Introduction to RAISE

Chris George

United Nations University
International Institute for Software Technology
Macao SAR, China

Macao

UNU-IIST

RAISE?

Radical **A**lternative for **I**nadequate **S**oftware **E**ngineers

Rambling **A**round **I**n **S**earch of **E**nlightenment

Rigorous Approach to Industrial Software Engineering

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

model-oriented (VDM, Z, ...)

property-oriented (Clear, ...)

concurrency (CSP, ...)

structuring (ML, ...)

tools

RAISE

RAISE Continuation

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

Large Scale **Co**rrect **S**ystems
Using Formal Methods

- industrial applications of RAISE
- evolution of
  RAISE method, language and tools

LaCoS Partners

Producers:

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

Consumers:

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

RAISE Specification Language (RSL)

Design objectives:

- Wide spectrum language
  - Abstract — concrete; specification — implementation in one language
  - Applicative and imperative styles
  - Sequential and concurrent styles
  - Maximal applicability (but better for information systems than control systems: time added later)
- Suitable for large descriptions; modular

Design objectives: type system

- Type checking simple:
  - decidable
  - minimal type inference required
  - separation of types and values (sets are not types)

Design objectives: user friendly

- User convenience preferred to tool writers' convenience:
  - No "define before use" restriction
  - Language tightly defined, nothing "implementation dependent" (such as evaluation order)
- Expressions have maximum expressivity; modular concepts are minimal
- Implementation relation is simple: just property preservation; no fitting morphisms
- Powerful logic: implementation relation can be expressed in RSL

Regularity

- Maximum reuse of binding, typing, pattern, etc.
- When a construct (expression, type, binding, etc.) is allowed, *any* form of the construct should be allowed.

Portability

ASCII syntax:

Sym	ASCII	Sym	ASCII	Sym	ASCII	Sym	ASCII
×	><	*	-list	ω	-inflist	→	->
$\xrightarrow{\sim}$	-~->	$\xrightarrow{m}$	-m->	$\xrightarrow[m]{\sim}$	-~m->	↔	<->
∧	/\	∨	\/	⇒	=>	∀	all
∃	exists	•	:-	□	always	≡	is
≠	~=	≤	<=	≥	>=	↑	**
∈	isin	∉	~isin	⊂	<<	⊆	<<=
⊃	>>	⊇	>>=	∪	union	∩	inter
†	!!	⟨	<.	⟩	.>	↦	+>
‖	||	∦	++	⟦	|=|	⟧	|^|
		λ	-\	°	#		

Conventions for tools

- Files have `.rsl` suffix
- One module per file
- Module name same as file base name

UNU-IIST RAISE tools

- Open source; Gnu Public Licence
- Written (effectively) in C, so very portable
- Command line tool using emacs to provide interface: aids portability

UNU-IIST RAISE tools: capabilities

- Type checking
- Pretty-printing
- Module dependency display (graph or table)
- Confidence condition generation
- Translation to SML and C++
- Translation to PVS for proofs
- Generation from UML class diagrams

Design achievements

- Unification of algebraic and model-based approaches
- Unification of channel-based concurrency with value passing

A simple example

```
scheme REGISTRATION =
  class
    type
      Database = Person-set,
      Person = Text
    value
      empty : Database = {},

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

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


Questions about REGISTRATION

- What happens if someone registers twice?
- What happens if two people have the same name?
- Could you use this to register the people for this course?
- Could you use it to register the people of China?

Another example

```
scheme SJH =
  class
    type
      Sdfv = Jdmjh-set,
      Jdmjh = Text
    value
      dfm : Sdfv = {},

      mjd : Jdmjh × Sdfv → Sdfv
      mjd(mn,dfmn) ≡ dfmn ∪ {mn},

      mjdwr : Jdmjh × Sdfv → Bool
      mjdwr(mn,dfmn) ≡ mn ∈ dfmn
  end
```

Characteristics of specifications

The two examples are isomorphic. To most mathematicians, this means they are the same.

Aims of specification (ordered):

1. Capture requirements precisely and clearly
2. Support the exploration of requirements; the raising of questions

42. Provide a basis for implementation

Specifications need *interpretation*, a relation between their types, values, modules, etc. and the real world.

Types and functions in RSL

Chris George

United Nations University

International Institute for Software Technology

Macao SAR, China

Types in RSL

Types may be abstract or concrete (and the two may be mixed)

```
type
  Database = Key m→ Data,
  Key,
  Data
```

Key and Data are abstract types: no definitions. Database is concrete — it is defined as the finite mapping (many-one relation) from Key to Data. Database could also be abstract; Key and Data could be concrete.

Both concrete and abstract types come with a built-in equality relation on their values.

Built-in types

- **Bool**
- **Int**
- **Nat** ($= \{|\, i : \textbf{Int} \bullet i \geq 0 \,|\}$)
- **Real**
- **Char**
- **Text** ($=$ **Char***)
- **Unit**

Bool	values:	**true**, **false**
	connectives:	$\wedge$, $\vee$, $\Rightarrow$, $\sim$
Int, **Nat**	values:	..., -2, -1, 0, 1, 2, ...
	operators:	$+$, $-$, $*$, /, $\uparrow$, $\backslash$, $<$, $\leq$, $>$, $\geq$, **abs**, **real**
Real	values:	..., -4.3, ..., 0.0, ..., 1.0, ...
	operators:	$+$, $-$, $*$, /, $\uparrow$, $<$, $\leq$, $>$, $\geq$, **abs**, **int**
Char	values:	$'a'$, ...
Text	values:	$''$Alice$''$, ...
	operators:	**hd**, **tl**, $\frown$, $\_(\_)$, **len**, **inds**, **elems**
Unit	value:	()

Type constructors

Product:	T × U, T × U × V, ...	(t,u), (t,u,v)
Set:	T-**set**, T-**infset**	{}, $\{t_1,t_2\}$
List:	T*, T^{ω}	⟨⟩, $\langle t_1,t_2\rangle$
Map:	T $\overrightarrow{m}$ U, T $\overset{\sim}{\overrightarrow{m}}$ U	[ ], $[t_1 \mapsto u_1, t_2 \mapsto u_2]$
Function:	T → U, T $\overset{\sim}{\rightarrow}$ U	λ x:T • u(x)

Integer sets and lists have ranged values, such as $\{0..10\}$, and $\langle 1..12\rangle$.

Sets, lists, and maps have comprehended values, such as
{ 2∗x+1 | x : **Int** • x ∈ {0..4} },
⟨ x | x **in** ⟨0..10⟩ • is_odd(x)⟩, and
[f(x) ↦ g(x) | x : **Int** • p(x)]

Products

A product is

an ordered finite collection
of
values of possibly different types

Examples:

(1,2)
(1,**true**,"John")

Product Type Expressions

type_expr$_1$ × ... × type_expr$_n$, n ≥ 2

Values:

(v_1,...,v_n), v_i : type_expr$_i$

Operators:

=
≠

Example: A System of Coordinates I

```
SYSTEM_OF_COORDINATES =
  class
    type
      Position = Real × Real
    value
      origin : Position = (0.0,0.0),

      distance : Position × Position → Real
      distance((x1,y1),(x2,y2)) ≡
        ((x2−x1)↑2.0 + (y2−y1)↑2.0)↑0.5
  end
```

Example: A System of Coordinates II

```
SYSTEM_OF_COORDINATES =
  class
    type Position = Real × Real
    value
      origin : Position = (0.0,0.0),
      distance : Position × Position → Real
      distance(p1, p2) ≡
        let
          (x1,y1) = p1,
          (x2,y2) = p2
        in ((x2−x1)↑2.0 + (y2−y1)↑2.0)↑0.5
        end
  end
```


Records: example 1

```
SYSTEM_OF_COORDINATES =
  class
    type
      Position ::
          x_coord : Real
          y_coord : Real
    value
      origin : Position = mk_Position(0.0,0.0),
      distance : Position × Position → Real
      distance(p1, p2) ≡
        ((x_coord(p2) − x_coord(p1))↑2.0 +
         (y_coord(p2) − y_coord(p1))↑2.0)↑0.5
  end
```


Records: example 2

```
type
  Book ::
        title: Title
        author : Author
        date : YearMonth
        price : Real ↔ change_price,
  YearMonth ::
        year : Year
        month : Month,
  Month = {| n : Nat • n ∈ {1..12} |}
```

mk_Book is a constructor of type Title × ... × **Real** → Book
title is a destructor of type Book → Title
change_price is a reconstructor of type **Real** × Book → Book

Variants

```
type
  Cowboy == good | bad | ugly,
  OptId == no_id | an_id(id : Id),
  Tree == nil | node(left : Tree, val : Val, right : Tree)
```

good, bad, ugly, no_id, an_id, nil, and node are constructors
id, left, val, and right are destructors.

nil has type Tree
node has type Tree × Val × Tree → Tree
val has type Tree $\xrightarrow{\sim}$ Val

Only variant type definitions may be recursive.

Case expressions

Lists and variants are often analysed by case expressions, as in:

```
value
  reverse : T* → T*
  reverse(t) ≡
    case t of
      ⟨⟩ → ⟨⟩,
      ⟨h⟩^t → reverse(t)^⟨h⟩
    end,
  traverse : Tree → Val*
  traverse(t) ≡
    case t of
      nil → ⟨⟩,
      node(l, v, t) → traverse(l) ^ ⟨v⟩ ^ traverse(t)
    end
```


Partial functions: example

```
value
  factorial : Nat ~→ Nat
  factorial(n) ≡
    if n = 1 then 1 else n * factorial(n − 1) end
  pre n > 0
```

A partial function has $\tilde{\rightarrow}$ in its signature and **pre** in its definition.

Implicit functions: example

```
value
  square_root : Real ~→ Real
  square_root(x) as r post r * r = x
  pre x ≥ 0.0
```

An implicit function uses **post**, usually with **as**, in its definition.

A better square_root specification?

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

An even better square_root specification?

```
value
  square_root : Real × Real ~→ Real
  square_root(x, ϵ) as r post abs(r ∗ r − x) ≤ ϵ ∧ r ≥ 0.0
  pre x ≥ 0.0 ∧ ϵ > 0.0
```


What about this specification?

```
value
  square_root : Real × Real ~→ Real
  square_root(x, ϵ) ≡
    if x = 0.0 then 0.0 else newton_raphson(x, ϵ, x/2.0) end
  pre x ≥ 0.0 ∧ ϵ > 0.0,

  newton_raphson : Real × Real × Real ~→ Real
  newton_raphson(x, ϵ, r) ≡
    if abs(r ∗ r − x) ≤ ϵ then r
    else newton_raphson(x, ϵ, (r + x/r)/2.0) end
  pre x ≥ 0.0 ∧ ϵ > 0.0 ∧ r > 0.0
```


One more version

```
value
  newton_raphson : Real × Real × (Real ~→ Real) × (Real ~→ Real) ~→ Real
  newton_raphson(r, ϵ, f, f′) ≡
    if abs(f(r)) ≤ ϵ then r
    else
      let r1 = r − f(r) / f′(r) in
        newton_raphson(r1, ϵ, f, f′)
      end
    end
  pre ϵ > 0.0 ∧ f′(r) ≠ 0.0,
```



```
value
  square_root : Real × Real ~→ Real
  square_root(x, ϵ) ≡
    if x = 0.0 then 0.0
    else
      let
        f = λ a : Real • a ∗ a − x,
        f′ = λ a : Real • 2.0 ∗ a
      in newton_raphson(x − f(x)/f′(x), ϵ, f, f′)
      end
    end
  pre x ≥ 0.0 ∧ ϵ > 0.0
```

```
value
  cube_root : Real × Real ~→ Real
  cube_root(x, ϵ) ≡
    if x = 0.0 then 0.0
    else
      let
        f = λ a : Real • a * a * a − x,
        f′ = λ a : Real • 3.0 * a * a
      in newton_raphson(x − f(x)/f′(x), ϵ, f, f′)
      end
    end
  pre x ≥ 0.0 ∧ ϵ > 0.0
```


Quiz 1

Letters could be Roman letters, Arabic letters, etc. There is a special letter *nil* used to indicate the end of a word. Words are lists of letters that satisfy *is_wf_Word*:

```
type Word = {| w : Letter* • is_wf_Word(w) |}
value
  is_wf_Word : Letter* → Bool
  is_wf_Word(w) ≡
    len w > 0 ∧ w(len w) = nil ∧
    (∀ i : Nat • i ≥ 1 ∧ i < len w ⇒ w(i) ≠ nil)
```

Lists are indexed from 1 in RSL: *w(1)* is the first element of the list *w*.

1. Is the list ⟨*nil*⟩ a word?
2. How many nils can there be in a word?

Quiz 2

What is the logical error in the following?

