Introduction to RAISE/RSL

Contents

- RAISE Introduction — History etc. 17
- RSL Syntax Overview 38
- RSL Syntax Conventions 48

What is RAISE?

Rigorous **A**pproach to **I**ndustrial **S**oftware **E**ngineering

RAISE is a product consisting of:

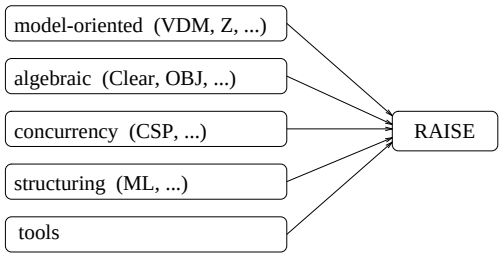
- a method for software development
- a formal specification language: RSL
- computer based tools

developed by:

- DDC/CRI (DK)
- STL/BNR (UK)
- ICL (UK)
- NBB/ABB/SYPRO (DK)

in an ESPRIT-I project, RAISE, 1985 - 1990

Background



Today: Many Formal Techniques & Design Calculi:

RAISE, VDM–SL, Z, B, LARCH, OBJ, Casl, CafeOBJ, CSP, CCS, . . .

RAISE Continuation I

ESPRIT-II project, **LaCoS**, 1990 - 1995

Large **S**cale **C**orrect **S**ystems Using Formal Methods

- industrial applications of RAISE by

BNR Europe (UK):	Network design toolset
Lloyd's Register (UK):	Ship engine monitoring
Bull (F):	Database
MATRA Transport (F):	Automatic train protection
Inisel Espacio (E):	Image processing
Space Software Italia (I):	Tethered satellite
Technisystems (GR):	Shipping transaction processing

- evolution of RAISE method, language and tools by

  CRI (DK)
  SYPRO (DK)
  BNR Europe (UK)

RAISE continuation II, 1995–

A new toolset has been developed at UNU/IIST in Macau, 1995–now.

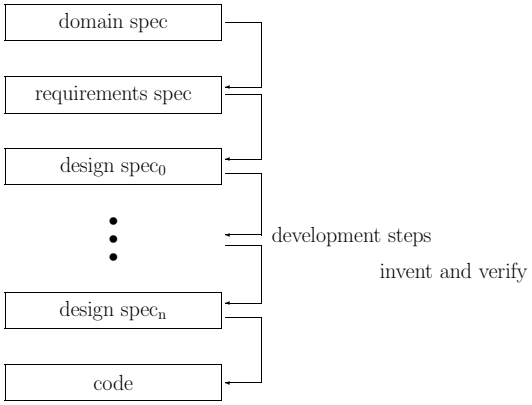
Todays centers of excellence:

IMM/DTU: Courses and research.

At UNU/IIST in Macau: Courses, research and tools development.

TERMA A/S: Books, tools and consultancy.

RAISE Method



Specification is formal (formulated in RSL)

Verification *might* be formal

Methods for Software Development

- Ad hoc
- Systematic
- Rigorous (R in RAISE: Rigorous)
- Formal

RAISE Specification Language, RSL

Features:

- Formal
- Wide spectrum
  - model-oriented and property-oriented
  - applicative and imperative
  - sequentiality and concurrency

RAISE Tools

`eden`: the original tool set for SUN workstations

`rsltc`: a new tool set for Windows and Linux providing:

- syntax and type checking
- showing module dependencies
- pretty printing (for LaTeX)
- translation from RSL to ML, C++, PVS and SAL
- translation to RSL from UML class diagrams
- formulation and execution of test cases
- proof support:
  - generation of confidence conditions,
  - formulation of user-defined proof obligations
  - translation to PVS and SAL

What is an RSL Specification?

Mathematical models describe real world

RSL specification ——————> program

Example: RSL Specification

Model-oriented

scheme SET_DATABASE =
  class
    type
      Database = Person-**set**,
      Person = **Text**

    value
      empty : Database = {},

      register : Person × Database → Database
      register(p,db) ≡ db ∪ {p},

      check : Person × Database → **Bool**
      check(p,db) ≡ p ∈ db
  end

RSL Specification

An RSL specification consists of

- module definitions

A module contains definitions of

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

Abstraction

"What" rather than "how"

RSL allows

- data (ie. representation) abstraction
- operation abstraction

Data abstraction, Examples

type Database = Person-**set**
type Database = Person*
type Database

type Person = **Text**
type Person = **Nat**
type Person

Operation abstraction, Examples

value
check : Person × Database → **Bool**
check(p,db) ≡ p ∈ db

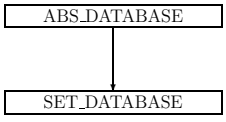
value
check : Person × Database → **Bool**
axiom
∀ p : Person •
check(p, empty) ≡ **false**,
∀ p : Person, db : Database •
check(p, register(p, db)) ≡ **true**
..

value
square_root : **Real** ~→ **Real**
square_root(r) **as** s
post s ∗ s = r ∧ s ≥ 0.0
pre r ≥ 0.0

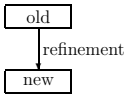
Example: RSL Specification
Property-oriented (algebraic)

scheme ABS_DATABASE =
class
type
Database,
Person
value
empty : Database,
register : Person × Database → Database,
check : Person × Database → **Bool**
axiom
∀ p : Person •
check(p, empty) ≡ **false**,
∀ p : Person, db : Database •
check(p, register(p, db)) ≡ **true**
..
end

A Development Step



Refinement Relation



1. new signature includes the old
   (statically decidable)
2. old properties hold in the new
   (⇒ proof obligations: "refinement conditions")

