing Hierarchy refactoring of [18].

4.3.2 Introduce entry actions of a state

If all transitions $t_1, \ldots, t_n$ into a state s have the same final sequence act of actions, remove these actions from the transitions and add act as the first actions of the entry action of s.

This can be directly shown to give a theory interpretation (as the identity morphism) of the original model into

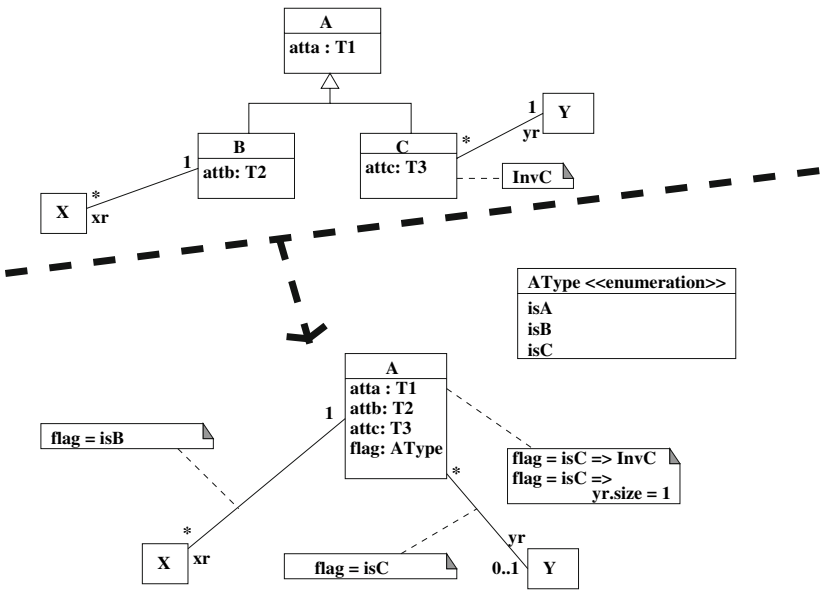


Fig. 9 Amalgamation of subclasses transformation

the new model, since the axioms $BSCOpP$ or $BSCOpM$ (depending on the form of state machine) in Sect. 3 will be unchanged between the new and old models, and no other axioms are affected by this transformation.

4.3.3 Design patterns

Design patterns can also be considered as model transformations: introducing the pattern results in a transformation of a model, for quality improvement or refinement purposes [50].

For example, consider the Facade pattern [19]. A generic facade with three client classes and two suppliers would be represented as a transformation from the original model which simply contains these five classes, to a new model where communication is via the facade class (right hand side of Fig. 10).

A particular application of Facade would extend this transformation by interpreting these general pattern components by the specific classes and associations in the system being considered.

The key invariant of Facade is that the new introduced class preserves the interconnections between the suppliers and clients:

$$\forall b \in \overline{B} \cdot \forall c3 \in \overline{C3} \cdot b \in br(c3) \equiv \exists f \in \overline{F} \cdot b \in br1(f) \wedge f \in fr(c3)$$

That is, the original br is implemented by the composition $fr; br1$ in the new model. Likewise for the other suppliers and clients.

The significant effect of the transformation is that invocations $br.op(x)$ in operation definitions of the clients $C1$, $C2$, $C3$ in the original model become invocations $fr.op(x)$ in the new model, and op on F is defined to call the original op in A or B.

Axioms of the original model hold in interpreted form in the new, where br is interpreted as $fr.br1$ and ar as $fr.ar1$: association constraints on $C1\_A$, etc., must now be expressed as constraints on both $C1\_F$ and F_A. Their syntax and semantics remain unchanged. The axioms OpP, RSC, $MRSC$, $BSCOpP$, $BSCOpM$ which define one operation in terms of others may be affected by this pattern if their right hand side involves one of the replaced supplier operations:

$$m(y) \supset \ldots br.op(x) \ldots$$

This axiom is interpreted as

$$m(y) \supset \ldots fr.br1.op(x) \ldots$$

in the new model. But this is provable from the corresponding axiom

$$m(y) \supset \ldots fr.op(x) \ldots$$

in the new model, since $fr.op(x) \supset fr.br1.op(x)$ and each of the statement composition operators is monotonic with respect to $\supset$ (Appendix B).

In the UML-RSDS tools, the pattern is identified as possibly relevant for a model if there are classes which are shared suppliers of two or more clients.

5 Extension of the semantics

The semantics of UML-RSDS can be extended to give a semantics for much larger subsets of UML 2.

1. State machines with OR and AND composite states can be reduced to equivalent state machines with only basic states, by flattening (cf., [12]), or by defining a state variable for each OR composite state/region and expressing transitions (including compound transitions) by pre and post-conditions on these state variables.
2. Exit actions of states in state machines can be expressed as initial actions of every transition that exits the state. Choice pseudostates can be replaced by normal states, with guarded outgoing completion-triggered transitions.

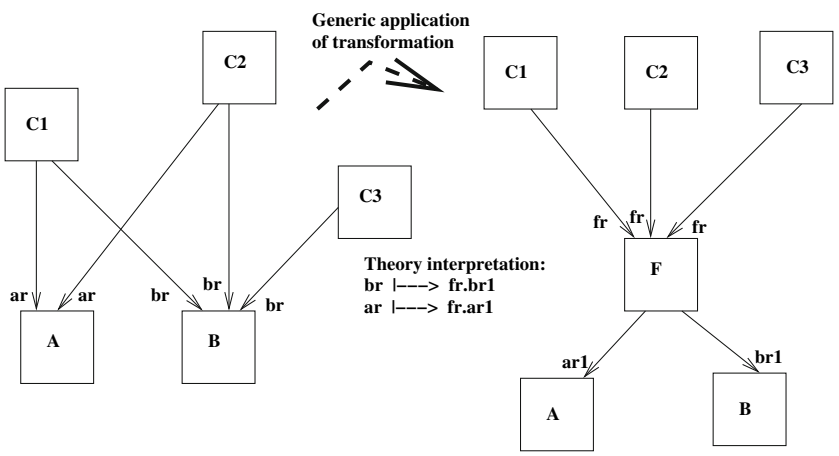


Fig. 10 Facade pattern application

Event deferral can be specified using permission guards to express that the operation of the event cannot start executing in the deferring state.
3. Class diagram constructs such as qualified associations and association classes can be replaced by semantically equivalent constructs in UML-RSDS (Sect. 2.8).

The semantics can also be extended directly to the use case and interaction notations of UML 2:

1. Use cases are treated as operations of a system, and may be defined using a behavioural state machine, as for operations of a class. They are semantically represented by action symbols in a system theory, and their behaviour is defined from their state machine as for operations of classes.
2. Interaction diagrams can be directly represented in our semantics as sets of constraints on event times $\leftarrow(op(x), i)$, $\rightarrow(op(x), i)$, and other times in the life histories of the objects whose lifelines appear in the interaction. Events to represent the return of a synchronous operation call as a message to the caller need to be introduced, as in [23]. Operators such as *par* can be defined in terms of these constraint sets (as union, in the case of *par*). The more precise sublanguage of the interaction notation defined in [26] can be given a semantics in this manner. Reasoning tools for RTL could potentially be used to support analysis of the semantics of interactions, as in [3].

6 Comparison

Other work on UML semantics has used the following approaches:

1. Expression of UML semantics (usually of a small part of UML) in a semantic representation outside of UML, using denotational [8,14,21,36,46], operational [12], axiomatic [27] or category-theoretic [49] semantics.
2. Metamodelling, representing UML semantics in terms of a small core UML notation together with OCL [11]. This is the approach used in the UML 2.0 Infrastructure and Superstructure documents, although in these many of the semantic definitions are informally expressed and not formalised in OCL (in some cases OCL is unable to express the definitions). Circularity in metamodelling still remains a problem for the semantics of OCL itself [10].

Denotational semantics as in [8,14,46] are very useful as an underpinning for axiomatic semantics, however they can be very complex and difficult to understand and implement directly, or to tailor to different semantic variation points of UML. For this reason we prefer to give an axiomatic semantics, which is closely linked to notations used for proof and semantic analysis (in classical mathematical and logical notation, or in B).