```
value
  /∗ check first n letters are the same ∗/
  match_n : Word × Word × Nat ~→ Bool
  match_n(w1, w2, n) ≡ first_n(w1, n) = first_n(w2, n)
  pre n ≤ len w1 ∧ n ≤ len w2,

  /∗ select the first n letters of a word ∗/
  first_n : Word × Nat ~→ Word
  first_n(w, n) ≡
    if n = 0 then ⟨⟩ else ⟨hd w⟩^first_n(tl w, n−1) end
  pre n ≤ len w
```


hd *w* gives the head (first element) of a list *w*, and **tl** *w* gives the tail (the list *w* with its head removed).

Hints:

- Read the code carefully
- Try checking confidence conditions
- Try some test cases. Try setting Letter to Char, nil to '0', and execute

```
test_case
[t1] first_n("abc0", 1),
[t2] first_n("abc0", 2),
[t3] first_n("abc0", 3),
[t4] first_n("abc0", 4)
```

Use the SML translator and the C++ translator. Any differences?

Exercises

These exercises are based on the type Tree defined by

type Tree $==$ nil | node(left : Tree, val : **Int**, right : Tree)

1. Define a function ‘depth’ that returns the depth of a tree.
2. Define a function ‘is_in’ to find if an integer is in a tree.
3. Define the subtype ‘Ordered_tree’. The subtype should not allow repetitions, so that an ordered tree models a set.
4. Define a function ‘is_in_ordered’ to find if an integer is in an ordered tree.
5. Define a total function ‘add’ to add an integer to an ordered tree.
6. Define a total function ‘remove’ to remove an integer from an ordered tree.

Sets, lists, and maps

Chris George

United Nations University
International Institute for Software Technology
Macao SAR, China

Sets

- finite and infinite sets
- set type expressions
- set operators
- set value expressions
- example of specification using sets

Sets

A set is:

an unordered collection
of
values of same type

Examples:

{1,3,5}
{"John","Peter","Ann"}

Set Type Expressions

- type_expr**-set**

  $\{v_1, \ldots, v_n\}$

  where n $\geq$ 0, v_i : type_expr

- type_expr**-infset**

  $\{v_1, \ldots, v_n\}$,
  $\{v_1, \ldots, v_n, \ldots\}$

  where n $\geq$ 0, v_i : type_expr

Associated Built-in Operators

$\cup$: T-**infset** $\times$ T-**infset** $\to$ T-**infset**	$\{1, 2, 3\} \cup \{3, 4\} = \{1, 2, 3, 4\}$
$\cap$: T-**infset** $\times$ T-**infset** $\to$ T-**infset**	$\{1, 2, 3\} \cap \{3, 4\} = \{3\}$
$\setminus$: T-**infset** $\times$ T-**infset** $\to$ T-**infset**	$\{1, 2, 3\} \setminus \{3, 4\} = \{1, 2\}$
$\in$: T $\times$ T-**infset** $\to$ **Bool**	$4 \in \{1, 2, 3\} =$ **false**
$\notin$: T $\times$ T-**infset** $\to$ **Bool**	$4 \notin \{1, 2, 3\} =$ **true**

$\subset$: T-**infset** $\times$ T-**infset** $\to$ **Bool**	$\{1, 3\} \subset \{1, 2, 3\} =$ **true** $\{1, 2, 3\} \subset \{1, 2, 3\} =$ **false**
$\subseteq$: T-**infset** $\times$ T-**infset** $\to$ **Bool**	$\{1, 3\} \subseteq \{1, 2, 3\} =$ **true** $\{1, 2, 3\} \subseteq \{1, 2, 3\} =$ **true** $\{1, 2, 3\} \subseteq \{1, 3\} =$ **false**
$\supset$ and $\supseteq$ are similar	
card : T-**infset** $\xrightarrow{\sim}$ **Nat**	**card** $\{1, 2, 5, 2, 2, 1, 5\} = 3$ **card** $\{n \mid n :$ **Nat**$\} \equiv$ **chaos**

Overloading of hd

Theory:

hd : T-**infset** $\xrightarrow{\sim}$ T
hd(s) **as** x **post** x $\in$ s
pre s $\neq \{\}$

Example:

hd {1,2} $\in$ {1,2}

i.e.

hd {1,2} $= 1 \vee$ **hd** {1,2} $= 2$

NB The overloading of **hd** for sets was added after the RSL book and method book were written.

Set Value Expressions

Enumerated: {expr$_1$,...,expr$_n$}

{1,2}
{1,2,1}

Ranged: {*integer*-expr$_1$ .. *integer*-expr$_2$}

{3 .. 7} = {3,4,5,6,7}
{3 .. 3} = {3}
{3 .. 2} = {}

Comprehended: {expr$_1$ | typing$_1$,...,typing$_n$ • *logical*-expr$_2$}

{2∗n | n : **Nat** • n $\leq$ 3}

```
RESOURCE_MANAGER =
  class
    type
      Resource,
      Pool = Resource-set

    value
      obtain : Pool ~→ Pool × Resource
      obtain(p) as (p1,r1) post r1 ∈ p ∧ p1 = p \ {r1}
      pre p ≠ {},

      release : Resource × Pool ~→ Pool
      release(r,p) ≡ p ∪ {r}
      pre r ∉ p
  end
```


Lists

- finite and infinite lists
- list type expressions
- list value expressions
- list indexing
- list operators
- example of specification using lists

Lists

A list is:

an ordered collection of values of same type

Examples:

⟨1,3,3,1,5⟩
⟨**true**,**false**,**true**⟩

List Type Expressions

- type_expr*

  $\langle v_1, ..., v_n \rangle$

  where n ≥ 0, v_i : type_expr

- type_expr$^\omega$

  $\langle v_1, ..., v_n \rangle$,
  $\langle v_1, ..., v_n, ... \rangle$

  where n ≥ 0, v_i : type_expr

List Value Expressions

Enumerated: ⟨expr$_1$,...,expr$_n$⟩

⟨1,3,3,1,5⟩
⟨**true**,**false**,**true**⟩

Ranged: ⟨*integer*-expr$_1$ .. *integer*-expr$_2$⟩

⟨3 .. 7⟩ = ⟨3,4,5,6,7⟩
⟨3 .. 3⟩ = ⟨3⟩
⟨3 .. 2⟩ = ⟨⟩

Comprehended: ⟨expr$_1$ | binding **in** *list*-expr$_2$ • *logical*-expr$_3$⟩

⟨2∗n | n **in** ⟨0 .. 3⟩⟩
⟨n | n **in** ⟨0 .. 100⟩ • is_even(n)⟩

List Indexing

Basic form:

list-expr(*integer*-expr$_1$)

Example:

⟨2,5,3⟩(2) = 5

Associated Built-in Operators

$\frown$: T* × T$^\omega$ → T$^\omega$	⟨e$_1$,...,e$_n$⟩ $\frown$ ⟨e$_{n+1}$,... ⟩ = ⟨e$_1$,...,e$_n$,e$_{n+1}$,... ⟩
hd : T$^\omega$ $\xrightarrow{\sim}$ T	**hd** ⟨e$_1$,e$_2$,... ⟩ = e$_1$
tl : T$^\omega$ $\xrightarrow{\sim}$ T$^\omega$	**tl** ⟨e$_1$,e$_2$,... ⟩ = ⟨e$_2$,... ⟩
len : T$^\omega$ $\xrightarrow{\sim}$ **Nat**	**len** ⟨e$_1$,...,e$_n$⟩ = n **len** il ≡ **chaos**
elems : T$^\omega$ → T-**infset**	**elems** ⟨e$_1$,e$_2$,... ⟩ = {e$_1$,e$_2$,... }
inds : T$^\omega$ → **Nat-infset**	**inds** fl = {1 .. **len** fl} **inds** il = {idx	idx : **Nat** • idx ≥ 1}

Overloading of ∈ and ∉

∈ : T × T$^\omega$ → **Bool**	′d′ ∈ ⟨′a′, ′b′, ′c′⟩ = **false**
∉ : T × T$^\omega$ → **Bool**	′a′ ∉ ⟨′a′, ′b′, ′c′⟩ = **false**

NB The overloading of ∈ and ∉ for lists (and maps) was added after the RSL book and method book were written.

```
QUEUE =
  class
    type
      Element,
      Queue = Element*
    value
      empty : Queue = ⟨⟩,

      enq : Element × Queue → Queue
      enq(e,q) ≡ q ⌢ ⟨e⟩,

      deq : Queue ~→ Queue × Element
      deq(q) ≡ (tl q,hd q)
      pre q ≠ empty
  end
```



```
SORTING = class
  value
    sort : Int* → Int*
    sort(l) as l1 post is_permutation(l1,l) ∧ is_sorted(l1)

    is_permutation : Int* × Int* → Bool,
    is_permutation(l1,l2) ≡ (∀ i : Int • count(i, l1) = count(i, l2)),

    count : Int × Int* → Nat
    count(i, l) ≡ card {idx | idx : Nat • idx ∈ inds l ∧ l(idx) = i},

