Structuring Mechanisms in RSL

Contents

- Modules:
  - schemes
  - objects
- Class expressions:
  - basic
  - extending
  - hiding
  - renaming
- Parameterization and instantiation
- How to build hierarchical specifications

Specifications

An RSL specification consists of

- module definitions

A module contains definitions of

- types
- values
- variables
- channels
- modules
- axioms

Modularity

Modules are building blocks which can be put together to form large, structured specifications.

Purposes:

- Readability
- Separate development
- Reuse

Schemes and Objects

Two kinds of modules: schemes and objects.

A scheme denotes the class of models denoted by its class_expr

> **scheme** id = class_expr

An object denotes a single model of those denoted by its class_expr

> **object** id : class_expr

Syntactic Forms of Class Expressions

- basic
- extending
- renaming
- hiding
- instantiation

Semantics of Class Expressions

A class expression denotes a class of models.

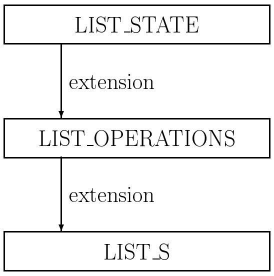
Example:

class
  value i : **Int**
  axiom i > 0
end

denotes

{
  [i $\mapsto$ 1] ,
  [i $\mapsto$ 2] ,
  $\vdots$
}

Extension - Example



scheme LIST_STATE =
  class variable list : $\mathbf{Int}^*$ **end**

scheme LIST_OPERATIONS =
  extend LIST_STATE **with**
  class
    value
      empty : **Unit** $\rightarrow$ **write** list **Unit**,
      is_empty : **Unit** $\rightarrow$ **read** list **Bool**,
      add : **Int** $\rightarrow$ **write** list **Unit**
  end

scheme LIST_S =
  extend LIST_OPERATIONS **with**
  class
    axiom forall i : **Int** •
      empty() $\equiv$ list := $\langle\rangle$,
      is_empty() $\equiv$ list = $\langle\rangle$,
      add(i) $\equiv$ list := $\langle i \rangle$ ^ list,
  end

short for

scheme LIST_S =
  class
    variable
      list : $\mathbf{Int}^*$
    value
      empty : **Unit** $\rightarrow$ **write** list **Unit**,
      is_empty : **Unit** $\rightarrow$ **read** list **Bool**,
      add : **Int** $\rightarrow$ **write** list **Unit**
    axiom forall i : **Int** •
      empty() $\equiv$ list := $\langle\rangle$,
      is_empty() $\equiv$ list = $\langle\rangle$,
      add(i) $\equiv$ list := $\langle i \rangle$ ^ list
  end

Extension

General form:

extend $\mathrm{class\_expr}_1$ **with** $\mathrm{class\_expr}_2$

$\mathrm{class\_expr}_1$ and $\mathrm{class\_expr}_2$ must be compatible

Context dependent expansion:

extend
  class $\mathrm{decl\_string}_1$ **end**
with
  class $\mathrm{decl\_string}_2$ **end**

expands to:

class
  $\mathrm{decl\_string}_1$
  $\mathrm{decl\_string}_2$
end

Renaming - Example

scheme STACK_S =
  use stack **for** list, push **for** add
  in LIST_S

expands to:

scheme STACK_S =
  class
    variable stack : **Int***
    value
      empty : **Unit** → **write** stack **Unit**,
      is_empty : **Unit** → **read** stack **Bool**,
      push : **Int** → **write** stack **Unit**,
    axiom forall i : **Int** •
      empty() ≡ stack := ⟨⟩,
      is_empty() ≡ stack = ⟨⟩,
      push(i) ≡ stack := ⟨i⟩ ^ stack,
  end

Renaming

General form:

use
  $\mathrm{id}_{\mathrm{new}_1}$ **for** $\mathrm{id}_{\mathrm{old}_1}$, ... ,$\mathrm{id}_{\mathrm{new}_n}$ **for** $\mathrm{id}_{\mathrm{old}_n}$
in class_expr

use
  ..., $\mathrm{id}_{\mathrm{new}}$ **for** $\mathrm{id}_{\mathrm{old}}$: type_expr, ...
in class_expr

Hiding

Hidden entities

1. are not visible outside
2. need not be implemented

Typically use:

1. prevention of unintended access to variables and/or channels
2. hiding of auxilliary functions

Hiding - Example

scheme ENCAPSULATED_LIST_S =
  hide list **in**
  class
    variable list : **Int***
    value
      empty : **Unit** → **write** list **Unit**,
      is_empty : **Unit** → **read** list **Bool**,
      add : **Int** → **write** list **Unit**
    axiom forall i : **Int** •
      empty() ≡ list := ⟨⟩,
      is_empty() ≡ list = ⟨⟩,
      add(i) ≡ list := ⟨i⟩ ^ list
  end

scheme ILLEGAL_S =
  extend
    ENCAPSULATED_LIST_S
  with
    class
      value length : **Unit** → **read** list **Nat**
      axiom length() ≡ **len** list
    end