In comparison to the system model defined by the UML semantics project [8], we use a more abstract and general semantic representation, avoiding low-level details of data storage mechanisms.

In general, the strategy of restricting to a subset of UML is necessary, since some of its notations, such as activities, have unclear semantics. However the subset should itself be a useful part of UML, so that the results of semantic analysis can be expressed in a comprehensible manner to software developers. Translating interactions to state machines, or flattening state machines, should be avoided for this reason. Our semantics can represent directly the semantics of interactions and unflattened state machines.

As [40] points out, the self-referential metamodelling approach can result in a semantics which fails to provide a consistent interpretation for the terms of UML. Instead, we have taken the first approach and given a semantics of an essential core of UML in a formalism which is entirely independent of UML, and based on well-established mathematical logic and set theory. In decomposing the semantics of UML models into theories linked by morphisms, we are also using a category-theoretic approach.

An approach closely related to ours is the formalisation of UML and OCL in PVS [27]. This deals with a restricted subset of class diagrams (association ends have maximum multiplicity 1). The expression of the semantics in PVS can be complex and difficult to relate to the UML source models, but automated proof can be applied as with model checking.

Our semantics represents the *extension* of a class C as a element $\overline{C}$ of the semantics, but does not represent the *intension* of C [40] as an element within the semantics. Instead this is represented as the theory $\mathcal{I}_C$. This means that the semantics of concepts such as $leaf$ and $root$ classes and operations cannot be expressed in our semantics. $private$ and $public$ modalities of features are also not represented. However, the correctness of models with regard to such constraints can be effectively checked by diagram editing and syntax-level analysis tools, so the inability to represent such concepts in our semantics does not impair the verifiability of UML-RSDS models.

7 Conclusions

We have shown that it is possible to construct a semantics for a substantial subset of UML, in a semantic representation which is independent of UML, and in a modular manner. The semantics is geared toward practical reasoning, and is designed for convenient use with the B verification

Table 14 UML-RSDS constraint syntax

$< value >$	::=	$< ident >$ |	Variable expression
		$< number >$ | $< string >$ |	Primitive literal
		$< boolean >$	expressions
$< objectref >$	::=	$< ident >$ |	
		$< objectref >.< ident >$ |	Navigation expression
		$< objectref >$ | ($< expression >$)	Select expression
$< arrayref >$	::=	$< objectref >$ |	
		$< objectref >[< value >]$	At expression
$< factor >$	::=	$< value >$ |	
		{ $< valueseq >$ } |	Collection literal
		Sequence{ $< valueseq >$ } |	expressions
		$< arrayref >$ |	
		$< factor >$ op1 $< factor >$	Infix binary operation call (1)
$< expression1 >$	::=	$< factor >$ op2 $< factor >$	Infix binary operation call (2)
$< expression >$	::=	$< expression1 >$ |	
		($< expression >$) |	
		$< expression1 >$ op3 $< expression >$	Infix binary operation call (3)
$< invariant >$	::=	$< expression >$ |	
		$< expression >$ => $< expression >$	

tools. Based on the semantics, a large toolset has been developed to support 'constraint-driven development' using UML, enabling the implementation of systems to be generated with a high degree of automation from their specifications in UML.

A Formal syntax of UML-RSDS

UML-RSDS is a subset of UML 2.0 (with minor variations). It provides a core class diagram and state machine notation in terms of which many other features of these notations can be defined.

A.1 OCL notation in UML-RSDS

The OCL language adopted in UML-RSDS has the syntax shown in Table 14. A *valueseq* is a comma-separated sequence of values. A factor level operator $op1$ can be:

1. $+, -, *, /, div, mod$
2. \/, /\ (union and intersection, also written as $\cup$ and $\cap$), $\frown$.

A comparator operator $op2$ is one of =, /=, <, >, <=, >=, :, <:, /:, / <:. A logical operator $op3$ is one of &, *or*.[1] The operator precedence rules of OCL are used [43, Sect. 7.4.7]. Identifiers are either class names, function names, features (attribute, operation or role names), elements of enumerated types, or represent variables or constants. Variables are implicitly universally quantified over the entire formula. Operations can also be written with parameters as $op(p_1, \ldots, p_n)$, etc. In contrast to OCL, we do allow operations to be invoked on collections of objects—this is interpreted as a concurrent (order undefined) execution of the operation on the individual objects. Reference to the value of an expression e at a precondition of an operation can be made in the postcondition using the notation $e@pre$ as usual.

[1] To avoid confusion we use these symbols for the syntax of UML-RSDS, and the symbols $\wedge, \vee$ for the corresponding semantic operators in RAL.

A.2 UML-RSDS class diagrams

Figure 11 shows the metamodel used to formally define UML-RSDS class diagrams as a subset of UML 2.0.

$StructuralFeature$ also inherits from $MultiplicityElement$. The metaclasses $Extension$, $ExtensionEnd$ and $Stereotype$ are defined as in the Profiles package of UML 2.0 [42, Sect. 18].

Only a subset of UML class diagram notation is currently used in UML-RSDS:

- Classes cannot be nested.
- Qualified associations and aggregation are omitted.
- Associations are binary:

$$memberEnd.size = 2$$

Association ends are never static:

$$memberEnd.isStatic = false$$

and there is no specialisation of associations.

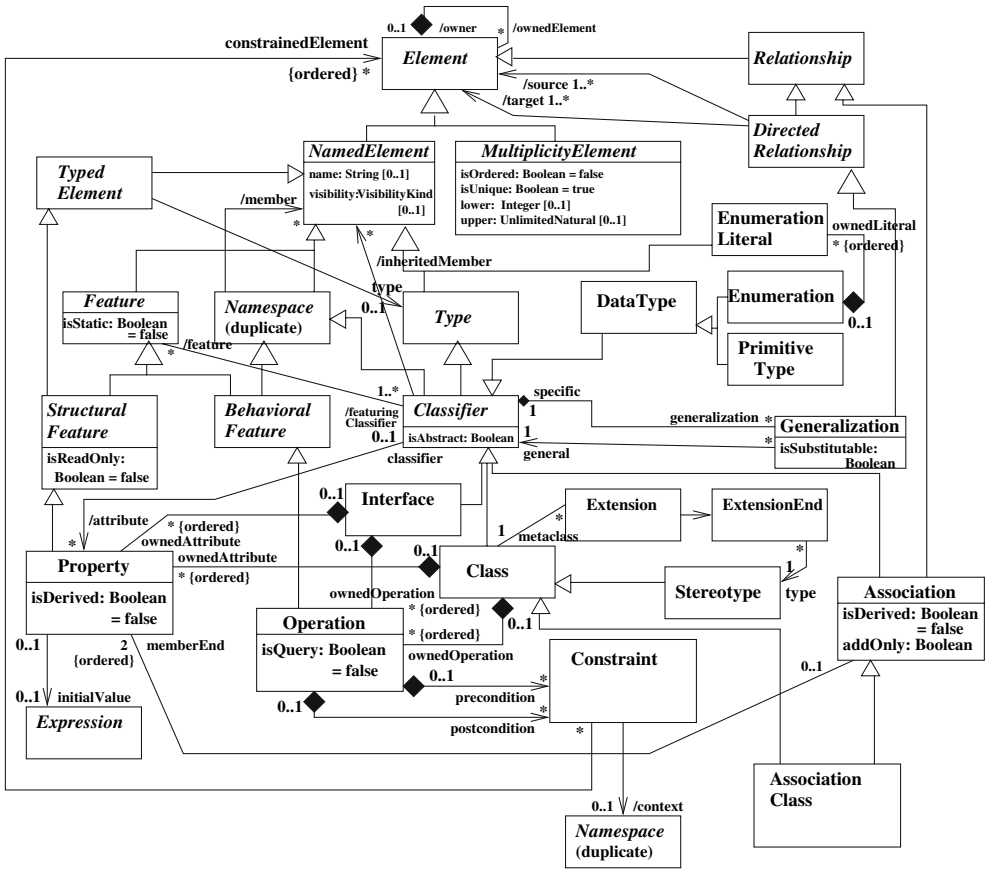


Fig. 11 UML-RSDS class diagram metamodel

- Association ends are either sets ($isUnique = true$ and $isOrdered = false$) or sequences ($isUnique = false$ and $isOrdered = true$).
- Attributes always have multiplicities 1..1:

$$attribute/ = \{\} \Longrightarrow attribute.lower = 1 \;\&\; attribute.upper = 1$$

- Navigability and visibility of elements are not represented.
- Behavioural features are assumed to have *in* parameters only, except for query operations, which may also have a single *return* parameter. Exceptions are not considered. A $bodyCondition$ is expressed instead by a postcondition.