    is_sorted : Int* → Bool
    is_sorted(l) ≡
      (∀ idx1,idx2 : Nat • {idx1,idx2}⊆inds l ∧ idx1<idx2 ⇒ l(idx1)≤l(idx2))
end
```


Maps

- map type expressions
- map value expressions
- map application
- map operators
- example of specification using maps

Maps

A map is:

an unordered collection
of
pairs of values

Examples:

[″Klaus″ ↦ 7, ″John″ ↦ 2, ″Mary″ ↦ 7]
[1 ↦ 2, 5 ↦ 10]

Maps may be:

- infinite
- partial
- non-deterministic

Map Type Expressions

- type_expr$_1$ $\overrightarrow{m}$ type_expr$_2$

  $[v_1 \mapsto w_1, \ldots, v_n \mapsto w_n]$

  where n $\geq$ 0, v_i : type_expr$_1$, w_i : type_expr$_2$
  and $v_i = v_j \Rightarrow w_i = w_j$
  Finite and deterministic when applied to elements in the domain

- type_expr$_1$ $\overset{\sim}{\overrightarrow{m}}$ type_expr$_2$

  $[v_1 \mapsto w_1, \ldots, v_n \mapsto w_n]$,
  $[v_1 \mapsto w_1, \ldots, v_n \mapsto w_n, \ldots]$,

  where n $\geq$ 0, v_i : type_expr$_1$, w_i : type_expr$_2$
  May be infinite and may be non-deterministic when applied to elements in the domain

NB The original RSL book only has $\overrightarrow{m}$, but with the meaning of $\overset{\sim}{\overrightarrow{m}}$. Finite maps were introduced and the symbols changed in the method book.

Examples

Nat $\overrightarrow{m}$ **Bool**

[ ]
[0 $\mapsto$ **true**]
[0 $\mapsto$ **true**, 1 $\mapsto$ **true**]

Nat $\overset{\sim}{\overrightarrow{m}}$ **Bool**

[ ]
[0 $\mapsto$ **true**]
[0 $\mapsto$ **true**, 1 $\mapsto$ **true**]
[0 $\mapsto$ **true**, 0 $\mapsto$ **false**]
[0 $\mapsto$ **true**, 0 $\mapsto$ **false**, 1 $\mapsto$ **true**]

Map Value Expressions

Enumerated: [expr$_1$ ↦ expr$_1$′,...,expr$_n$ ↦ expr$_n$′]

[3 ↦ **true**, 5 ↦ **false**]
[″Klaus″ ↦ 7, ″John″ ↦ 2, ″Mary″ ↦ 7]

Comprehended: [expr$_1$ ↦ expr$_2$ | typing$_1$,...,typing$_n$ • *logical*-expr$_3$]

[n ↦ 2∗n | n : **Nat** • n ≤ 2] = [0 ↦ 0, 1 ↦ 2, 2 ↦ 4]
[n ↦ 2∗n | n : **Nat**] = [0 ↦ 0, 1 ↦ 2, 2 ↦ 4, ...]

Map Application

Basic form:

map-expr(expr$_1$)

Examples:

[″Klaus″ ↦ 7, ″John″ ↦ 2, ″Mary″ ↦ 7](″John″) = 2

[3 ↦ **true**, 3 ↦ **false**](3) = **true** ⨅ **false**

Application is always to values in the domain; otherwise the result is non-terminating (in fact **swap**, a kind of deadlock).

Associated Built-in Operators

dom : (T$_1$ $\xrightarrow[m]{\sim}$ T$_2$) → T$_1$-**infset**	**dom** [3 ↦ **true**, 5 ↦ **false**] = {3, 5} **dom** [3 ↦ **true**, 5 ↦ **false**, 5 ↦ **true**] = {3, 5}
rng : (T$_1$ $\xrightarrow[m]{\sim}$ T$_2$) → T$_2$-**infset**	**rng** [3 ↦ **false**, 5 ↦ **false**] = {**false**} **rng** [3 ↦ **false**, 5 ↦ **false**, 5 ↦ **true**] = {**false**, **true**}
† : (T$_1$ $\xrightarrow[m]{\sim}$ T$_2$) × (T$_1$ $\xrightarrow[m]{\sim}$ T$_2$) → (T$_1$ $\xrightarrow[m]{\sim}$ T$_2$)	[3 ↦ **true**, 5 ↦ **false**] † [5 ↦ **true**] = [3 ↦ **true**, 5 ↦ **true**]
∪ : (T$_1$ $\xrightarrow[m]{\sim}$ T$_2$) × (T$_1$ $\xrightarrow[m]{\sim}$ T$_2$) → (T$_1$ $\xrightarrow[m]{\sim}$ T$_2$)	[3 ↦ **true**, 5 ↦ **false**] ∪ [5 ↦ **true**] = [3 ↦ **true**, 5 ↦ **false**, 5 ↦ **true**]

\ : (T$_1$ $\xrightarrow[m]{\sim}$ T$_2$) × T$_1$-**infset** → (T$_1$ $\xrightarrow[m]{\sim}$ T$_2$)	m \ s = [d ↦ m(d) | d : T$_1$ • d ∈ **dom** m ∧ d ∉ s] [3 ↦ **true**, 5 ↦ **false**] \ {5, 7} = [3 ↦ **true**]
/ : (T$_1$ $\xrightarrow[m]{\sim}$ T$_2$) × T$_1$-**infset** → (T$_1$ $\xrightarrow[m]{\sim}$ T$_2$)	m / s = [d ↦ m(d) | d : T$_1$ • d ∈ **dom** m ∧ d ∈ s] [3 ↦ **true**, 5 ↦ **false**] / {5, 7} = [5 ↦ **false**]
° : (T$_2$ $\xrightarrow[m]{\sim}$ T$_3$) × (T$_1$ $\xrightarrow[m]{\sim}$ T$_2$) → (T$_1$ $\xrightarrow[m]{\sim}$ T$_3$)	m$_1$ ° m$_2$ = [x ↦ m$_1$(m$_2$(x)) | x : T$_1$ • x ∈ **dom** m$_2$ ∧ m$_2$(x) ∈ **dom** m$_1$] [3 ↦ **true**, 5 ↦ **false**] ° [″Klaus″ ↦ 3, ″John″ ↦ 7] = [″Klaus″ ↦ **true**]
∈ : T$_1$ × (T$_1$ $\xrightarrow[m]{\sim}$ T$_2$) → **Bool**	3 ∈ [3 ↦ **true**] = **true**

```
DATABASE =
  class
    type
      Database = Key →m Data,
      Key, Data
    value
      empty : Database = [ ],

      insert : Key × Data × Database → Database
      insert(k,d,db) ≡ db † [ k ↦ d ],

      remove : Key × Database → Database
      remove(k,db) ≡ db \ {k},
```



```
      defined : Key × Database → Bool
      defined(k,db) ≡ k ∈ db,

      lookup : Key × Database ~→ Data
      lookup(k,db) ≡ db(k)
      pre defined(k,db)
  end
```

RAISE logic

Chris George

United Nations University
International Institute for Software Technology
Macao SAR, China

Computing involves *partial* functions

- division
- head of a list
- loops

So we need a logic that can deal with expressions that may not be well defined.

By *well defined* we mean has (or evaluates to) a value.

Expressions and values

An expression may or may not evaluate to a value:

Expression	Value
true	**true**
$1 + 0$	1
1 / 0	?
factorial(3)	6
factorial(-1)	?
factorial(x)	?
if x $>$ 0 **then** factorial(x) **else** 0 **end**	$\surd$
while true do skip end	$\times$

chaos

Used to represent undefinedness

while true do skip end $\equiv$ **chaos**

/ : **Real** $\times$ **Real** $\xrightarrow{\sim}$ **Real**

1.0/0.0 is under-specified

1.0/0.0 might evaluate to **chaos**

f : **Real** $\rightarrow$ **Real**
f(x) $\equiv$ **if** x $\neq$ 0.0 **then** 1.0/x **else** 0.0 **end**

If expressions

Example:

if x $>$ 0 **then** factorial(x) **else** 0 **end**

More general form:

if *logical*-expr **then** expr$_1$ **else** expr$_2$ **end**

Properties:

if true then expr$_1$ **else** expr$_2$ **end** $\equiv$ expr$_1$
if false then expr$_1$ **else** expr$_2$ **end** $\equiv$ expr$_2$
if chaos then expr$_1$ **else** expr$_2$ **end** $\equiv$ **chaos**

Non-strictness:

if true then expr$_1$ **else chaos end** $\equiv$ expr$_1$
if false then chaos else expr$_2$ **end** $\equiv$ expr$_2$

Connectives

Definitions:

$\sim$e $\equiv$ **if** e **then false else true end**

e1 $\wedge$ e2 $\equiv$ **if** e1 **then** e2 **else false end**

e1 $\vee$ e2 $\equiv$ **if** e1 **then true else** e2 **end**

e1 $\Rightarrow$ e2 $\equiv$ **if** e1 **then** e2 **else true end**

gives conditional logic

Truth tables

$\wedge$	**true**	**false**	**chaos**
true	**true**	**false**	**chaos**
false	**false**	**false**	**false**
chaos	**chaos**	**chaos**	**chaos**

$\vee$	**true**	**false**	**chaos**
true	**true**	**true**	**true**
false	**true**	**false**	**chaos**
chaos	**chaos**	**chaos**	**chaos**

$\Rightarrow$	**true**	**false**	**chaos**
true	**true**	**false**	**chaos**
false	**true**	**true**	**true**
chaos	**chaos**	**chaos**	**chaos**

Note:

e1 $\wedge$ e2 $\equiv$ e2 $\wedge$ e1
e1 $\vee$ e2 $\equiv$ e2 $\vee$ e1

are not tautologies

Quantified expressions

Examples:

$\forall$ x : **Nat** • (x = 0) $\vee$ (x $>$ 0)

$\exists$ x : **Int** • x = 7

$\exists!$ x : **Int** • (x $\geq$ 0) $\wedge$ (x $\leq$ 0)

$\forall$ x : **Nat** • x = -7

$\forall$ x, y : **Nat** • ($\exists!$ z : **Nat** • x+y = z)

General form:

quantifier typing$_1$, ... , typing$_n$ • *logical*-expr

All quantification is over values in the types stated,
i.e. not over **chaos**.

$\equiv$ versus $=$

Assume e_1 and e_2 are defined, deterministic, without effects and without communication.

Assume e_i evaluates to v_i, $v_1 \neq v_2$.

$\equiv$	e_1	e_2	**chaos**
e_1	**true**	**false**	**false**
e_2	**false**	**true**	**false**
chaos	**false**	**false**	**true**

$=$	e_1	e_2	**chaos**
e_1	**true**	**false**	**chaos**
e_2	**false**	**true**	**chaos**
chaos	**chaos**	**chaos**	**chaos**

$=$ and $\equiv$ differ in terms of :

- '$\equiv$' is two valued — the result is never **chaos**
- '$=$' is strict, '$\equiv$' is not
- '$\equiv$' is reflexive, '$=$' is not

Use of "$\equiv$ true"

Note that "p $\equiv$ **true**" is **true** is **true** if p is, **false** otherwise, and so is always defined.

"$\equiv$ **true**" is implicitly included in some logical expressions in RSL to ensure definedness:

- axioms
- predicates in quantified expressions
- predicates in comprehended expressions
- pre and post conditions

Conditional logic: conclusions

- **Pro**
  - Deals with undefinedness
  - Logical connectives are executable
- **Con**
  - Some classical laws require definedness:
    - "excluded middle"
    - commutativity of $\wedge$ and $\vee$

Total and partial functions

total functions:

$\text{type\_expr}_1 \rightarrow \text{type\_expr}_2$

partial functions:

$\text{type\_expr}_1 \xrightarrow{\sim} \text{type\_expr}_2$

$\forall\ f_{tot} : T_1 \rightarrow T_2, f_{par} : T_1 \xrightarrow{\sim} T_2, x : T_1 \bullet$

	defined (not **chaos**)	deterministic
$f_{tot}(x)$	yes	yes
$f_{par}(x)$	might be	might be

$\exists!\ y : T_2 \bullet f_{tot}(x) \equiv y$

Proof rules for RAISE

Chris George

United Nations University
International Institute for Software Technology
Macao SAR, China

Proof rules: purpose

- Provide formation rules to determine if a specification is well-formed
- Provide rules for reasoning:
  - is a predicate true?
  - are two terms equivalent?
  - is one specification a refinement of another?

Axiomatic and denotational semantics

The proof rules provide an axiomatic semantics.

There is also a denotational semantics. Why?

- provides a model for the axiomatic semantics
- hence shows the axiomatic semantics is consistent

Proof rules for definition: example

$$\frac{\text{context} \vdash \text{value\_expr} :\preceq \text{opt\_access\_desc\_string } \textbf{Bool} \quad \text{context} \vdash \textbf{read-only-convergent } \text{value\_expr}}{\text{context} \vdash \textbf{if } \text{value\_expr } \textbf{then } \text{value\_expr } \textbf{else true end} \simeq \textbf{true}}$$

Proof rule structure

$$\frac{\text{premise}_1 \ \ldots \ \text{premise}_n}{\text{conclusion}}$$

meaning the conclusion is true when all the premises are.

Premises are well-formedness conditions and applicability conditions.

Conclusions are commonly equivalences, but also include typing rules and refinement relations.

Proof rules for proof

- Aim is doing proof
- and doing so automatically as far as possible.
- Need derived rules as well as basic ones, where basic rules correspond to the axiomatic semantics rules.
- Tools can handle well-formedness and contexts: so make these implicit.

Original proof tool had about 300 basic rules, 1700+ derived ones.

Proof rules for proof: example

[if_annihilation1]
if eb **then** eb **else true end** $\simeq$ **true**
when convergent(eb) $\wedge$ **readonly**(eb)

Proof rule structure

- Contexts and well-formedness premises have gone
- Premises introduced by **when**
- Use of *special functions* built into prover (and often automatically dischargeable)
- Conventions for term variable names, e.g.
  - e: value expression
  - b: Bool type
  - i: Int type
  - s: set type
  - e, e′, e″ etc. have same type
  - e, e1, e2 etc. may have different types

Soundness

Which of these rules are sound?

[subset_difference]
es $\subseteq$ es$'$ $\setminus$ es$''$ $\simeq$ **true**
when convergent(es) $\wedge$ **readonly**(es) $\wedge$
convergent(es$'$) $\wedge$ **readonly**(es$'$) $\wedge$
convergent(es$''$) $\wedge$ **readonly**(es$''$) $\wedge$ es $\subseteq$ es$'$ $\wedge$ es $\cap$ es$''$ $= \{\}$

[proper_subset_difference]
es $\subset$ es$'$ $\setminus$ es$''$ $\simeq$ **true**
when convergent(es) $\wedge$ **readonly**(es) $\wedge$
convergent(es$'$) $\wedge$ **readonly**(es$'$) $\wedge$
convergent(es$''$) $\wedge$ **readonly**(es$''$) $\wedge$ es $\subset$ es$'$ $\wedge$ es $\cap$ es$''$ $= \{\}$

More soundness tests

[subset_expansion1]
es $\subseteq$ es$'$ $\simeq \sim$ (es $\supset$ es$'$)

[proper_subset_expansion1]
es $\subset$ es$'$ $\simeq \sim$ (es $\supseteq$ es$'$)

A problem

How do you find all the errors on 2000+ rules?

One answer

Use another theorem prover, assume faults are independent, and prove your proof rules.

Using the PVS translator, found 6 errors affecting 11 rules in 1000+.

Another problem

How do you know the built-in procedures in your proof tool are sound?

One answer

1. Use a proof tool that has built-in procedures and can output the proof in terms of basic proof rules.
2. Rerun the proof in another prover with no procedures and only basic proof rules.

This only helps with individual proofs: correspond to test cases for the proof tool.

But could be used on a critical project.

Imperative RSL

Chris George

United Nations University
International Institute for Software Technology
Macao SAR, China

Imperative Specification: Example

