# Course 02263
# Formal Aspects of Software Engineering

## *Introduction to RAISE*

Anne E. Haxthausen

DTU Informatics (IMM)
Technical University of Denmark
ah@imm.dtu.dk



## Contents





## What is RAISE?

**RAISE** = **R**igorous **A**pproach to **I**ndustrial **S**oftware **E**ngineering

**RAISE** is a product consisting of:

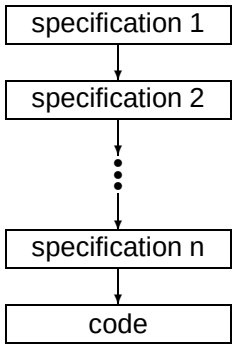
- a method for software development
- a formal specification language: RSL
- computer based tools (for writing, checking, analysing and translating specs)

developed by a number of companies in DK, UK, F, E, I, and GR in two EU ESPRIT projects, RAISE and LaCoS. Main contractor was DDC, later CRI A/S in Denmark. Has later been further developed at the UN university UNU/IIST.



## RAISE Method

Principle: stepwise refinement using invent-and-verify paradigm.




Specifications are formal (formulated in RSL).
Verification of refinement steps *might* be formal.

## RAISE Specification Language, RSL

Key features of RSL:

- is formal
- is wide spectrum, i.e. provides many styles in one language:
  - property-oriented (abstract) and model-oriented (concrete) styles
  - applicative and imperative styles
  - sequentiality and concurrency styles
- has structuring facilities (modules)

In this course will exploit all these features, except for 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



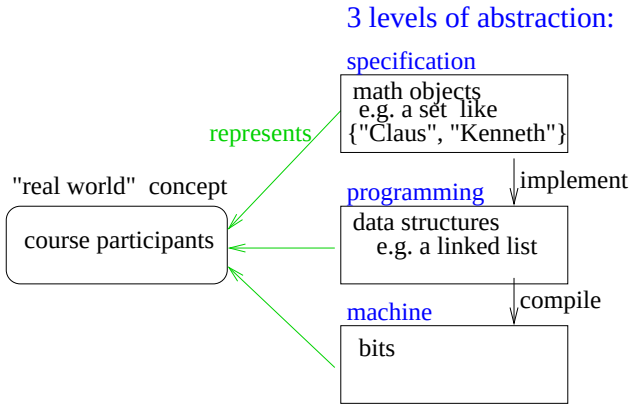
## Examples of industrial applications of RAISE

- 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
- ...



## RSL Specifications

RSL specifications are mathematical abstractions of software.



## Example: Model-oriented RSL Specification

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

    value
      empty : Database = {},

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

      is_in : Person × Database → Bool
      is_in(p,db) ≡ p ∈ db
  end
```



## Abstraction

"What" rather than "how"

RSL allows

- data (ie. representation) abstraction
- operation abstraction



## Data abstraction, Examples

- Some possible data abstractions of databases:

  **type** Database = Person-**set**
  **type** Database = Person$^*$
  **type** Database

- Some possible data abstractions of persons:

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



## Operation abstraction, Examples

```
value
  is_in : Person × Database → Bool
  is_in(p,db) ≡ p ∈ db
```

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

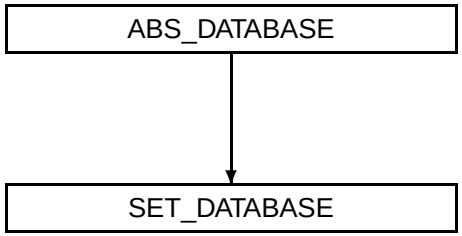
```
value
  square_root : Real ~→ Real
  square_root(r) as s
    post s ∗ s = r ∧ s ≥ 0.0
    pre r ≥ 0.0
```

## Example: Property-oriented RSL Specification

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



## A Development Step




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

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

    value
      empty : Database = {},

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

      is_in : Person × Database → Bool
      is_in(p,db) ≡ p ∈ db
  end
```



## Refinement relation

SET_DATABASE *refines* ABS_DATABASE if:

- SET_DATABASE provides declarations of (at least) the same types and values as ABS_DATABASE (statically decidable)
- all properties (axioms) of ABS_DATABASE hold in SET_DATABASE ⇒ proof obligations called *refinement conditions*:
  - ⌞∀ p : Person • is_in(p, empty) ≡ **false**⌟
  - ⌞∀ p : Person, db : Database • is_in(p, register(p, db)) ≡ **true**⌟



## Verification of a refinement condition

⌞is_in(p, register(p, db)) ≡ **true**⌟
unfold register:
⌞is_in(p, db ∪ {p}) ≡ **true**⌟
unfold is_in:
⌞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**