A.3 UML-RSDS state machines

Figure 12 shows the metamodel of the state machine notation.

$Parameter$ also inherits from $TypedElement$ and $MultiplicityElement$. State machines can have entry actions to states, and state invariants. Behaviour state machine transitions are written with the syntax

$$s \longrightarrow_{ev[G]/acts} t$$

where s is the source state, t the target state, ev a trigger event (an operation call), G a guard condition, and $acts$ a list of actions $objs.op(p)$ to be performed on supplier objects or on the $self$ object. Protocol transitions have a postcondition in place of the actions. The trigger, guard and actions/postcondition can all be omitted. The default guard is $true$.

The following restrictions apply compared to UML 2.0 state machines:

- Only state machines that consist only of basic (non-composite) states are used. Concurrent composite states are not permitted except at the top level of the system specification:

$$region.size = 1$$

is an invariant of $StateMachine$ in Fig. 12.

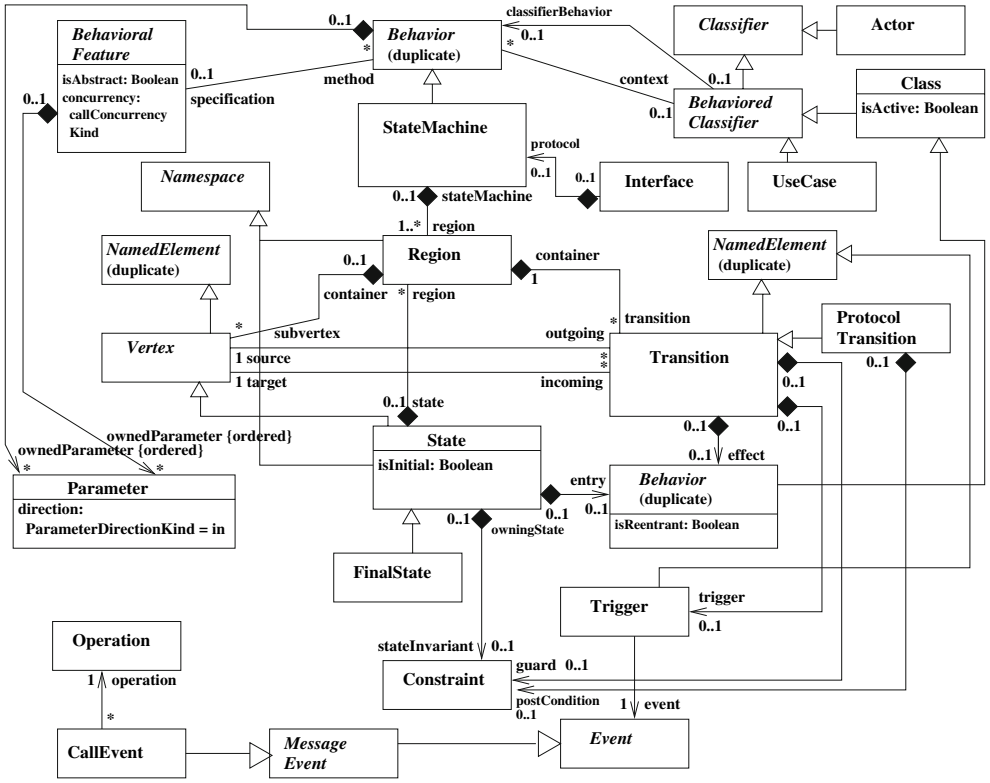


Fig. 12 UML-RSDS state machine metamodel

- A transition cannot have multiple triggers.
- There are no pseudostates such as history states. Initial states are represented by the $isInitial$ attribute of $State$.
- If a state machine describes the behaviour of objects of a class, then all the triggers of its transitions are call events of operations of this class:

$$
\begin{aligned}
&specification = \{\} \implies \\
&\quad (t : region.transition \implies t.trigger.size = 1) \ \& \\
&\quad region.transition.trigger.event <: CallEvent \ \& \\
&\quad region.transition.trigger.event.operation <: \\
&\quad context.feature
\end{aligned}
$$

- If a state machine describes the behaviour of an operation, then its transitions have no triggers (they are triggered by completion events of their source states [42, p. 555]):

$$
\begin{aligned}
&specification \neq \{\} \implies \\
&\quad region.transition.trigger = \{\}
\end{aligned}
$$

Behavioural state machines can have state invariants, in UML-RSDS: we consider this is useful to support verification of the algorithms described by these machines, e.g., by the usual weakest-precondition analysis as supported by B.

B Real time action logic

This appendix presents the underlying RAL formalism used for UML-RSDS semantics.

B.1 Core formalism

The core logic of RAL is an extension of the Object Calculus of Fiadeiro and Maibaum [16,17] to cover durative actions and real-time constraints, based on RTL. The syntactic elements of an RAL theory are *type* symbols, *function* symbols, *attribute* symbols denoting time-varying data items, and *action* symbols denoting actions which may change the value of these attributes. Each theory has a collection of axioms relating these symbols.[2]

Formally, a signature Σ of an RAL theory is a finite set of symbols, with $\mathrm{Att}(\Sigma)$ and $\mathrm{Ac}(\Sigma)$ the sets of attribute and action symbols in Σ. The sets of type, function and predicate symbols are, respectively, $\mathrm{T}(\Sigma)$, $\mathrm{F}(\Sigma)$ and $\mathrm{P}(\Sigma)$. $\mathrm{Att}(\Sigma) \cap \mathrm{Ac}(\Sigma) = \{\}$, $\mathrm{Att}(\Sigma) \cap \mathrm{T}(\Sigma) = \{\}$, and similarly for the other subsets of Σ.

$$\Sigma = \mathrm{Att}(\Sigma) \cup \mathrm{Ac}(\Sigma) \cup \mathrm{T}(\Sigma) \cup \mathrm{F}(\Sigma) \cup \mathrm{P}(\Sigma)$$

[2] Theories are also termed 'modules' in the following.

Each action symbol $\alpha \in \mathrm{Ac}(\Sigma)$ has a *(write) frame* $\mathcal{F}(\alpha) \subseteq \mathrm{Att}(\Sigma)$, which is the set of attributes whose value it may change. Each action, function, predicate and attribute symbol p has an *arity* $arity(p) \in \mathbb{N}$, and a sequence *parameters* $(p) \in seq(\mathrm{T}(\Sigma))$ of parameter types. $arity(p)$ is the length of $parameters(p)$.

We include the usual type, function and predicate symbols of predicate calculus and ZF set theory in each RAL theory [39]. The function $card$ gives the cardinality of a set (the finite or infinite cardinal isomorphic to the set). Functions are defined as particular sets of ordered pairs, as usual. The type of functions from D to R is denoted $D \to R$. The range of a function f is denoted $ran(f)$, the domain is $dom(f)$. The types $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{R}$ and $\mathbb{S}$ (of strings) will usually be assumed to exist in $\mathrm{T}(\Sigma)$ with the usual axioms. A 'universal type' corresponding to $OclAny$ [43] could also be added.

We also assume there is a type $TIME$ of times, with $\mathbb{N} \subseteq TIME$, $TIME$ is totally ordered by a relation $<$, with least element 0, and satisfying the axioms of a totally ordered ring with addition $+$ and unit 0, and multiplication operation $*$ with unit 1. We will usually assume there is an attribute $now : TIME$.