```
COUNTER =
  class
    variable
      counter : Nat := 0
    value
      increase : Unit → write counter Nat
      increase() ≡ counter := counter + 1 ; counter
  end
```


Imperative Expressions

No syntactic distinction between

- statements and
- expressions

Imperative expressions:

- assignments (id := value_expr)
- sequencing (*unit*-value_expr$_1$; value_expr$_2$)
- iterative expressions (while, until, for)
- if expressions
- ...

Meanings of expressions

In general expressions may have both

- effects and
- values

Effects are changes to variables and input or output on channels.

For expressions to be equivalent (≡) they must have the same potential effects as well as the same values.

For expressions to be equal (=) they must have the same values.

Evaluation Order

Evaluation order is critical when we may have effects.

Evaluation in RSL is left-to-right.

For example, suppose we have a **variable** x : **Int**:

$$\langle x := 1 ; x \ , x := 2 ; x \rangle \equiv x := 2 ; \langle 1,2 \rangle$$
$$\langle x := 2 ; x \ , x := 1 ; x \rangle \equiv x := 1 ; \langle 2,1 \rangle$$

$$x + (x := x + 1 ; x) \equiv x := x + 1 ; 2 * x - 1$$
$$(x := x + 1 ; x) + x \equiv x := x + 1 ; 2 * x$$

Equivalence versus Equality

$=$ and $\equiv$ differ in terms of

- undefinedness (**chaos**)
- non-determinism
- effects (variables and communication)

otherwise they are the same.

For example, we can say

factorial(3) = 6
or
factorial(3) $\equiv$ 6

They are both true.

When equivalence and equality differ

Assume the variable x currently holds the value 0.

Expression	Evaluation
$1 \parallel 2 = 1 \parallel 2$	**true** $\parallel$ **false**
$1 \parallel 2 \equiv 1 \parallel 2$	**true**
while true do skip end $=$ **chaos**	**chaos**
while true do skip end $\equiv$ **chaos**	**true**
$((x := x + 1 ; 1) = (x := x + 1 ; x))$	$x := 2$; **false**
$((x := x + 1 ; 1) \equiv (x := x + 1 ; x))$	**true**

Equivalence versus Equality

- $\equiv$ and $=$ are the same if
  the arguments are convergent and pure.
- $\equiv$ is always defined.
- $\equiv$ compares effects as well as results;
  $=$ only compares results
- $\equiv$ has hypothetical evaluation;
  $=$ has left-to-right evaluation.
- $\equiv$ gives no effects;
  $=$ may give effects.

Applicative to imperative transformation

- Remove definition of type of interest.
- Add variable(s) to model concrete type of interest.
- Remove type of interest from function signatures; fill holes with **Unit**.
- Give type "**Unit** $\to$ **write** ... **Unit**" to each constant "c" of type of interest, and replace "=" by "c() $\equiv$".
- Insert write accesses to generators.
- Insert read accesses to observers.
- Remove formal parameters representing type of interest.
- Replace occurrences of type of interest parameters with references to variable(s).

- For generators, insert assignments.

This is for "leaf" modules in a hierarchy. Non-leaf modules involves a similar but simpler transformation.

Imperative example

I_DATABASE =
  class
    type Key, Data
    variable database : Key $\xrightarrow[m]{}$ Data
    value
      empty : **Unit** $\to$ **write** database **Unit**
      empty() $\equiv$ database := [ ],

      insert : Key $\times$ Data $\to$ **write** database **Unit**
      insert(k,d) $\equiv$ database := database $\dagger$ [k $\mapsto$ d],

      remove : Key $\to$ **write** database **Unit**
      remove(k) $\equiv$ database := database $\setminus$ {k},

      defined : Key $\to$ **read** database **Bool**
      defined(k) $\equiv$ k $\in$ **dom** database,

      lookup : Key $\xrightarrow{\sim}$ **read** database Data
      lookup(k) $\equiv$ database(k)
      pre defined(k)
  end

While expressions

```
FRACTION_SUM = class
    variable
        counter : Nat,  result : Real
    value
        fraction_sum : Nat ~→ write counter, result Unit
        fraction_sum(n) ≡
            counter := n ;  result := 0.0 ;
            while counter > 0 do
                result := result + 1.0/(real counter) ;
                counter := counter − 1
            end
        pre n > 0
end
```


$1 + 1/2 + ... + 1/n$

Is the precondition of fraction_sum necessary?

Until Expressions

```
FRACTION_SUM = class
    variable
        counter : Nat,  result : Real
    value
        fraction_sum : Nat ~→ write counter, result Unit
        fraction_sum(n) ≡
            counter := n ;  result := 0.0 ;
            do
                result := result + 1.0/(real counter) ;
                counter := counter − 1
            until counter = 0 end
        pre n > 0
end
```


For Expressions

```
FRACTION_SUM =
  class
    variable
        result : Real
    value
        fraction_sum : Nat ~→ write result Unit
        fraction_sum(n) ≡
            result := 0.0 ;
            for i in ⟨1 .. n⟩ do
                result := result + 1.0/(real i)
            end
        pre n > 0
end
```

Concurrent RSL

Chris George

United Nations University
International Institute for Software Technology
Macao SAR, China

Concurrency

Concurrency is necessary in particular for describing distributed systems.

Concurrent systems in general may communicate through

- shared variables, or
- message passing

RSL uses message passing.

Message passing is more abstract: shared variables may be modelled using message passing.

Composition of Expressions

Composition:

- sequential:

  value_expr_1 ; value_expr_2

- concurrent:

  $\text{value\_expr}_1 \parallel \text{value\_expr}_2$

  1. has type **Unit**
  2. value_expr_1 and value_expr_2 must have type **Unit**
  3. value_expr_1 and value_expr_2 recommended to be assignment-disjoint

Communication Expressions

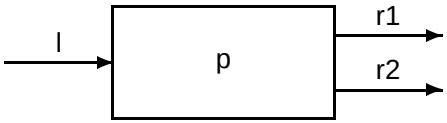
channel id : type_expr

Communication expressions:

- input expressions: id ?
- output expressions: id ! value_expr

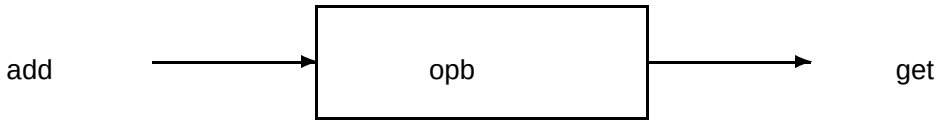
Input expressions have the same type as the channel.
Output expressions have type **Unit**.

Example

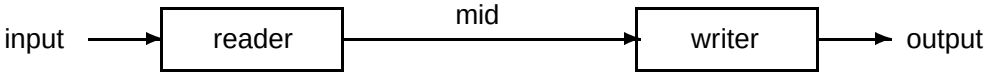



```
channel
    l, r1, r2: Int

value
    p : Unit → in l out r1, r2 Unit
    p() ≡
        let e = l? in (r1!e || r2!e) end; p()
```


Another example




```
ONE_PLACE_BUFFER =
    class
        type Elem
        channel add, get : Elem
        value
            opb : Unit → in add out get Unit
            opb() ≡ let v = add? in get!v end ; opb()
    end
```

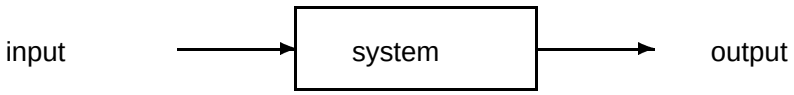

```
READER_WRITER =
    class
        type Elem
        channel input, output, mid : Elem
        value
            reader : Unit → in input out mid Unit
            reader() ≡
                let v = input? in mid ! v end ; reader(),
            writer : Unit → in mid out output Unit
            writer() ≡
                let v = mid? in output ! v end ; writer()
    end
```



```
SYSTEM =  extend READER_WRITER with
    class
        value
            system : Unit →
                        in input, mid out output, mid Unit
            system() ≡ reader() || writer()
    end
```

```
system() ≡
    let v = input? in mid ! v end ; reader()
    ||
    let v = mid? in output ! v end ; writer()
```

We should make the channel *mid* unavailable to any other processes.