Hiding - Example

scheme SORTING_LIST_S =
  hide is_sorted, is_permutation **in**
  extend LIST_S **with**
  class
    value
      sort : **Unit** → **write** list **Unit**
      sort()
      post is_sorted(list) ∧ is_permutation(list, list′),

      is_sorted : **Int*** → **Bool**,
      is_permutation : **Int*** × **Int*** → **Bool**
    axiom ...
  end

Hiding

General form:

hide id_1,...,id_n **in** class_expr

hide ..., id : type_expr, ... **in** class_expr

Objects

scheme LIST_S =
  class
    variable list : **Int***
    value
      is_empty : **Unit** → **read** list **Bool**
      ...
  end

object
  LIST1 : LIST_S,
  LIST2 : LIST_S,

  SYS :
    class
      value
        one_is_empty :
          Unit → **read** LIST1.list LIST2.list **Bool**
        one_is_empty() ≡
          LIST1.is_empty() ∨ LIST2.is_empty()
    end

LIST1.list and LIST2.list are two distinct variables.

Module Nesting

object
  SYS :
    class
      object
        LIST1 : LIST_S,
        LIST2 : LIST_S
      value
        one_is_empty :
          Unit → **read** LIST1.list LIST2.list **Bool**
        one_is_empty() ≡
          LIST1.is_empty() ∨ LIST2.is_empty()
    end

Parameterization - Example

scheme LIST_S =
  class
    type Elem
    variable list : Elem*
    value
      empty : **Unit** → **write** list **Unit**,
      add : Elem → **write** list **Unit**
    axiom forall e : Elem •
      empty() ≡ list := ⟨⟩,
      add(e) ≡ list := ⟨e⟩ ⌢ list
  end

is better expressed using parameterization

scheme PARAM_LIST(E : ELEMENT) =
  class
    variable list : E.Elem*
    value
      empty : **Unit** → **write** list **Unit**,
      add : E.Elem → **write** list **Unit**
    axiom forall e : E.Elem •
      empty() ≡ list := ⟨⟩,
      add(e) ≡ list := ⟨e⟩ ⌢ list
  end
scheme ELEMENT = **class type** Elem **end**

Instantiation - Example 1

object
  INTEGER :
    class
      type Elem = **Int**
    end,

  INTEGER_LIST : PARAM_LIST(INTEGER)

Last line is equivalent to (replace E with INTEGER):

  INTEGER_LIST :
    class
      variable list : INTEGER.Elem*
        empty : **Unit** → **write** list **Unit**,
        add : INTEGER.Elem → **write** list **Unit**
      axiom forall e : INTEGER.Elem •
        empty() ≡ list := ⟨⟩,
        add(e) ≡ list := ⟨e⟩ ⌢ list
    end

Instantiation - Example 2

object
  COMMAND :
    class
      type
        Key = **Nat**,
        Data = **Text**,
        Command ==
          mk_empty |
          mk_insert(Key, Data) |
          mk_remove(Key) |
          mk_lookup(Key)
    end,

  COMMAND_LIST :
    PARAM_LIST(COMMAND{Command **for** Elem})

E.Elem is replaced with
  COMMAND{Command **for** Elem}.Elem, i.e. with
  COMMAND.Command

More Complex Parameters

scheme
  EQUIVALENCE = **extend** ELEMENT **with**
    class
      value eq : Elem × Elem → **Bool**
      axiom forall e,e1,e2,e3 : Elem •
        [eq_reflexive]
          eq(e,e),
        [eq_transitive]
          eq(e1,e2) ∧ eq(e2,e3) ⇒ eq(e1,e3),
        [eq_symmetric]
          eq(e1,e2) ⇒ eq(e2,e1)
    end,

  PARTIAL_ORDER = **extend** EQUIVALENCE **with**
    class
      value leq : Elem × Elem → **Bool**
      axiom forall e1,e2,e3 : Elem •
        [leq_reflexive]
          eq(e1,e2) ⇒ leq(e1,e2),
        [leq_transitive]
          leq(e1,e2) ∧ leq(e2,e3) ⇒ leq(e1,e3),
        [leq_antisymmetric]
          leq(e1,e2) ∧ leq(e2,e1) ⇒ eq(e1,e2)
    end

scheme
  PARAM_ORDERED_LIST(T : PARTIAL_ORDER) =
    extend PARAM_LIST(T) **with**
    class
      value
        is_ordered : **Unit** $\rightarrow$ **read** list **Bool**
      axiom
        is_ordered() $\equiv$
          ($\forall$ idx1,idx2 : **Nat** $\bullet$
            {idx1,idx2} $\subseteq$ **inds** list $\wedge$ idx1 < idx2 $\Rightarrow$
              T.leq(list(idx1),list(idx2)))
    end

Instantiation - Example 3

scheme
  TEXT_S =
    class
      type Elem = **Text**
      value
        eq : **Text** $\times$ **Text** $\rightarrow$ **Bool**,
        leq : **Text** $\times$ **Text** $\rightarrow$ **Bool**
      axiom forall t1,t2 : **Text** $\bullet$
        eq(t1,t2) $\equiv$ t1 = t2,
        leq(t1,t2) $\equiv$ ($\exists$ t : **Text** $\bullet$ t1 $\hat{}$ t = t2)
    end

object
  TEXT : TEXT_S,

  TEXT_LIST : PARAM_ORDERED_LIST(TEXT)

Actual versus Formal Parameters

  scheme S(X : FC)
  object A : AC,

... S(A) ...