For each action α there are function symbols $\leftarrow(\alpha, i)$, $\rightarrow(\alpha, i)$, $\uparrow(\alpha, i)$ and $\downarrow(\alpha, i)$, where the parameter i ranges over $\mathbb{N}_1$. These correspond to the RTL event occurrence operators for operation events, and have the following meanings:

1. $\rightarrow(\alpha, i)$ is the time that the i-th request for execution of α is received (by the specific target object). Equivalently, it is the request time of the i-th invocation instance of α, since we enumerate these instances in the order of their requests.
2. $\uparrow(\alpha, i)$ is the activation time of the i-th invocation instance of α.
3. $\downarrow(\alpha, i)$ is the termination time of the i-th invocation instance of α.
4. $\leftarrow(\alpha, i)$ is the time of the invocation which created the i-th request for execution of α.

The parameters of these functions are those of α plus $i : \mathbb{N}_1$.

In UML terms, (α, i) can be considered as an instance of the Behaviour denoted by α, considered as a class [42, Sect. 13]. The times $\leftarrow(\alpha, i)$, $\rightarrow(\alpha, i)$, $\uparrow(\alpha, i)$, $\downarrow(\alpha, i)$ are the times of events associated with this instance ($Message\ Event$, $CallEvent$ and $ExecutionEvent$s, respectively). The semantics also relates directly to the concept of a *stimulus* in the UML profile for performance and time [32,41].

Local attributes of (α, i) are written as $(\alpha, i).att$ and are represented as attributes of the module, with parameters those of α, plus i, plus any defined for att itself. These attributes can represent local variables of α or denote the identity of the sender of the request.

The only other elements of the core language are predicates of the form $\varphi \circledcirc t$ "φ holds at time $t : TIME$", where φ is a predicate; and terms of the form $e \circledast t$ "the value of term e at time $t : TIME$". Otherwise, terms and formulae are constructed as for classical predicate calculus with equality and with connectives $\wedge$, $\vee$, $\Longrightarrow$, $\neg$, $\forall$ and $\exists$. $\forall x : T \cdot \varphi$ abbreviates $\forall x \cdot x \in T \Longrightarrow \varphi$ as usual. The connectives $\circledcirc$ and $\circledast$ bind more closely than any other binary operators. Thus $x = y \circledast t$ means $x = (y \circledast t)$.

now has the characteristic property that

$$\forall t : TIME \cdot now \circledast t = t$$

B.2 Derived constructs

For each action instance we can express the $delay$ in its activation and $duration$ of its execution:

1. $delay(\alpha, i) = \uparrow(\alpha, i) - \rightarrow(\alpha, i)$
2. $duration(\alpha, i) = \downarrow(\alpha, i) - \uparrow(\alpha, i)$

We can express that one action always calls another when it executes:

$$\alpha \supset \beta \equiv \forall i : \mathbb{N}_1 \cdot \exists j : \mathbb{N}_1 \cdot \uparrow(\alpha, i) = \uparrow(\beta, j) \wedge \downarrow(\alpha, i) = \downarrow(\beta, j)$$

"α calls β". This is also used to express that α is defined by a (composite) action β.

Some important properties of $\supset$ are that it is transitive:

$$(\alpha \supset \beta) \wedge (\beta \supset \gamma) \Longrightarrow (\alpha \supset \gamma)$$

and that statement constructs such as ; and $if\ then\ else$ (Appendix C.4) are monotonic with respect to it:

$$(\alpha_1 \supset \alpha_2) \wedge (\beta_1 \supset \beta_2) \Longrightarrow (\alpha_1 ; \beta_1 \supset \alpha_2 ; \beta_2)$$

and

$$(\alpha_1 \supset \alpha_2) \wedge (\beta_1 \supset \beta_2) \Longrightarrow if\ E\ then\ \alpha_1\ else\ \beta_1 \supset if\ E\ then\ \alpha_2\ else\ \beta_2$$

In UML terms the input pool of received and waiting to be processed messages of an object are all those $m(x), i$ instances for which $\rightarrow(m(x), i) \leq now$ and $\uparrow(m(x), i) > now$. x are the input parameter values of the invocation of m.

We can define counters $\#req(\alpha)$, $\#act(\alpha)$, $\#fin(\alpha)$ and $\#snd(\alpha)$ for requests, activations, terminations and invocations of action α:

1. $\#req(\alpha) \circledast t = card(\{j : \mathbb{N}_1 | \rightarrow(\alpha, j) \leq t\})$
   This is the number of distinct request events for α which have occurred so far.
2. $\#act(\alpha) \circledast t = card(\{j : \mathbb{N}_1 | \uparrow(\alpha, j) \leq t\})$
3. $\#fin(\alpha) \circledast t = card(\{j : \mathbb{N}_1 | \downarrow(\alpha, j) \leq t\})$
4. $\#snd(\alpha) \circledast t = card(\{j : \mathbb{N}_1 | \leftarrow(\alpha, j) \leq t\})$

The number of currently executing instances of α (at a time t) is therefore

$$\#active(\alpha)\circledast t \;=\; \#act(\alpha)\circledast t - \#fin(\alpha)\circledast t$$

whilst the number waiting to be activated is

$$\#waiting(\alpha)\circledast t \;=\; \#req(\alpha)\circledast t - \#act(\alpha)\circledast t$$

Using these counters we can express a wide range of mutual exclusion, synchronisation and prioritisation properties. For example, a set S of actions are *fully mutually exclusive*, $fmutex(S)$ if at most one instance of these actions can be executing at any time:

$$\forall t : TIME \cdot (\#active(\alpha_1) + \ldots + \#active(\alpha_n) \;\leq\; 1)\circledcirc t$$

where $S = \{\alpha_1, \ldots, \alpha_n\}$. In particular if (the behaviour of) an operation m is not reentrant (in UML terms), then $fmutex(\{m\})$ holds.

The operators $\bigcirc$ "next", $\Box$ "always in the future" and $\diamond$ "eventually" of linear temporal logic can be defined in terms of the activation times of execution instances. For example, $\Box\phi$ "ϕ holds at all future instants" is interpreted as meaning "ϕ holds at all future activation times of an action of the system":

$$\begin{array}{l}(\Box_S\phi)\circledcirc t \;\equiv \\ \quad \forall i : \mathbb{N}_1 \cdot \uparrow(\alpha_1, i) \geq t \implies \phi\circledcirc\uparrow(\alpha_1, i) \;\wedge \ldots \wedge \\ \quad \forall i : \mathbb{N}_1 \cdot \uparrow(\alpha_n, i) \geq t \implies \phi\circledcirc\uparrow(\alpha_n, i)\end{array}$$

where the set of actions is $S = \{\alpha_1, \ldots, \alpha_n\}$.

The motivation for this definition is that in a concurrent environment, invariant properties of a module must be true at all time points where the state of a system can be *observed*. At the specification level the effects of operations are defined by comparing the state at initiation of the operation to the state at termination. So states at the initiation and termination of operations are the critical 'observable' points.

Similarly we define $\bigcirc_S\phi$ and $\diamond_S\phi$. We usually drop the subscript S where it is clear from context.

There are corresponding temporal operators which refer to *all* times:

$$\begin{array}{l}(\Box^\tau\varphi)\circledcirc t \equiv \\ \qquad \forall s : TIME \cdot s \geq t \implies \varphi\circledcirc s\end{array}$$

Finally, the weakest precondition operator $[\alpha]P$ "every execution of α establishes P" of B [1] and modal action logic [47] can be defined, where P may contain terms of the form $e@pre$ denoting the value of expression e at initiation of α:

$$\begin{array}{l}([\alpha]P)\circledcirc t \equiv \\ \quad \forall i : \mathbb{N}_1 \cdot \uparrow(\alpha, i) = t \implies P[e\circledast\uparrow(\alpha, i)/e@pre]\circledcirc\downarrow(\alpha, i)\end{array}$$

$E[ex/v]$ denotes the substitution of expression(s) ex for identifier(s) v in E. In this substitution each pre-state expression $e@pre$ in P is replaced by the value $e\circledast\uparrow(\alpha, i)$ of e at initiation of α.

The [ ] operator can be used to concisely express properties of action invocations without requiring reference to the index of these invocations. It also provides a general way of expressing the effect of actions.