```
SYSTEM = class
    type Elem
    channel input, output : Elem
    value
        system : Unit → in input out output Unit
        system() ≡
            local
                channel mid : Elem
                value
                    reader : Unit → in input out mid Unit
                    reader() ≡ let v = input? in mid ! v end ; reader(),
                    writer : Unit → in mid out output Unit
                    writer() ≡ let v = mid? in output ! v end ; writer()
            in reader() || writer() end
end
```


External choice

The value expression

```
v:=c1? [] c2!e
```

will:

- input from *c1* if a value expression is willing to output to *c1* but no value expression is willing to input from *c2*;
- output to *c2* if a value expression is willing to input from *c2* but no value expression is willing to output to c_1;
- either input from *c1* or output to *c2* if a value expression is willing to output to *c1* and a value expression is willing to input from *c2*;
- deadlock if no value expression is ever willing to output to *c1* and no value expression is ever willing to input from *c2*.

Internal choice

The value expression

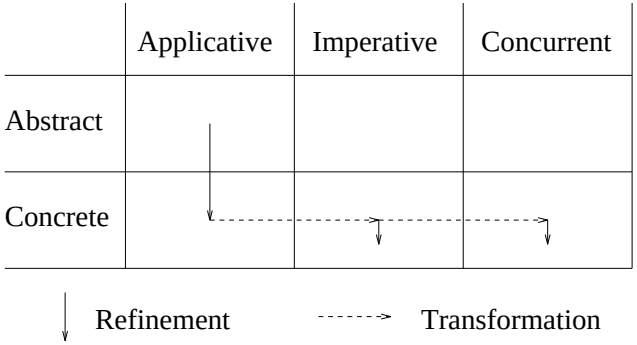
```
v:=c1? ⌈] c2!e
```

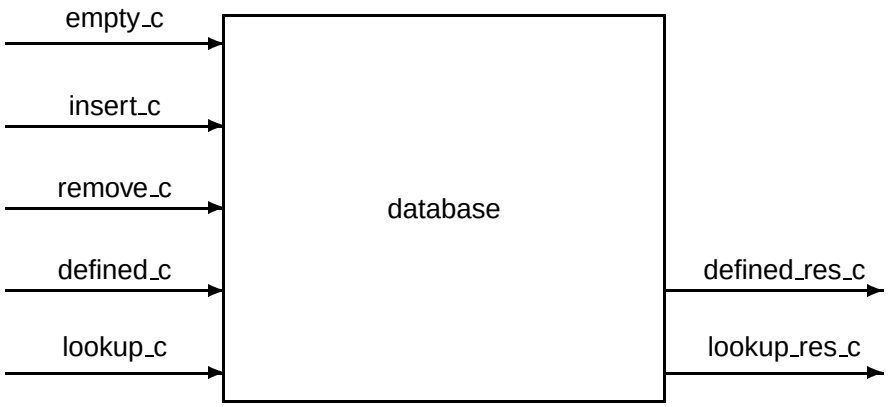
will:

- either deadlock or input from *c1* if a value expression is willing to output to *c1* but no value expression is willing to input from *c2*;
- either deadlock or output to *c2* if a value expression is willing to input from *c2* but no value expression is willing to output to *c1*;
- either input from *c1* or output to *c2* if a value expression is willing to output to *c1* and a value expression is willing to input from *c2*;
- deadlock if no value expression is ever willing to output to *c1* and no value expression is ever willing to input from *c2*.

```
channel
    empty : Unit, add, get : Elem
value
    mpb : Unit → in empty, add out get Unit
    mpb() ≡
        local
            variable buffer : Elem* := ⟨⟩
        in
            while true do
                empty? ; buffer := ⟨⟩
                []
                let v = add? in buffer := buffer ⌢ ⟨v⟩ end
                []
                if buffer ≠ ⟨⟩ then get ! hd buffer ; buffer := tl buffer
                else stop end
            end
        end
```


Typical Development





Imperative to concurrent transformation

- Insert an object instantiating the imperative sequential module, and hide it.
- Define channels for each observer and generator; at least one channel for each. Hide them.
- Define a “server” process:
  - type “**Unit → in** ... **out** ... **write** I.**any Unit**”
  - body is a **while true do** loop
  - loop body is an external choice between clauses, one clause for each observer and each generator
  - each clause inputs parameters (if any); calls corresponding function I.f; outputs result (if any). Must do at least one communication.

  Hide it.

- Define an “init” process with the same type as the server that initialises the imperative object and calls the server.
- Define “interface functions” mirroring clauses in server. These *have no accesses to the imperative object*.

This is for “leaf” modules in a hierarchy. Non-loeaf modules are similar but easier.


```
C_DATABASE = hide I, database in
  class
    object I : I_DATABASE
    type
      Key = I.Key,
      Data = I.Data,
      Result == not_found | res(Data)
    channel
      empty_c : Unit,
      insert_c : Key × Data,
      remove_c, defined_c, lookup_c : Key,
      defined_res_c : Bool,
      lookup_res_c : Result
```

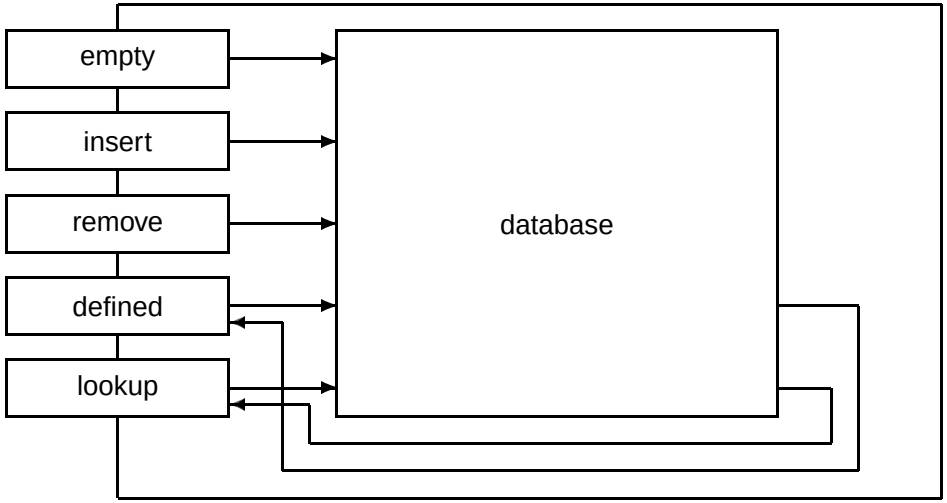


```
value
  init : Unit → in empty_c, insert_c, remove_c, defined_c, lookup_c
                 out defined_res_c, lookup_res_c write I.any Unit
  init() ≡ I.empty() ; database(),
```



```
database : Unit → in ... out ... write I.any Unit
database() ≡
  while true do
    empty_c? ; I.empty()
    []
    let (k,d) = insert_c? in I.insert(k,d) end
    []
    let k = remove_c? in I.remove(k) end
    []
    let k = defined_c? in defined_res_c ! I.defined(k)
    end
    []
    let k = lookup_c? in
      if I.defined(k) then lookup_res_c ! res(I.lookup(k))
      else lookup_res_c ! not_found end end end end
```

Encapsulation with Interface Functions




```
INTERFACED_DATABASE =
  hide empty_c, insert_c, remove_c, defined_c, lookup_c,
       defined_res_c, lookup_res_c in
  extend C_DATABASE with
  class
    value
      empty : Unit → out any Unit
      empty() ≡ empty_c ! (),

      insert : Key × Data → out any Unit
      insert(k,d) ≡ insert_c ! (k,d),

      remove : Key → out any Unit
      remove(k) ≡ remove_c ! k,
```



```
      defined : Key → in any out any Bool
      defined(k) ≡ defined_c ! k ; defined_res_c?,

      lookup : Key → in any out any Result
      lookup(k) ≡ lookup_c ! k ; lookup_res_c?
  end
```

Modularity in RSL

Chris George

United Nations University
International Institute for Software Technology
Macao SAR, China

Specifications

An RSL specification consists of

- module definitions

A module contains definitions of

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

Modularity

Modules are the building blocks.

Purposes:

- Readability
- Separate development
- Reuse

Schemes and Objects

Modules are either schemes or objects.

A scheme denotes a class of models

scheme id = class_expr

An object denotes a single model

object id : class_expr

Class Expressions

- basic
- with
- extending
- renaming
- hiding
- instantiation

With class expression

General form:

with *element*-object_expr-*list* **in** class_expr

with X **in** class_expr

means that a name *n* in *class_expr* can mean either *n* or *X.n*. This means, providing there is no confusion, that qualifications like *X.* can be omitted.

Extension

General form:

extend class_expr$_1$ **with** class_expr$_2$

appends the second class to the first.

class_expr$_1$ and class_expr$_2$ must be compatible

Renaming

General form:

use
id$_{new_1}$ **for** id$_{old_1}$, ... ,id$_{new_n}$ **for** id$_{old_n}$
in class_expr

For example

scheme BUFFER $=$
use
add **for** enq, get **for** deq, Buffer **for** Queue
in QUEUE

Hiding

General form:

hide id$_1$,...,id$_n$ **in** class_expr

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 auxiliary functions

Objects

```
scheme BUFFER =
  class
    variable buff : Int*
    value
      is_empty : Unit → read buff Bool
      ...
  end

object
  B1 : BUFFER,
  B2 : BUFFER
```

B1 and B2 are distinct, global objects and we can use them ...

Using objects

```
scheme SYS =
  class
    value
      one_is_empty : Unit → read B1.buff B2.buff Bool
      one_is_empty() ≡ B1.is_empty() ∨ B2.is_empty()
  end
```

B1.buff and B2.buff are distinct

Module Nesting

```
scheme
  SYS =
    class
      object
        B1 : BUFFER,
        B2 : BUFFER
      value
        one_is_empty : Unit → read B1.buff B2.buff Bool
        one_is_empty() ≡ B1.is_empty() ∨ B2.is_empty()
    end
```

B1 and B2 are distinct, embedded objects.

Building hierarchies

Suppose we have a system that needs a database component.

There are several ways we can construct the specification:

- merging the system and database definitions in one class
- extending the database class with the system class
- making a hierarchy with a database object

Merging the definitions in one class

```
scheme SYSTEM =
  class
    /* database */
    ⋮
    /* system */
    ⋮
  end
```

- Hard to read
- Database cannot be reused
- Hard to make database private to system
- Problem of name clashes between two parts

Extending the database

```
scheme DATABASE = ...

scheme SYSTEM =
  extend DATABASE with ...
```

- Easier to read
- Database can be reused
- Hard to make database private to system
- Problem of name clashes between two parts

Making a hierarchy with a database object

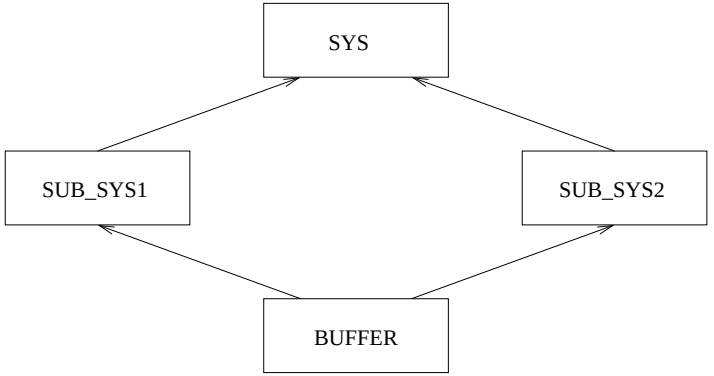
```
scheme DATABASE = ...

scheme SYSTEM =
  class
    object DB : DATABASE
    ⋮
  end
```

- Easier to read
- Database can be reused
- Easy to make database private to system:

```
scheme SYSTEM =
    hide DB in
    class
        object DB : DATABASE
        ⋮
    end
```

- No problem of name clashes between two parts

Sharing




```
scheme BUFFER = ...

scheme SUB_SYS1 = class object B : BUFFER ... end
scheme SUB_SYS2 = class object B : BUFFER ... end

scheme SYS =
   class
      object
         O1 : SUB_SYS1,
         O2 : SUB_SYS2
      ⋮
   end
```

We get two buffer variables (O1.B.buff and O2.B.buff)

Sharing using global objects

```
object B : BUFFER

scheme
   SUB_SYS1 = class ... B.buff ... end,
   SUB_SYS2 = class ... B.buff ... end,

   SYS = class
            object
               O1 : SUB_SYS1,
               O2 : SUB_SYS2
         end
```

We get only one buffer: B.buff

Sharing using parameterization

```
scheme BUFFER = ...
scheme SUB_SYS1(B: BUFFER) = ...
scheme SUB_SYS2(B: BUFFER) = ...

scheme SYS =
  class
    object
      B : BUFFER,
      O1 : SUB_SYS1(B),
      O2 : SUB_SYS2(B)
    ⋮
  end
```


Parameterization - Example

```
scheme BUFFER =
  class
    type Elem
    variable buff : Elem*
    value
      empty : Unit → write buff Unit
      empty() ≡ buff := ⟨⟩,

      add : Elem → write buff Unit
      add(e) ≡ buff := buff ⌢ ⟨e⟩
  end
```

is better expressed using parameterization:


```
scheme ELEM = class type Elem end

scheme BUFFER(E : ELEM) =
  class
    variable buff : E.Elem*
    value
      empty : Unit → write buff Unit
      empty() ≡ buff := ⟨⟩,

      add : E.Elem → write buff Unit
      add(e) ≡ buff := buff ⌢ ⟨e⟩
  end
```


Instantiation - Example

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

  INTEGER_BUFFER :  BUFFER(INTEGER)
```