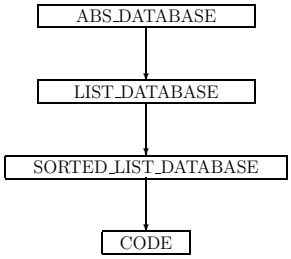
Refinement Conditions

⌞∀ p : Person • check(p, empty) ≡ **false**⌟
⌞∀ p : Person, db : Database •
check(p, register(p, db)) ≡ **true**⌟

Verification

⌞check(p, register(p, db)) ≡ **true**⌟
unfold register:
⌞check(p, db ∪ {p}) ≡ **true**⌟
unfold check:
⌞p ∈ (db ∪ {p}) ≡ **true**⌟
isin_union:
⌞p ∈ db ∨ p ∈ {p} ≡ **true**⌟
isin_singleton:
⌞p ∈ db ∨ p = p ≡ **true**⌟
equality_annihilation:
⌞p ∈ db ∨ **true** ≡ **true**⌟
or_true:
⌞**true** ≡ **true**⌟
is_annihilation:
⌞**true**⌟
qed

Alternative Development



Summary

RAISE

- Method
  - stepwise development
  - invent and verify
  - rigorous
- Specification language (RSL)
  - formal
  - wide spectrum
  - structuring facilities
- Tools

Course Contents

We will use RAISE to teach you about

- model-oriented and property-oriented (algebraic) specification styles
- applicative (functional) and imperative specification styles
- specification structuring facilities
- the development paradigm of stepwise refinement and verification

RSL Syntax Overview

Language Description

A language is a system of symbols described by

- syntax (combination of symbols)
- semantics (meaning)
- pragmatics (use)

syntactic object	denotes ——>	semantic object
1 + 2		3
value expr		value
+		addition
operator		function

Specifications

An RSL specification consists of

- module definitions

Module Definitions

scheme id =
  class
    declaration$_1$
    ⋮
    declaration$_n$
  end

A module contains declarations of

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

No special order of declarations is required.

Declarations

A declaration is a list of definitions of the same kind:

type, value, axiom, variable, channel,

scheme, object, e.g.:

type
  type_definition$_1$,
  ⋮
  type_definition$_n$
value
  value_definition$_1$,
  ⋮
  value_definition$_n$

Type Definitions

- sorts
  type id

  type Database

- abbreviations
  type id = type_expr

  type Database = Person**-set**

Types

Types are collections of related values.

Type Expressions

RSL provides type expressions for

- atomic types (**Int**)
- composite types (**Int-set**)
- subtypes ($\{|\ i : \mathbf{Int} \cdot i \geq 0\ |\}$)
- types given a name in type definitions (Person)

All types have associated built-in operators, including = and $\neq$.

Atomic types

Bool

values: **true**, **false**
operators and connectives: $=, \neq, \wedge, \vee, \Rightarrow, \sim$

Int

values: ..., -2, -1, 0, 1, 2, ...
operators: $=, \neq, +, -, *, /, \uparrow, \backslash, <, \leq, >, \geq$, **abs**, **real**

Nat = $\{|\ i : \textbf{Int} \cdot i \geq 0\ |\}$

Real

values: ..., -4.3, ..., 0.0, ..., 1.0, ...
operators: $=, \neq, +, -, *, /, \uparrow, <, \leq, >, \geq$, **abs**, **int**

Char

values: 'a', ...
operators: $=, \neq$

Text = **Char***

values: "Alice", ...

Unit

value: ()

More about **Bool** in next lecture.

Composite Types

- products (T1 × T2)
- functions (T1 → T2, T1 $\xrightarrow{\sim}$ T2)
- sets (T-**set**, T-**infset**)
- lists (T*, T$^\omega$)
- maps (T1 $\xrightarrow[m]{}$ T2, T1 $\xrightarrow[m]{\sim}$ T2)

More about these types in the comming lectures.

Value Definitions

- explicit

value
check : Person × Database → **Bool**
check(p,db) ≡ p ∈ db

- axiomatic

value
check : Person × Database → **Bool**
axiom
∀ p : Person •
check(p, empty) ≡ **false**,
∀ p : Person, db : Database •
check(p, register(p, db)) ≡ **true**
..

- implicit

value
square_root : **Real** $\xrightarrow{\sim}$ **Real**
square_root(r) **as** s
post s ∗ s = r ∧ s ≥ 0.0
pre r ≥ 0.0

Value expressions

Constructed from

- literals (1, **true**, ...)
- operators (+, ...)
- connectives (∧, ...)
- ids introd. in value definitions (empty, check, ...)
- function applications (f(x))
- if expressions (**if** ...)
- quantified expressions (∀, ...)
- equivalence expressions
- ...

RSL Syntax Conventions

set_type_expr ::=
  finite_set_type_expr|
  infinite_set_type_expr

finite_set_type_expr ::=
  type_expr-**set**

opt-x ::=
  nil-x|
  x

x-string ::=
  x|
  x x-string

x-list ::=
  x|
  x , x-list

Syntax — Continued

x-list2 ::=
  x , x|
  x , x-list2

x-choice ::=
  x|
  x | x-choice

x-choice2 ::=
  x | x|
  x | x-choice2

x-product2 ::=
  x × x|
  x × x-product2

x-product ::=
  x|
  x × x-product

Syntax — Continued

axiom_prefix_expr ::=
  • *logical*-value_expr

restriction ::=
  • *readonly_logical*-value_expr

Syntax — Continued

prefix	context condition
unit	must have type **Unit**
logical	must have type **Bool**
integer	must have type **Int**
list	must have a list type
map	must have a map type
function	must have a function type

prefix	context condition
pure	must be pure
readonly	must be read-only

prefix	context condition
type	must represent a type
value	must represent a value
variable	must represent a variable
channel	must represent a channel
scheme	must represent a scheme
object	must represent an object