B.3 Axioms of RAL

We take the axioms of classical predicate logic with equality in this language, with the following modifications.

The predicate logic axiom $\forall$-elimination:

$$(\forall v : T \cdot \varphi) \implies \varphi[e/v]$$

is only valid if e is free for the variable v in φ, and the substitution does not introduce new occurrences of attributes within modal operators ($\circledast$ and $\circledcirc$ in the core language) in φ. Similarly the equality axiom

$$e_1 = e_2 \implies (\varphi[e_1/v] \equiv \varphi[e_2/v])$$

is only asserted when e_1 and e_2 are free for the variable v in φ, and all free occurrences of v in φ are outside the scope of a modal operator.

$$(\Box^\tau(e_1 = e_2))\circledcirc 0 \implies (\varphi[e_1/v] \equiv \varphi[e_2/v])$$

for any formula φ, where e_1 and e_2 are terms free for the variable v in φ.

If v_i is a variable not free in the terms e or t, then:

$$\exists v_i \cdot (v_i = e)\circledcirc t$$

"Expressions always evaluate to a value".

The equality axiom

$$e = e$$

is valid for all terms e.

Variables act as logical constants over time:

$$\forall v_i : X \cdot \forall t : TIME \cdot v_i = v_i\circledast t$$

The core logical axioms assumed are:

$$(C1) : \forall i : \mathbb{N}_1 \cdot \rightarrow(\alpha, i) \;\leq\; \rightarrow(\alpha, i+1)$$

for each action α. This expresses that the index i identifies an execution instance of α by the order in which the request for the execution arrives at the target object.

$$(C2) : \forall i : \mathbb{N}_1 \cdot \leftarrow(\alpha, i) \;\leq\; \rightarrow(\alpha, i) \;\leq\; \uparrow(\alpha, i) \;\leq\; \downarrow(\alpha, i)$$

for each action α. "Each invocation instance must be sent before it is requested, requested before it can activate, and must activate before it can terminate".

This axiom does not require that executions initiate in the order of their requests, this additional property can be asserted by a constraint if required.

The *compactness* condition is that for all $p \in \mathbb{N}$ there are only finitely many $i : \mathbb{N}_1$ and $x : X$ such that $\uparrow(\alpha(x), i) < p$, for each action α. Similar conditions are required for the $\rightarrow$, $\leftarrow$ and $\downarrow$ times.

The *frame* axioms express that attributes of a module M can only change in value over intervals in which an action of M executes – these axioms are a form of *locality* property in the sense of [17]:

For each attribute $att \in \text{Att}(\Sigma)$, where Σ is the signature of M, let $\alpha_1, \ldots, \alpha_n$ be all the actions $\alpha \in \text{Ac}(\Sigma)$ which have $att \in \mathcal{F}(\alpha)$. Then $Frame_{att}$ is the axiom

$$\begin{aligned}\forall t_1, t_2 : TIME \cdot \\ t_1 < t_2 \wedge att@t_1 \neq att@t_2 \implies \\ \exists t : TIME \cdot t_1 \leq t < t_2 \wedge \\ ((\#active(\alpha_1) > 0)@t \vee \ldots \vee \\ (\#active(\alpha_n) > 0)@t)\end{aligned}$$

In words: "If the value of att changes from t_1 to t_2, there must be an action with att in its write frame which executes in that interval".[3]

These axioms are particularly relevant when defining the meaning of subclassing. They are used to define a class as being an 'open' or 'extendible' type in the sense of [48]: new behaviour and data can be added to a class but must preserve the behaviour of the superclass.

($C3$): Axioms for $@$:

$$\begin{aligned}(\varphi@s)@t &\equiv \varphi@(s@t) \\ (\varphi \wedge \phi)@t &\equiv \varphi@t \wedge \phi@t \\ (\varphi \vee \phi)@t &\equiv \varphi@t \vee \phi@t \\ (\varphi \implies \phi)@t &\equiv (\varphi@t \implies \phi@t) \\ (\neg\varphi)@t &\equiv \neg(\varphi@t) \\ (\forall v : T \cdot \varphi)@t &\equiv \forall v : T \cdot (\varphi@t) \\ (\exists v : T \cdot \varphi)@t &\equiv \exists v : T \cdot (\varphi@t)\end{aligned}$$

In the last two cases, v must not be free in t.

($C4$): Axioms for $@$:

$$\varphi@t \equiv \varphi^{*t}$$

where φ contains no modal operators, and φ^{*t} is φ with each outermost term e occurring in a subformula replaced by $e@t$, where t has no free variables.

Also ($C5$):

$$g(e_1, \ldots, e_n)@t = g(e_1@t, \ldots, e_n@t)$$

for each $g \in \text{F}(\Sigma)$ of arity n, $t : TIME$.

[3] When α_i has parameters $x_i : X_i$ we use $(\exists x_i : X_i \cdot \#active(\alpha_i(x_i)) > 0)@t$ in the conclusion.

$C3$, $C4$ and $C5$ are essential to prove the completeness of the RAL formalism with respect to its denotational semantics [29].

The usual concept of inference, denoted by $\vdash$, is taken. The inference rules are those of classical predicate calculus: modus ponens and $\forall$-introduction. In addition there is the rule of $\Box^\tau$-introduction:

$$\frac{\Gamma \vdash \varphi}{\Gamma \vdash \forall t : TIME \cdot \varphi@t}.$$

C Theory refinement and composition

UML-RSDS supports modular specification to decompose a system into analysable parts. Specifications are structured at the three levels of objects, classes and subsystems (sub-models). The semantics for UML-RSDS also follows this structural decomposition by defining a corresponding structure of theories for objects, classes and subsystems, and combining these theories to form the semantics of complete models. The main way in which theories can be combined is via morphisms between the theories.

C.1 Theory morphisms

Let M and M' be two theories with signatures Σ and Σ', respectively.

A *signature morphism* $\sigma : \Sigma \rightarrow \Sigma'$ must map attribute symbols to attribute symbols, action symbols to action symbols, etc., and preserve the arities of these symbols:

$$\begin{aligned}\sigma(|\text{Att}(\Sigma)|) &\subseteq \text{Att}(\Sigma') \\ \sigma(|\text{Ac}(\Sigma)|) &\subseteq \text{Ac}(\Sigma') \\ \sigma(|\text{T}(\Sigma)|) &\subseteq \text{T}(\Sigma') \\ \sigma(|\text{P}(\Sigma)|) &\subseteq \text{P}(\Sigma') \\ \sigma(|\text{F}(\Sigma)|) &\subseteq \text{F}(\Sigma')\end{aligned}$$

with the arity in Σ' of $\sigma(f)$ being the same as $arity(f)$ in Σ for each $f \in \text{F}(\Sigma) \cup \text{P}(\Sigma) \cup \text{Ac}(\Sigma) \cup \text{Att}(\Sigma)$, and with parameter types also translated via σ for corresponding function, predicate, attribute and action symbols in the two theories.

Normally σ maps the standard types $TIME$, $\mathbb{N}$, etc., in M to the corresponding types in M'.

For each action symbol $\alpha \in \text{Ac}(\Sigma)$,

$$\mathcal{F}(\sigma(\alpha)) \subseteq \sigma(|\mathcal{F}(\alpha)|)$$

In other words, the frame of an action may become more restrictive in M'.

σ can be extended to general terms and formulae of M in the usual way, so that $\sigma(t)$ is a term of M' if t is a term of M, etc.

σ is a *theory morphism* if

$$M \vdash \varphi \implies M' \vdash \sigma(\varphi)$$

for each formula φ of M.

In particular, this means that the frame axiom for each attribute att of M must be true in interpreted form in M':

$$\begin{array}{l}\forall t_1, t_2 : TIME \cdot \\ \quad t_1 < t_2 \wedge \sigma(att) \circledast t_1 \neq \sigma(att) \circledast t_2 \implies \\ \quad \exists t : TIME \cdot t_1 \leq t < t_2 \wedge \\ \quad ((\#active(\sigma(\alpha_1)) > 0) \circledcirc t \vee \ldots \vee \\ \quad (\#active(\sigma(\alpha_n)) > 0) \circledcirc t)\end{array}$$

where the α_i are all the actions of M with att in their write frame. In other words, (the interpretation of) att can only change value over intervals where (the interpretation of) one of its updating actions of M is executing. But this means that every new action

$$\beta \in \mathrm{Ac}(\Sigma') \setminus \sigma(|\mathrm{Ac}(\Sigma)|)$$

which has $\sigma(att)$ in $\mathcal{F}(\beta)$ co-executes with (or calls) one of the $\sigma(\alpha_i)$.