If we expand BUFFER(INTEGER):

```
INTEGER_BUFFER :
  class
    variable buff : INTEGER.Elem*
    value
      empty : Unit → write buff Unit
      empty() ≡ buff := ⟨⟩,

      add : INTEGER.Elem → write buff Unit
      add(e) ≡ buff := buff ⌢ ⟨e⟩
  end
```


Actual versus Formal Parameters

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

... S(A) ...
```

Context condition: AC must statically implement FC

RAISE Method

Chris George

United Nations University
International Institute for Software Technology
Macao SAR, China

Small print

- *The licensed material is provided "as is" without warranty of any kind.*
- *The Vendor disclaims ... conformance between the software and ... manuals*
- *The entire risk ... is with the Licensee*
- *... in no event will the Vendor be liable for any damages ...*
- *The Licensee shall ... hold harmless the Vendor against all claims ...*

Claims of competence, perhaps?

Software Crisis

- for every six new large-scale software systems that are put in operation, two others are cancelled.
- the average software development project overshoots its schedule by half
- some three quarters of all large systems do not function as intended or are not used at all.

"Software Hell – Bugs. Viruses. Complexity.
Is there any way out of this mess?" (Business Week, 1999)

Denver Airport Baggage-Handling, 1994

- Twice the size of Manhattan, 10 times the breadth of Heathrow, three jets can land simultaneously in bad weather.
- The subterranean baggage-handling system consists of 34 km of track with 4000 independent "telecars" routing and delivering luggage between counters, gates and claim areas. It is controlled by a network of 100 computers with 5000 sensors, 400 radio receivers and 56 bar-code scanners.
- Despite his woes, the contractor says the project's worth it: "Who would turn down a USD 193 million contract? You'd expect to have a little trouble for that kind of money." (New York Times, 18 Mar 1994)

Denver Airport Baggage-Handling, 1994 (cont.)

- Software did not work!
- Too little time for system testing.
- The delay of the opening of the airport was 9 months.
- They decided to build another baggage handling system — the conventional kind with conveyor belts — for another USD 50 million.

Ariane 5, 1996

- On 4 June 1996 Ariane 5 rocket exploded,
- Caused by software in the inertial guidance system.
- An inertial platform from the Ariane 4 was used aboard the Ariane 5 without proper testing.
- When subjected to the higher accelerations produced by the Ariane 5 booster, the software (calibrated for an Ariane 4) ordered an ”abrupt turn 30 seconds after liftoff”.
- A precondition of the software was violated.

What is the acceleration due to gravity?

Mars Climate Orbiter, 1999

- Sept. 1999 Mars Climate Orbiter disappeared after successfully travelling 416 million miles in 41 weeks.
- Lockheed Martin Astronautics used acceleration data in Imperial units (feet per second per second).
- Jet Propulsion Laboratory (JPL) did its calculations with metric units (metres per second per second).
- Integration testing should have been revealed this fault!
- NASA started a $50,000 project to discover how this could have happened.

The stories continue

Peter Neumann's Risk Forum

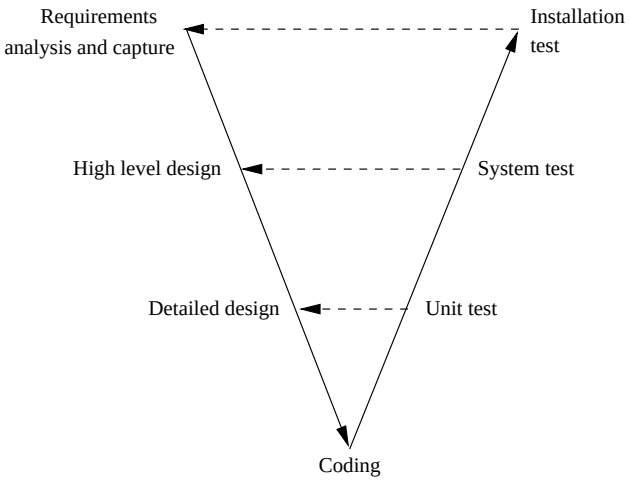
http://catless.ncl.ac.uk/Risks/

Edsger W. Dijkstra

In academia, in industry, and in the commercial world, there is a widespread belief that computing science as such has been all but completed and that, consequently, computing has matured from a theoretical topic for the scientists to a practical issue for the engineers, the managers, and the entrepreneurs. [...]

I would therefore like to posit that computing's central challenge, "How not to make a mess of it," has not been met. On the contrary, most of our systems are much more complicated than can be considered healthy, and are too messy and chaotic to be used in comfort and confidence. The average customer of the computing industry has been served so poorly that he expects his system to crash all the time, and we witness a massive worldwide distribution of bug-ridden software for which we should be deeply ashamed. (Communications of the ACM, Mar 2001)

V-diagram model of software life cycle



The V-diagram illustrates the typical re-work cycles when we discover errors by testing.

We aim to *find errors earlier*.

We concentrate on the early stages:

- *requirements analysis and capture*
- *high level design*

Why formal methods?

To produce software that is

- more likely to be correct
- more reliable
- better documented
- more easily maintainable

What is formality?

- a language — symbols and grammar rules for constructing terms
- (usually) rules for deciding if terms are well formed (e.g. scope, typing rules)
- a semantics — a description of what terms mean
- a logic — a set of rules for determining if predicates about terms are true

Programming languages are not formal according to this definition because they lack a logic.

Characteristics of formal methods

- Precise notation
- Abstraction (*what* rather than *how*)
- Stepwise development (gradual commitment)
- Proof opportunities and justifications
- Structuring based on compositionality
- Guidelines for quality assurance

Rigorous methods

Choice of level of formality. E.g.

1. No proof opportunities generated or checked
2. Proof opportunities generated and inspected but not proved
3. Proof opportunities generated and proved with some informal steps — “it follows immediately that ...”
4. Proof opportunities generated and proved formally

All formal methods are in fact rigorous. But only a method with a formal basis can be rigorous, because it must always be possible to say “I am not sure if it does follow. Please prove it.”

Current state of the art is the first three levels.

Implementation relation

- new signature includes the old one (statically decidable)
- old properties preserved by the new one ($\Rightarrow$ implementation conditions)

Example 1

scheme S0 =
  class
    value x : **Int**
    axiom x $\geq$ 0
  end

scheme S1 =
  class
    value
      x : **Int** = 2
  end

scheme S2 =
  class
    value
      x : **Int** = 2
      y : **Int** = 0
  end

Does S1 or S2 implement S0?

Does S2 implement S1?

Example 2

scheme S0 =
  hide z **in class**
    value x, y, z : **Int**
    axiom x $>$ z $\wedge$ z $>$ y
  end

scheme S1 =
  class
    value
      x : **Int** = 1
      y : **Int** = 0
  end

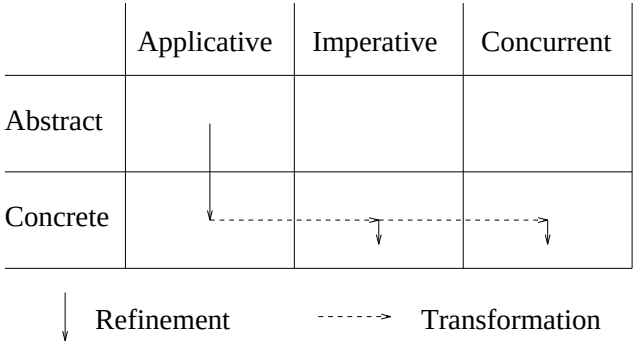
scheme S2 =
  class
    value
      x : **Int** = 2
      y : **Int** = 0
  end

Does S1 or S2 implement S0?

Design

- removing underspecification
  - abstract types to concrete types
  - more explicit value definitions
- changing style
  - applicative/imperative
  - sequential/concurrent
- providing more efficient algorithms

Typical Development



Translation

- manual translation
- automatic translation (to SML and C++)

of low-level RSL (e.g. concrete types and explicit value definitions)

An example RAISE development

Chris George

United Nations University
International Institute for Software Technology
Macao SAR, China

An example

A message system with possible overtaking:

- Messages can be inserted and extracted.
- There may be some delay between a message being inserted and it being available for extraction.
- The extraction order should be the same as the insertion order, except that there should be some possibility of higher priority messages "overtaking" lower priority ones.
- It is not necessary to guarantee that the next message extracted is the highest priority one in the system. This is ideal, but may not always be possible.

TYPES

```
scheme
  TYPES =
    class
      type Message

      value
        priority : Message → Nat,

        leq : Message × Message → Bool
        leq(m1, m2) ≡ priority(m1) ≤ priority(m2)
    end
```


A_MESSAGE0

```
scheme A_MESSAGE0 =
  hide buffered, permutation, count in
  class
    type Buffer
    value
      put : T.Message × Buffer → Buffer,
      get : Buffer ~→ T.Message × Buffer,
      can_get : Buffer → Bool,
      buffered : Buffer × T.Message* → Bool
    axiom
      [ can_get_ax ]
      ∀ buff : Buffer • buffered(buff, ⟨⟩) ⇒ ~ can_get(buff),
```

[buffered_put]
∀ buff, buff′ : Buffer, l : T.Message*, m : T.Message •
buffered(buff, l) ∧ put(m, buff) = buff′ ⇒ buffered(buff′, l ⌢ ⟨m⟩),

[buffered_get]
∀ buff, buff′ : Buffer, l : T.Message*, m1, m2 : T.Message •
buffered(buff, l) ∧ can_get(buff) ∧ get(buff) = (m1, buff′) ⇒
(∃ l1, l2 : T.Message* •
l = l1 ⌢ ⟨m1⟩ ⌢ l2 ∧ buffered(buff′, l1 ⌢ l2) ∧
(m2 ∈ **elems** l1 ⇒ ∼ T.leq(m1, m2))),

[no_loss_or_gain]
∀ buff : Buffer, l1, l2 : T.Message* •
buffered(buff, l1) ∧ buffered(buff, l2) ⇒ permutation(l1, l2)

end

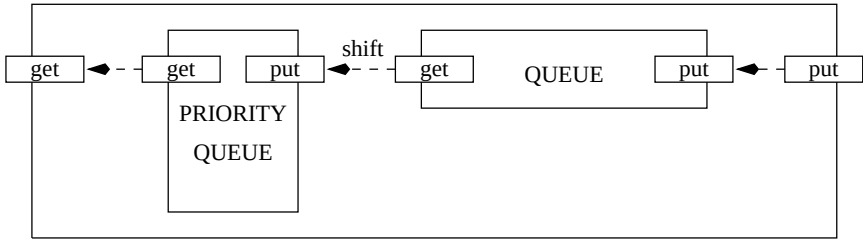
buffered is a relation between the abstract state *Buffer* and the list of messages input but not yet extracted.

This is naturally a relation (rather than a function) because it is many-many. Suppose *m1* and *m2* are messages, with *m2* higher priority.

Output *m2* followed by *m1* has two possible inputs.

Input *m1* followed by *m2* has two possible outputs.

Design idea



Parameter classes

scheme ELEM = **class type** Elem **end**

```
scheme A_QUEUE(E : ELEM) =
  class
    type Queue = E.Elem*
    value
      empty : Queue = ⟨⟩,

      put : E.Elem × Queue → Queue
      put(e, s) ≡ s ⌢ ⟨e⟩,

      get : Queue ~→ E.Elem × Queue
      get(s) ≡ (hd s, tl s) pre ∼ is_empty(s),

      is_empty : Queue → Bool
      is_empty(s) ≡ s = ⟨⟩
  end
```



```
scheme
  PARTIAL_ORDER(E : ELEM) =
    class
      value
        leq : E.Elem × E.Elem → Bool

      axiom
        [ reflexive ] ∀ a : E.Elem • leq(a, a),

        [ transitive ]
          ∀ a, b, c : E.Elem • leq(a, b) ∧ leq(b, c) ⇒ leq(a, c)
    end
```