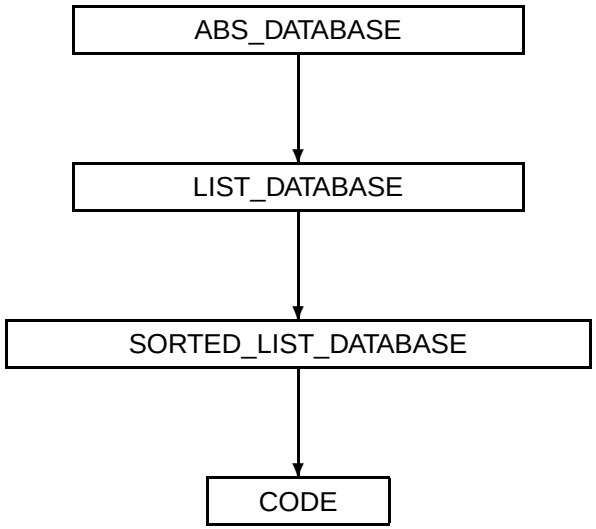
```
scheme SET_DATABASE =
  class
    type
      Database = Person-set,
      Person = Text

    value
      empty : Database = {},

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

      is_in : Person × Database → Bool
      is_in(p,db) ≡ p ∈ db
  end
```

## Alternative Development






## Summary

RAISE consists of

- a method
  - stepwise development
  - invent and verify
  - rigorous
- a 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



## Contents

## RSL Specifications

An RSL specification consists of

- module definitions



## RSL 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.



## RSL Declarations

A declaration is a list of definitions of the same kind:
**type, value, axiom, variable, channel, scheme, object**, e.g.:

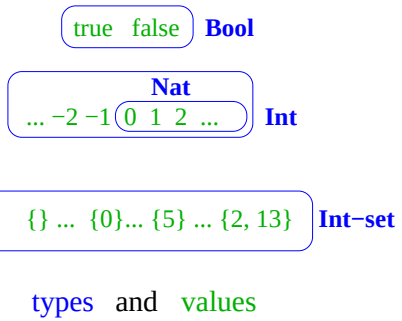
**type**
type_definition$_1$,
⋮
type_definition$_n$
**value**
value_definition$_1$,
⋮
value_definition$_n$

In the first part of the course we will only consider types, value and axioms.



## The concept of types and values

A type is a set of related values.



## RSL Type Definitions

Two kinds:

- *sort definitions*:

  **type** id

  just gives the name id to a type.

  Example: **type** Database

- *abbreviation definitions*:

  **type** id = T

  gives the name id to a type T.

  Example: **type** Database = Person**-set**



## RSL Type Expressions

Types T can be:

- *type names* introduced in other type definitions (e.g. Person)
- *atomic types* (e.g. **Int**)
- *composite types* (e.g. **Int-set**) built from other types by applying type constructors (e.g. **-set**) to these.

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



## RSL Atomic types

**Bool** values: **true**, **false**
operators: =, $\neq$, $\wedge$, $\vee$, $\Rightarrow$, $\sim$, $\forall$, $\exists$

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

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

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

**Char** values: ‘a’, . . .
operators: =, $\neq$

**Text** values: "Alice", . . .
operators: =, $\neq$, ...

More about **Bool** later.



## RSL Composite types: overview

| type | values |
|---|---|
| $T_1 \times ... \times T_n$ | products |
| T**-set** and T**-infset** | sets |
| $T^*$ and $T^\omega$ | lists |
| $T_1 \xrightarrow[m]{} T_2$ and $T_1 \xrightarrow[m]{\sim} T_2$ | maps |

where T, $T_1$,$T_2$ and $T_n$ are types

A *product* is an *ordered* collection of – *not necessarily distinct* – values of some (possibly different) given types.

A *set* is an *unordered* collection of *distinct* values of the same type.

A *list* (sequence) is an *ordered* collection of – *not necessarily distinct* – values elements of the same type.

A *map* (or table) is an unordered collection of pairs of values.

More about these types in the coming lectures.

## RSL Value Definitions

Three forms:

- *explicit function definition* as in

```
value
  is_in : Person × Database → Bool
  is_in(p,db) ≡  p ∈ db
```

- *implicit function definition* as in

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

- *axiomatic definition* as in

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



## RSL Value Expressions

Constructed from

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

Are e.g. used in explicit and implicit value definitions.



## RSL Reference Description and Grammar

In part II and III of the RSL book you find a reference description and a grammar for RSL.

During the course, please consult these parts when relevant.