This form of encapsulation of data is similar to that found in languages such as B [1], or in the subtyping definition of [37]: only the actions declared in the same module as a particular data item can directly write that data. Actions of other modules must invoke these actions in order to change the data.

C.2 Class and instance theories

In an object-oriented system, we may have theories $\mathcal{I}_C$ representing a *typical instance* (or *object*) of a class C, and a theory Γ_C representing the class itself (including all its current instances) [6].

RAL attributes will represent UML instance scope and class scope attributes, roles (association ends) and query operations (collectively referred to as *data features*), and RAL actions will represent instance and class scope update operations. An instance theory $\mathcal{I}_C$ will have an attribute $att : X'$ for each declared attribute $att : X$ in the text of a UML class C, for each query operation of C and for each opposite association end of an association attached to C. X' is the semantic type corresponding to X. There will be an action $\alpha(X')$ for each update operation with input parameter type X.

In instance theories instance-level properties can be proved, independent of object identity. In the class theory these properties then become available as theorems about all objects of the class.

We represent class scope (static) features in the instance theories, since these features are available at the instance level. Their special property is that their values are always identical in every instance, this follows since there is a single semantic representation of the static feature.

In the class theory Γ_C, there will be a type $@C$ of possible instances of C, and an attribute

$$\overline{C} : \mathbb{F}(@C)$$

representing the set of currently existing instances, together with actions $kill_C(@C)$ and $create_C(@C)$ to delete and add elements to this set. $\overline{C}$ corresponds to $C.allInstances()$ in OCL [43].

Every element of $\overline{C}$ will have an associated value for each data feature $f : X$ declared in the class. An additional parameter of type $@C$ representing the object is added to each (instance scope) attribute $att : X'$ and action $\alpha(X')$ of $\mathcal{I}_C$ to produce a parameterised attribute or action of Γ_C:

$$\begin{array}{l} att(@C) : X' \\ \alpha(@C, X') \end{array}$$

For a in $@C$ we usually write $att(a)$ as $a.att$ and $\alpha(a, x)$ as $a.\alpha(x)$ for consistency with standard OO notation.

Attributes or actions which represent class scope (static) features do not gain the additional parameter.

This general construction is termed an A-morphism [16], where A is the set of object identifiers/references, and this involves a modified form of signature morphism $\sigma : \Sigma \to \Sigma'$ in which

$$\begin{array}{l} arity(\sigma(att)) = arity(att) + 1 \\ arity(\sigma(\alpha)) = arity(\alpha) + 1 \end{array}$$

for instance scope $att \in \mathrm{Att}(\Sigma)$, $\alpha \in \mathrm{Ac}(\Sigma)$, and the new parameter has type $A \in \mathrm{T}(\Sigma')$ and is the first parameter of $\sigma(att)$ or $\sigma(\alpha)$ in the second theory. Otherwise σ is as previously defined.

The analogy of a theory morphism in this case is that

$$M \vdash \varphi \implies M' \vdash \forall a : A \cdot a.\sigma(\varphi)$$

where $a.\psi$ is ψ with a substituted into each new parameter slot created by the morphism.

We construct the class theory Γ_C as a combination of $\mathcal{I}_C$ via a $@C$-morphism, and a generic *class manager* theory M via a theory morphism μ:

$$\begin{array}{rcl} @X & \longmapsto & @C \\ \overline{X} & \longmapsto & \overline{C} \\ create_X & \longmapsto & create_C \\ kill_X & \longmapsto & kill_C \end{array}$$

Figure 13 shows this structure.

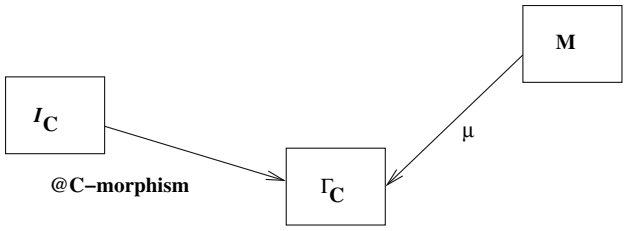


Fig. 13 Class theory construction

M has type symbol $@X$, attribute $\overline{X} : \mathbb{F}(@X)$, actions $create_X(@X)$ and $kill_X(@X)$ and axioms

$(\overline{X} = \{\}) \circledcirc 0$

$\forall a : @X \cdot [create_X(a)](\overline{X} = \overline{X}@pre \cup \{a\})$

$\forall a : @X \cdot [kill_X(a)](\overline{X} = \overline{X}@pre - \{a\})$

The frames of $kill_X$ and $create_X$ are both $\{\overline{X}\}$.

C.3 Time variables

In the specification of real-time or hybrid systems, two kinds of attributes can be identified:

1. Discrete data, corresponding to discrete data in the real world, or discretised approximations of continuous data.
2. Continuous data, or 'time variables'.

Both can be represented by RAL attributes. However, whilst discrete variables are conventional variables of a computational system, time variables represent physical quantities and may vary as arbitrary functions of time.

Thus these two separate forms of attribute need to be distinguished. Discrete instance attributes $att : X$ of a class can be modelled by RAL attributes which have the following properties:

1. att only takes on finitely many values over the lifetime of the system:

$$\{att \circledast t | t \in TIME\} \in \mathbb{F}(X')$$

2. for each value $val : X'$, $att = val$ is true for a finite collection I_{val} of intervals of non-zero length:

$$\forall val : X' \cdot \exists I_{val} : \mathbb{F}(INTERVAL(TIME)) \cdot \{t | t \in TIME \wedge att \circledast t = val\} = \bigcup_{J_i \in I_{val}} J_i$$

$INTERVAL(TIME)$ is the set of convex sets $J \subseteq TIME$ such that

$$\forall t, t' : J \cdot t < t' \Longrightarrow \forall t'' : TIME \cdot t \leq t'' \leq t' \Longrightarrow t'' \in J$$

In contrast, time variables need not have any constraints on the variation of their values over time. However, it is often assumed that they are at least piecewise continuous functions of time when their values range over a subset of $\mathbb{R}$. They are usually modified only by special actions which model the behaviour of physical systems [22].

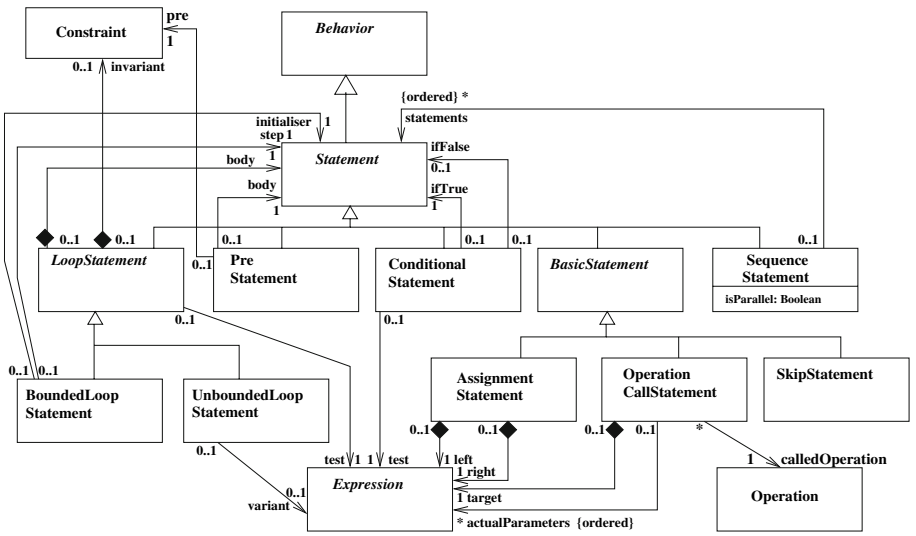
The prime example of a time variable, included in every UML-RSDS instance theory, is the attribute $now : TIME$, which satisfies the axiom

$$\forall t : TIME \cdot now \circledast t = t.$$

C.4 Composite and procedural actions

We will introduce a small procedural language (Fig. 14) to allow procedural-style definitions of behaviour for UML operations. It is also used as the target for translations to B and Java in the UML-RSDS tools, so assigning a semantics

Fig. 14 Statement metamodel



to such constructs allows us to verify the correctness of these translations.