```
scheme
  TOTAL_ORDER(E : ELEM) =
    extend PARTIAL_ORDER(E) with
      class
        axiom
          [ linear ] ∀ a, b : E.Elem • leq(a, b) ∨ leq(b, a)
      end
```



```
scheme A_PRI_QUEUE(E : ELEM, T : TOTAL_ORDER(E)) =
  hide is_ordered in
  class
    type Pri_queue = {| l : E.Elem* • is_ordered(l) |}

    value
      is_ordered : E.Elem* → Bool
      is_ordered(l) ≡
        (∀ i, j : Nat • {i, j} ⊆ inds l ∧ i < j ⇒ T.leq(l(j), l(i))),
```

```
    empty : Pri_queue = ⟨⟩,

    put : E.Elem × Pri_queue → Pri_queue
    put(e, s) ≡
      case s of
        ⟨⟩ → ⟨e⟩,
        ⟨h⟩ ⌢ t → if T.leq(e, h) then ⟨h⟩ ⌢ put(e, t) else ⟨e, h⟩ ⌢ t end
      end,

    get : Pri_queue ~→ E.Elem × Pri_queue
    get(s) ≡ (hd s, tl s) pre ∼ is_empty(s),

    is_empty : Pri_queue → Bool
    is_empty(s) ≡ s = ⟨⟩
end
```


Confidence conditions

RSL	Confidence condition
value empty : Pri_queue = ⟨⟩	is_ordered(⟨⟩)
hd s	s ≠ ⟨⟩
PQ.get(s)	PQ.is_ordered(s) ∧ ∼PQ.is_empty(s)

Concrete applicative message system

```
scheme A_MESSAGE1 =
  hide PQ, Q in
  class
    object
      PQ : A_PRI_QUEUE(T{Message for Elem}, T),
      Q : A_QUEUE(T{Message for Elem})

    type Buffer = PQ.Pri_queue × Q.Queue

    value
      put : T.Message × Buffer → Buffer
      put(m, (pq, q)) ≡ (pq, Q.put(m, q)),
```



```
      get : Buffer ~→ T.Message × Buffer
      get(pq, q) ≡
        let (e, pq′) = PQ.get(pq) in (e, (pq′, q)) end
        pre can_get(pq, q),

      can_get : Buffer → Bool
      can_get(pq, q) ≡ ∼ PQ.is_empty(pq),

      shift : Nat × Buffer → Buffer
      shift(n, (pq, q)) ≡
        if n = 0 ∨ Q.is_empty(q) then (pq, q)
        else
          let (m, q′) = Q.get(q), pq′ = PQ.put(m, pq) in
            shift(n − 1, (pq′, q′))
          end end
  end
```

Implementation relation

Class B *implements* a class A (written B $\preceq$ A) if and only if

1. the signature of B includes the signature of A
2. all the *properties* of A hold in B

The signature check is static and done by tools.

Properties of a class

These arise from

- axioms
- value definitions
- subtype conditions on values, variables and channels
- initialisations of variables
- properties of objects defined in the class

Showing A_MESSAGE1 $\preceq$ A_MESSAGE0

Extend A_MESSAGE1 with a definition of *buffered*:


```
value
  buffered : Buffer × T.Message* → Bool
  buffered((pq, q), l) ≡
    (∃ l1 : T.Message* • l = l1 ⌢ q ∧ pq = sort(l1)),

  sort : T.Message* → T.Message*
  sort(l) ≡
    if l = ⟨⟩ then ⟨⟩
    else
      let i = first(l) in
        ⟨l(i)⟩ ⌢ sort(sublist(l, 1, i − 1) ⌢ sublist(l, i + 1, len l))
      end
    end
```

```
first : T.Message* ~→ Nat
first(l) as i post
    i ∈ inds l ∧
    (∀ j : Nat • j ∈ {1 .. i − 1} ⇒ ∼ T.leq(l(i), l(j))) ∧
    (∀ j : Nat • j ∈ {i + 1 .. len l} ⇒ T.leq(l(j), l(i)))
pre l ≠ ⟨⟩,

sublist : T.Message* × Nat × Nat → T.Message*
sublist(l, i, j) as l1 post
    if i < 1 ∨ j > len l ∨ i > j then l1 = ⟨⟩
    else
        len l1 = j − i + 1 ∧
        (∀ k : Nat • k ∈ inds l1 ⇒ l1(k) = l(k + i − 1))
    end
```

and copy definitions of *permutation* and *count* to the extension.

Can then generate 4 axioms of A_MESSAGE0 as properties to prove of extended A_MESSAGE1 (with definitions of *sort*, *first*, *sublist*, *permutation* and *count*).

NB: Extension must be *conservative*.

Theories

RAISE allows *theories* to be stated and used in justifications.

```
[ permutation_transitive ]
in A_MESSAGE1_EXT ⊢
    ∀ l1, l2, l3 : T.Message* •
        permutation(l1, l2) ∧ permutation(l2, l3) ⇒ permutation(l1, l3),

[ count_concatenation ]
in A_MESSAGE1_EXT ⊢
    ∀ m : T.Message, l1, l2 : T.Message* •
        count(m, l1^l2) = count(m, l1) + count(m, l2)
```


"Internal" functions

We have added *shift* to the interface, but we expect it to become internal. What properties should it have?

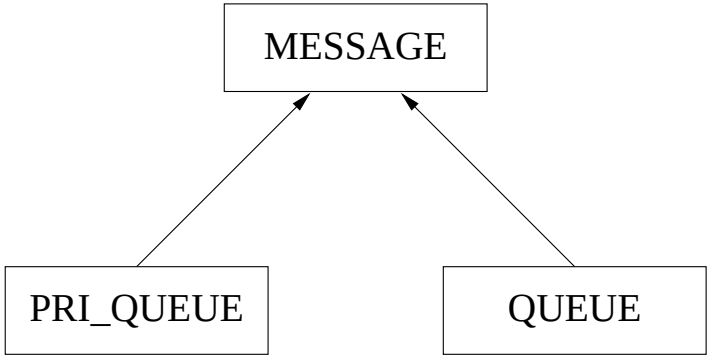
shift is not "invisible": can change *can_get*, for example.

Use the relation buffered:

```
∀ buff, buff′ : Buffer, n : Nat •
    buff′ = shift(n, buff) ⇒
    (∃ l : T.Message* • buffered(buff, l) ∧ buffered(buff′, l))
```

No loss or gain of messages.

Module dependencies



Applicative to imperative

- when all types of interest are concrete
- module by module: preserves structure
- “leaf” modules will have imperative state
- transformation: correct *by construction*

Imperative sequential queue

```
scheme I_QUEUE(E : ELEM) =
  hide v, Q in class
    object Q : A_QUEUE(E)

    variable v : Q.Queue := Q.empty

    value
      empty : Unit → write any Unit
      empty() ≡ v := Q.empty,

      put : E.Elem → write any Unit
      put(e) ≡ v := Q.put(e, v),
```



```
      get : Unit ~→ write any E.Elem
      get() ≡
        let (e, v′) = Q.get(v) in v := v′ ; e end
        pre ∼ is_empty(),

      is_empty : Unit → read any Bool
      is_empty() ≡ Q.is_empty(v)
end
```

Imperative sequential system

```
scheme I_MESSAGE1 =
  hide IPQ, IQ in class
    object
      IPQ : I_PRI_QUEUE(T{Message for Elem}, T),
      IQ : I_QUEUE(T{Message for Elem})
    value
      put : T.Message → write any Unit
      put(m) ≡ IQ.put(m),

      get : Unit ~→ write any T.Message
      get() ≡ IPQ.get() pre can_get(),
      can_get : Unit → read any Bool
      can_get() ≡ ∼ IPQ.is_empty(),

      shift : Nat → write any Unit
      shift(n) ≡
        if n = 0 ∨ IQ.is_empty() then skip
        else
          let m = IQ.get() in
            IPQ.put(m) ; shift(n − 1) end end
  end
```


Imperative to concurrent

- allows concurrent function calls without interference
- module by module: preserves structure
- “leaf” modules will have imperative state and “server” processes
- transformation: correct *by construction*

Partial functions

We need to make interface functions total. *get* needs to return either a message or a result “no messages”.

```
scheme
  ELEM_RES =
    extend ELEM with
      class type Result == nil | result(elem : Elem) end
```

Concurrent queue

```
scheme C_QUEUE(E : ELEM_RES) =
  hide I, CH in class
    object
      I : I_QUEUE(E),
      CH :
        class
          channel
            empty : Unit,
            put : E.Elem,
            get : E.Result,
            is_empty : Bool
        end
```


The server process

```
init : Unit → write any in any out any Unit
init() ≡ I.empty() ; main(),

main : Unit → write any in any out any Unit
main() ≡
  while true do
    CH.empty? ; I.empty()
    []
    CH.get ! if ∼ I.is_empty() then E.result(I.get()) else E.nil end
    []
    let e = CH.put? in I.put(e) end
    []
    CH.is_empty ! I.is_empty()
  end,
```


The interface processes

```
    empty : Unit → out any Unit
    empty() ≡ CH.empty ! (),

    get : Unit → in any E.Result
    get() ≡ CH.get?,

    put : E.Elem → out any Unit
    put(e) ≡ CH.put ! e,

    is_empty : Unit → in any Bool
    is_empty() ≡ CH.is_empty?
  end
```


Concurrent system

```
scheme C_MESSAGE1 =
  hide PQ, Q, shift in class
    object
      PQ : C_PRI_QUEUE(T1{Message for Elem}, T1),
      Q : C_QUEUE(T1{Message for Elem})

    value
      init : Unit → write any in any out any Unit
      init() ≡ PQ.init() || Q.init() || shift(),
```

```
    put : T1.Message → in any out any Unit
    put(m) ≡ Q.put(m),

    get : Unit → in any out any T1.Result
    get() ≡ PQ.get(),

    can_get : Unit → in any out any Bool
    can_get() ≡ ∼ PQ.is_empty(),

    shift : Unit → in any out any Unit
    shift() ≡
      while true do
        case Q.get() of T1.nil → skip, T1.result(m) → PQ.put(m) end
      end
end
```


Consequences of design paradigm

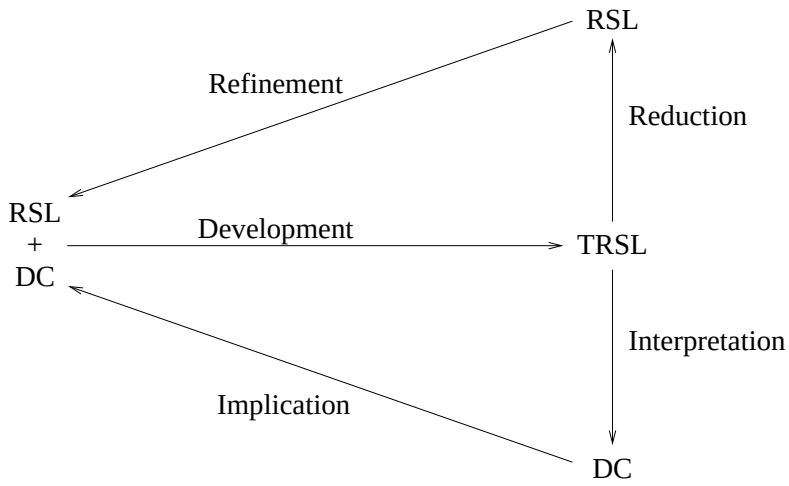
- States of leaf modules are independent: no interference.
- Interface functions of different leaf modules may be called sequentially or concurrently.
- Freedom from deadlock easy to check:
  - all channels hidden
  - all servers started by top-level *init*
  - interface functions and servers match.
- Emphasis on system design: some modules will be assumptions about the software and/or hardware environment.

Adding time

Timed RSL essentially just the addition of a **wait** expression.