Context condition: AC must statically implement FC.
Confidence condition: AC must fully implement FC (AC $\preceq$ FC).

Implementation Relation

$$\boxed{\text{class\_expr}_1} \xrightarrow{\text{implemented by}} \boxed{\text{class\_expr}_2}$$

class_expr$_2$ implements class_expr$_1$

  class_expr$_2$ $\preceq$ class_expr$_1$

iff

1. class_expr$_2$ *statically implements* class_expr$_1$, i.e.: class_expr$_2$ defines at least all the names defined in class_expr$_1$ with same kinds and maximal types (statically decidable)
2. theory(class_expr$_1$) holds in class_expr$_2$ ($\Rightarrow$ implementation conditions)

theory(class_expr$_1$) is the collection of (explicit and implicit) axioms in class_expr$_1$.

Implementation Conditions: Example

$\llcorner$TEXT_S $\preceq$ PARTIAL_ORDER$\lrcorner$

- $\llcorner$eq(e,e)$\lrcorner$
- $\llcorner$eq(e1,e2) $\wedge$ eq(e2,e3) $\Rightarrow$ eq(e1,e3)$\lrcorner$
- $\llcorner$eq(e1,e2) $\Rightarrow$ eq(e2,e1)$\lrcorner$
- $\llcorner$eq(e1,e2) $\Rightarrow$ leq(e1,e2)$\lrcorner$
- $\llcorner$leq(e1,e2) $\wedge$ leq(e2,e3) $\Rightarrow$ leq(e1,e3)$\lrcorner$
- $\llcorner$leq(e1,e2) $\wedge$ leq(e2,e1) $\Rightarrow$ eq(e1,e2)$\lrcorner$

Two Ways of Building Hierarchies

1. using extend, e.g.

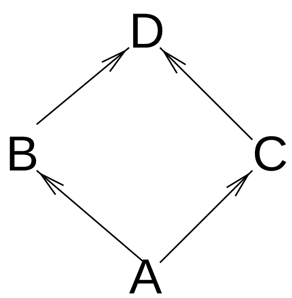
   extend A **with**
     class
       ...
     end

2. using objects, e.g.

   class
     object AO : A
     ...
   end

Building Hierarchies

Sharing versus copying



Be careful:
a module (D) must not have mutiple extensions of the same module (A), like in:

scheme
  B = **extend** A **with** ...,
  C = **extend** A **with** ...,
  D = **extend** B **with extend** C **with** ...

If A is to be **shared** by B and C, then this can be achieved by using one global A object AO:

object AO : A
scheme
  B = **class** ... uses AO... **end**,
  C = **class** ... uses AO... **end**,
  D = **extend** B **with extend** C **with** ...

If B and C should have each their **copy** of A, then this can be achieved by using two global A object AO1 and AO2:

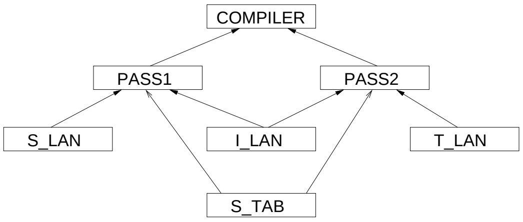
object AO1 : A, AO2 : A
scheme
  B = **class** ... uses AO1... **end**,
  C = **class** ... uses AO2... **end**,
  D = **extend** B **with extend** C **with** ...

In both cases D could alternatively have been built using B and C objects instead of extend:

  D =
    class
      object BO : B, CO : C
      ... uses BO and CO
    end

Sharing using Global Objects

Example



Sharing using Global Objects

Example

object
  S_LAN : **class type** Prog ... **end**,
  T_LAN : **class type** Instrs ... **end**,
  I_LAN : **class type** I_repr ... **end**,

  S_TAB : **class type** Table ... **end**

scheme
  PASS1 =
    class
      value
        pass1 :
          $\text{S\_LAN.Prog} \rightarrow \text{I\_LAN.I\_repr} \times \text{S\_TAB.Table}$
    end,

  PASS2 =
    class
      value
        pass2 :
          $\text{I\_LAN.I\_repr} \times \text{S\_TAB.Table} \rightarrow \text{T\_LAN.Instrs}$
    end,

COMPILER =
  class
    object
      P1 : PASS1,
      P2 : PASS2
    value
      compile : $\text{S\_LAN.Prog} \rightarrow \text{T\_LAN.Instrs}$
      compile(p) $\equiv$ P2.pass2(P1.pass1(p))
  end

Summary

Modules:

- schemes
- objects

Class expressions:

- basic
- extending
- hiding
- renaming
- instantiation