Normally $\leftarrow(S,i) = \rightarrow(S,i) = \uparrow(S,i)$ is assumed for such composed actions S, since they are normally invoked by the same object on which they execute.

Assignment $t_1 := t_2$ can be defined as the action $\alpha_{t_1:=t_2}$ where t_1 is an attribute symbol, the write frame of this action is $\{t_1\}$, and

$$\forall i : \mathbb{N}_1 \cdot t_1 \circledast \downarrow(\alpha_{t_1:=t_2}, i) = t_2 \circledast \uparrow(\alpha_{t_1:=t_2}, i)$$

For formulae P without time variables, occurrences of modal operators or $@pre$, this means

$$([\alpha_{t_1:=t_2}]P) \circledcirc t \;\equiv\; P[t_2/t_1] \circledcirc t$$

as usual for assignment, if no other action co-executes with this action.

Similarly sequential composition ; and parallel composition || of actions can be expressed as derived combinators:

$$\forall i : \mathbb{N}_1 \cdot \exists j, k : \mathbb{N}_1 \cdot$$
$$\uparrow(\alpha;\beta,i) = \uparrow(\alpha,j) \wedge \downarrow(\alpha;\beta,i) = \downarrow(\beta,k) \wedge \uparrow(\beta,k)$$
$$= \downarrow(\alpha,j)$$

and

$$\forall j, k : \mathbb{N}_1 \cdot$$
$$\uparrow(\beta,k) = \downarrow(\alpha,j) \implies$$
$$\exists i : \mathbb{N}_1 \cdot \uparrow(\alpha;\beta,i) = \uparrow(\alpha,j) \;\wedge\; \downarrow(\alpha;\beta,i) = \downarrow(\beta,k)$$

These two conditions yield the usual axiom that $[\alpha;\beta]\varphi \equiv [\alpha][\beta]\varphi$ for φ without occurrences of $@pre$.

For parallel $\gamma = \alpha||\beta$:

$$\forall i : \mathbb{N}_1 \cdot \exists\, j, k : \mathbb{N}_1 \cdot$$
$$\uparrow(\gamma,i) = \uparrow(\alpha,j) \;\wedge\; \uparrow(\gamma,i) = \uparrow(\beta,k) \;\wedge$$
$$\downarrow(\gamma,i) = \downarrow(\beta,k) \;\wedge\; \downarrow(\gamma,i) = \downarrow(\alpha,j)$$

and

$$\forall j, k : \mathbb{N}_1 \cdot \uparrow(\beta,k) = \uparrow(\alpha,j) \;\wedge\; \downarrow(\beta,k) = \downarrow(\alpha,j) \implies$$
$$\exists i : \mathbb{N}_1 \cdot \uparrow(\gamma,i) = \uparrow(\alpha,j) \;\wedge\; \downarrow(\gamma,i) = \downarrow(\alpha,j)$$

The usual property

$$(P_1 \implies [\alpha]Q_1) \wedge (P_2 \implies [\beta]Q_2)$$
$$\implies (P_1 \wedge P_2 \implies [\gamma](Q_1 \wedge Q_2))$$

can be derived. The ; and || composite actions have write frames the union of the write frames of their component actions.

Conditional actions α representing $if\ E\ then\ S_1\ else\ S_2$ are defined to have the properties:

$$\forall i : \mathbb{N}_1 \cdot E \circledcirc \uparrow(\alpha,i) \implies$$
$$\exists j : \mathbb{N}_1 \cdot \uparrow(\alpha,i) = \uparrow(S_1,j) \;\wedge\; \downarrow(\alpha,i) = \downarrow(S_1,j)$$

and:

$$\forall i : \mathbb{N}_1 \cdot \neg E \circledcirc \uparrow(\alpha,i) \implies$$
$$\exists j : \mathbb{N}_1 \cdot \uparrow(\alpha,i) = \uparrow(S_2,j) \;\wedge\; \downarrow(\alpha,i) = \downarrow(S_2,j)$$

Occurrences of $if\ E\ then\ S_1\ else\ S_2$ are either occurrences of S_1 if E holds at commencement of this action, or occurrences of S_2, if $\neg E$ holds. This action has write frame the union of those of S_1 and S_2.

Occurrences of $while\ E\ do\ S$ are a sequence of occurrences $(S,i_1),\ldots,(S,i_n)$ of S, where E holds at the commencement of each of these actions, and where E fails to hold at termination of (S,i_n). The $while$ action has the same write frame as S. Bounded loops can be defined in terms of unbounded loops.

Preconditioned actions β: $pre\ Pre\ then\ S$ are defined to have

$$\forall i : \mathbb{N}_1 \cdot Pre \circledcirc \uparrow(\beta,i) \implies$$
$$\exists j : \mathbb{N}_1 \cdot \uparrow(\beta,i) = \uparrow(S,j) \;\wedge\; \downarrow(\beta,i) = \downarrow(S,j)$$

and

$$\forall i : \mathbb{N}_1 \cdot Pre \circledcirc \uparrow(S,i) \implies$$
$$\exists j : \mathbb{N}_1 \cdot \uparrow(\beta,j) = \uparrow(S,i) \;\wedge\; \downarrow(\beta,j) = \downarrow(S,i)$$

This means that

$$([pre\ P\ then\ S]Post) \circledcirc t \implies$$
$$(\forall i : \mathbb{N}_1 \cdot P \circledcirc \uparrow(S,i) \wedge t = \uparrow(S,i)$$
$$\implies Post[e \circledast \uparrow(S,i)/e@pre] \circledcirc \downarrow(S,i))$$

D The UML-RSDS tools

A large integrated toolset has been developed to support software development with UML-RSDS, including code generation from constraints. Figure 15 shows a screenshot of the UML-RSDS tools.

The tool facilities include:

1. Diagram creation and editing for class diagrams, use cases, interactions and state machines.
2. Syntactic and semantic checks on diagram correctness, including consistency and completeness of constraints.
3. Semantics-preserving transformations on UML models, as described in Sect. 4.3.
4. Translation from UML-RSDS specifications into SMV [2], the B notation (Sect. 4.2), and Java.

The translation and diagram checking operations are fully automated. Transformations are also automatically applied, but must be selected manually by the tool user. The tool is available at: http://www.dcs.kcl.ac.uk/staff/kcl/umlrsds.