```
value
  δ : Time • δ > 0.0,

  shift : Unit → in any out any Unit
  shift() ≡
    while true do
      wait δ ;
      case Q.get() of T1.nil → skip, E.result(m) → PQ.put(m) end
    end
```


Timed development



Timer variables

- may be started (set to zero) or reset (set negative)
- if not negative, are always incremented by **wait**s
- measure durations

An alarm

The alarm system may be enabled or disabled. When enabled, if it is disturbed, an alarm sounds. When disabled, disturbances are ignored.

The timing requirements are that after being enabled there is a period T1 before a disturbance causes an alarm. When it is enabled and there is a disturbance, there is a period T2 before the alarm sounds; if the system is disabled within this time there is no alarm.

Untimed server

```
while true do
    enable? ; I.enable()
    []
    disable? ; I.disable()
    []
    disturb? ;
    if I.state() = enabled then (I.disturb() ⌈ skip) end
end
```


Timed server

```
while true do
    enable? ; I.enable() ; since_disturb := reset ; since_enable := 0.0
    []
    disable? ; I.disable() ; since_disturb := reset ; since_enable := reset
    []
    disturb? ;
    if I.state() = enabled ∧ since_enable ≥ T1 ∧ since_disturb ≤ 0.0
    then since_disturb := 0.0 end
    []
    delay() ;
    if since_disturb ≥ T2 then I.disturb() end
end
```

```
delay : Unit → write any Unit
delay() ≡
    wait δ ;
    if since_enable ≥ 0.0
    then since_enable := since_enable + δ
    end ;
    if since_disturb ≥ 0.0
    then since_disturb := since_disturb + δ
    end
```


Conclusions

- wide spectrum, modular language
- effective method
- good documentation
- robust tools

Developing a National Financial Information System

Trần Mai Liên, Lê Linh Chi, Nguyễn Lê Thu,
Đỗ Tiến Dũng, Phùng Phương Nam, Hoàng Xuân Huấn,
and Chris George

National Financial Information System

- Taxation
- Treasury
- Budget
- Spending ministries
- External loans and aid

- Large project
- Extensive introduction of computers
- Previous development uneven
- Customer has limited technical knowledge and capacity
- Requirements changing:
  - New kinds of tax
  - New accounting rules

Prime candidate for failure.

The gap

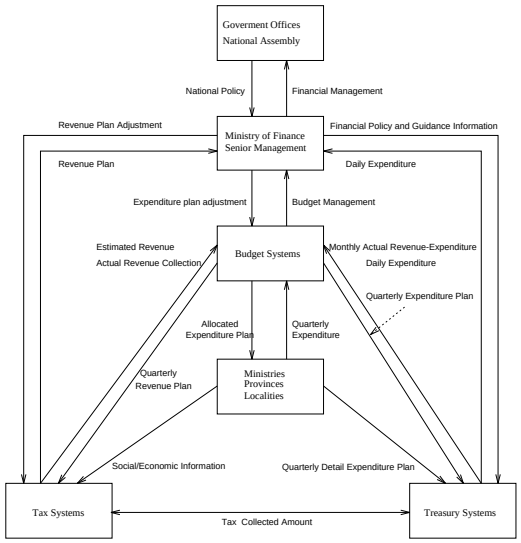
1. We need a national financial information system to collect reliable data, make budgets, assess the affects of possible changes, etc.
2. A small local office will need 8 PCs, ...

Aim of the specification

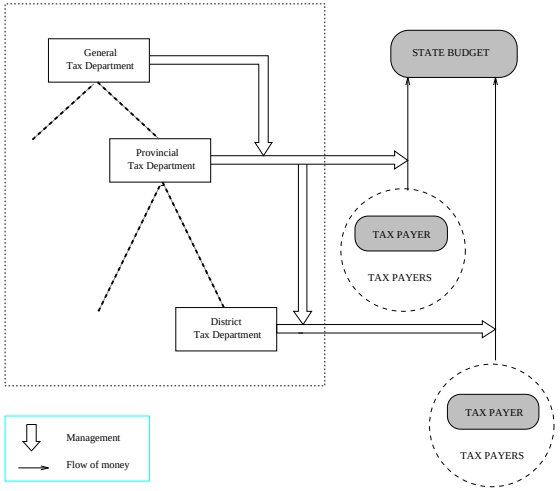
- Act as a high level design: allow design decisions to be explored.
- Define responsibilities for data storage, collection, analysis, reporting etc.
- and so provide detailed requirements for component offices.

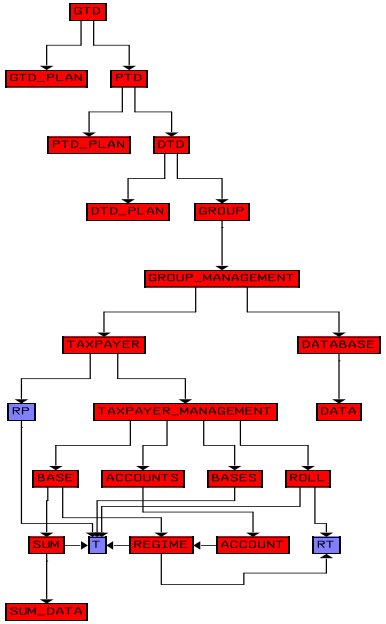
Not intended to be directly implemented! But provides a framework for gradual computerisation.

MoF data flow diagram



Tax system





Order of development

- Tax accounting
  - Rapid prototype (via translation) and testing
- Report generation and summarising
- Tax system
- Budget and treasury systems
- External loans and aid systems

Separate, parallel study on future of tax system.

Specification styles

- Applicative
  - convenient for specification
  - convenient for proofs
- Imperative sequential
  - convenient for implementation
  - twice as hard for proofs
- Imperative concurrent
  - convenient for implementation
  - 5 – 10 times as hard for proofs

Abstract axiom styles

- Applicative

  is_empty(empty) $\equiv$ **true**

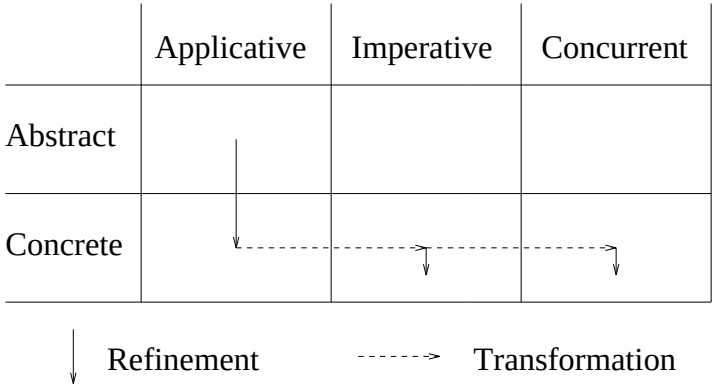
- Imperative sequential

  empty() ; is_empty() $\equiv$ empty() ; **true**

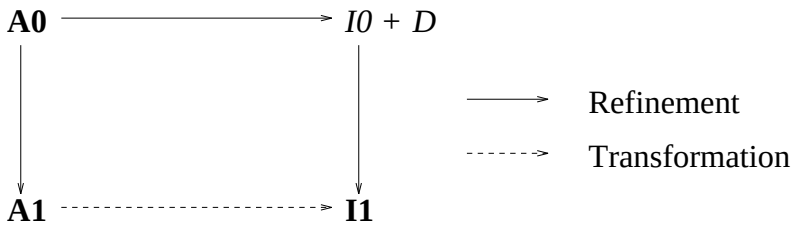
- Imperative concurrent

  $\forall$ test : **Bool** $\xrightarrow{\sim}$ **Unit** •
  (main() $\parallel$ empty()) $\parallel$ test(is_empty()) $\equiv$
  (main() $\parallel$ empty()) $\parallel$ test(**true**)

Ideal development route



Transformation theorem



Theorem:
If **A1** $\preceq$ **A0** and **A1** is transformed to **I1** then
$\exists$ module *I0*, definitions *D* •

- **I1** $\preceq$ *I0*
- *D* conservatively extends *I0*
- **extend** *I0* **with** *D* $\preceq$ **A0**

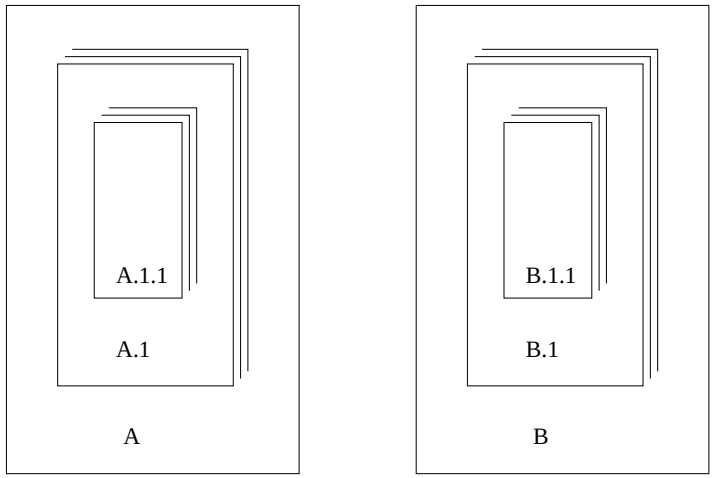
Transformation properties

- could be automated (?); amenable to quality assurance
- applied compositionally on modules

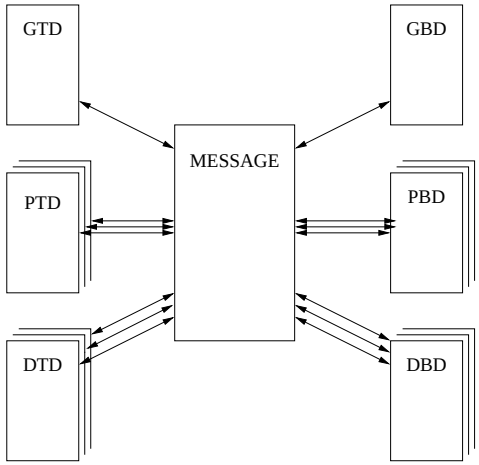
Consequences

- deadlock freedom guaranteed
- for trees:
  - imperative state only in leaf modules; their states are independent
  - functions of lower level modules may be called sequentially or concurrently
- for acyclic graphs: calling sequences need more care

Separate hierarchical subsystems



Distributed and combined subsystems



Hierarchical to distributed

Should be a *transformation*: reliable and repeatable.

- Preserve properties already checked
- Use standard modules
- Restrict editing to regular changes, i.e. with a pattern that can be checked

Generic modules

- MESSAGE system for accepting and delivering messages
- CODE for transforming messages between global and subsystem types
- IN_TRAY for receiving and storing messages
- SECRETARY for filling IN_TRAY and, perhaps, dealing with some messages
- COUNTER for generating (locally unique) message numbers

Each office has instances of the last four, plus stub modules to replace lower-level office module.

Stub modules

For each function in lower module called by upper:

define a function of the same name and type to

1. get a new message number
2. code and send message to lower module
3. collect message with this number from in-tray
4. decode and return data from message

Provided communication works, the stub function behaves just like the original function.

Conclusions

- Metatheorem that semantics of hierachical calls are preserved (almost)
- Reliable transformational method supported by metatheory gives “correctness by construction”; proofs are avoided.

References

[1] Do Tien Dung, Le Linh Chi, Nguyen Le Thu, Phung Phuong Nam, Tran Mai Lien, and Chris George. Developing a Financial Information System. Technical Report 81, UNU-IIST, P.O.Box 3058, Macau, September 1996.

[2] Do Tien Dung, Chris George, Hoang Xuan Huan, and Phung Phuong Nam. A Financial Information System. Technical Report 115, UNU-IIST, P.O.Box 3058, Macau, July 1997. Partly published in *Requirements Targeting Software and Systems Engineering*, LNCS 1526, Springer-Verlag, 1998.