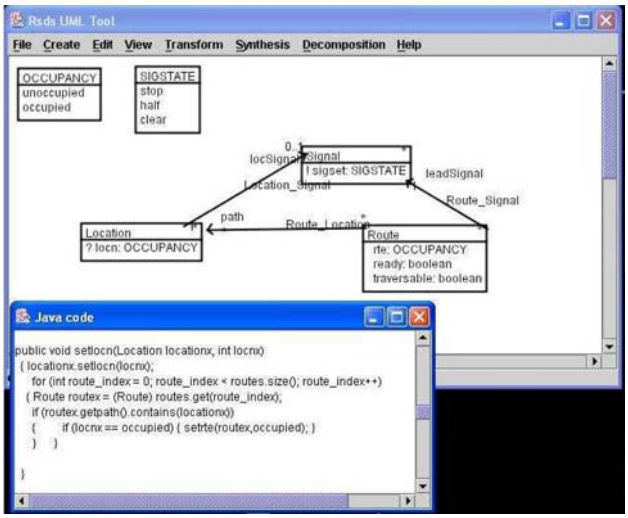


Fig. 15 Interface of the UML-RSDS tool

References

1. Abrial, J.-R.: The B Book: Assigning Programs to Meanings. Cambridge University Press, Cambridge (1996)
2. Androutsopoulos, K.: Verification of reactive system specifications using model checking. Ph.D. thesis, King's College, London (2004)
3. Aruchamy, G., Kim Cheng, A.M.: Translating real-time UML timing constraints into real-time logic formulas. Technical Report UH-CS-06-07, University of Houston, Houston (2006)
4. B4Free (2006) B4Free, http://www.b4free.com
5. Bicarregui, J.C., Lano, K.C., Maibaum, T.S.E.: Objects, associations and subsystems: a hierarchical approach to encapsulation. In: ECOOP 97, LNCS. Springer, Heidelberg (1997)
6. Bicarregui, J.C., Lano, K.C., Maibaum, T.S.E.: Towards a compositional interpretation of object diagrams. In: Proceedings of IFIP TC2 Working Conference on Algorithmic Languages and Calculi, February, 1997
7. Bowen, J., Hinchey, M.: Ten commandments ten years on: An assessment of formal methods usage. In: Eleftherakis, M. (ed.) SEEFM05: 2nd South-East European Workshop on Formal Methods, pp. 1–16 (2006)
8. Broy, M., Cengarle, M., Rumpe, B.: Towards a system model for UML. UML Semantics Project document 06.06.04 System Model Part 1 (2006)
9. B-Core UK Ltd. The BToolkit (2005)
10. Chiaradia, J.M., Pons, C.: Improving the OCL semantics definition by applying dynamic meta modelling and design patterns. In: OCL for (Meta-) Models in Multiple Application Domains. TUD-FI06-04 (2006)
11. Clark, T., Evans, A., Kent, S., Sammut, P.: The MMF approach to engineering object-oriented design languages. In: Workshop on Language Descriptions, Tools and Applications, LDTA (2001)
12. Damm, W., Josko, B., Pnueli, A., Votintseva, A.: A discrete-time UML semantics for concurrency and communication in safety-itical applications. Sci. Comput. Program. **55**, 81–115 (2005)
13. Dierks, H.: Comparing model-checking and logical reasoning for real-time systems. In: ESSLLI '98, Workshop proceedings, pp. 13–22 (1998)
14. Evans, A., Kent, S.: Core meta-modelling semantics of UML: The pUML approach. In: UML '99, pp. 140–155 (1999)
15. Fecher, H., Schonborn, J., Kyas, M., de Roever, W.-P.: 29 new unclarities in the semantics of UML 2.0 state machines. In: Formal Methods and Software Engineering ICFEM 2005, vol. 3785, LNCS, pp. 52–65. Springer, Heidelberg (2005)
16. Fiadeiro, J., Maibaum, T.: Describing, structuring and implementing objects. In: Foundations of Object Oriented languages, vol. 489 of LNCS. Springer, Heidelberg (1991)
17. Fiadeiro, J., Maibaum, T.: Sometimes "tomorrow" is "sometime". In: Temporal Logic, vol. 827 of Lecture Notes in Artificial Intelligence, pp. 48–66. Springer, Heidelberg (1994)
18. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Reading (2000)
19. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading (1995)
20. Glinz, M.: Problems and deficiencies of UML as a requirements specification language. In: Proceedings of 10th International Workshop on Software Specification and Design (IWSSD-10), pp. 11–22 (2000)
21. Gogolla, M., Radfelder, O., Richters, M.: A UML Semantics FAQ: The View From Bremen. University of Bremen, Bremen (1999)
22. Goldsack, S., Lano, K., Sanchez, A.: Transforming continuous into discrete specifications with VDM^{++}. In: IEE C8 colloquim digest on hybrid control for real-time systems. IEE (1996)
23. Graf, S., Ober, I., Ober, I.: Timed annotations with UML. In: SVERTS '03 (2003)
24. Grand, M.: (1998) Patterns in Java, vol. 1. Wiley, London
25. Jahanian, F., Mok, A.K.: Safety analysis of timing properties in real-time systems. IEEE Trans. Softw. Eng. **SE-12**, 890–904 (1986)
26. Knapp, A., Wuttke, J.: Model checking of UML 2.0 interactions. In: 5th International Workshop CSDUML (2006)
27. Kyas, M., Fecher, H., de Boer, F., Jacob, J., Hooman, J., van der Kwaag, M., Arons, T., Kugler, H.: Formalizing UML models and OCL constraints in PVS. In: SFEDL '04 (2004)
28. Lamport, L.: The temporal logic of actions. Technical Report 79, Digital Equipment Corporation, Systems Research Center, December, 1991
29. Lano, K.: Logical specification of reactive and real-time systems. J. Logic Comput. **8**(5), 679–711 (1998)
30. Lano, K.: UML to B: Formal verification of object-oriented models. In: IFM '04 (2004)
31. Lano, K., Androutsopolous, K.: Automated synthesis of high-integrity systems using model-driven development. In: 5th International Workshop CSDUML (2006)
32. Lano, K., Androutsopolous, K., Clark, D.: Concurrency specification in UML-RSDS. In: MARTES '06, MODELS Conference (2006)
33. Lano, K., Clark, D., Androutsopolous, K.: From implicit specifications to explicit designs in reactive system development. In: IFM '02 (2002)
34. Lano, K., Clark, D., Androutsopolous, K., Kan, P.: Invariant-based synthesis of fault-tolerant systems. In: FTRTFT. Springer, Heidelberg (2000)
35. Lano, K., Clark, D., Androutsopolous, K.: RSDS: A subset of UML with precise semantics. L'Objet **9**(4), 53–73 (2003)
36. Lano, K., Evans, A.: Rigorous development in UML. In: FASE '99, vol. 1577 of Lecture Notes in Computer Science, pp. 129–144. Springer, Heidelberg (1999)
37. Liskov, B., Wing, J.: Specifications and their use in defining subtypes. In: ZUM '95 Proceedings, vol. 967 of LNCS. Springer, Heidelberg (1995)
38. Meyer, B.: Object-Oriented Software Construction. Prentice Hall, Englewood (1997)
39. Monk, J.D.: Mathematical Logic. Springer, Heidelberg (1976)
40. Naumenko, A., Wegmann, A.: Triune continuum paradigm and problems of UML semantics. icwww.epfl.ch/publications/documents/ IC_TECH_REPORT_200344.pdf (2003)

41. OMG. UML profile for schedulability, performance and time. Version 1.1 (2005)
42. OMG. UML superstructure, version 2.0. OMG document formal/05-07-04 (2005)
43. OMG. UML OCL 2.0 specification, final/06-05-01 (2006)
44. Ostroff, J.S.: Temporal Logic for Real-Time Systems. Wiley, London (1989)
45. Pnueli, A.: Applications of temporal logic to the specification and verification of reactive systems: A survey of current trends. In: Current Trends in Concurrency, vol. 224 of LNCS. Springer, Heidelberg (1986)
46. Richters, M.: OCL semantics. Annex A of [43] (2005)
47. Ryan, M., Fiadeiro, J., Maibaum, T.S.E.: Sharing actions and attributes in modal action logic. In: Proceedings of International Conference on Theoretical Aspects of Computer Science (TACS '91). Springer, Heidelberg (1991)
48. Simons, A.: The theory of classification, part 8: Classification and inheritance. J. Object Technol. **2**(4), 55–64 (2003)
49. Smith, J., DeLoach, S., Kokar, M., Baclawski, K.: Category theoretic approaches of representing precise UML semantics. In: Proceedings ECOOP Workshop on Defining Precise Semantics for UML (2000)
50. Sunyé, G., Le Guennec, A., Jézéquel, J.M.: Design patterns application in UML. In: ECOOP 2000, number 1850 in Lecture Notes in Computer Science, pp. 44–62. Springer, Heidelberg (2000)
51. Verhoef, M., Larsen, P., Hooman, J.: Modelling and Validating Distributed Embedded Systems with VDM^{++}. Engineering College of Aarhus, Denmark (2006)

Author's Biography

Dr. Lano has worked on the integration of formal and object-oriented methods, through involvement in the pUML and EROS groups, and in a number of UK and European projects. He is the author of several papers and books on formal object-oriented methods.