Formal Software Development Using RAISE

Chris George

United Nations University

International Institute for Software Technology

Macao SAR, China

Contents

Introduction What are formal methods? What is RAISE?

Types The types defined in the RAISE Specification Language (RSL) and how to define new ones.

Subtypes Subtypes, maximal types and type checking.

Sets, lists, and maps Type constructors defined in RSL.

Logic The conditional logic used in RSL.

Proof rules Proof rules for language definition and proof rules for proof.

Imperative RSL So far everything is applicative; now we make things imperative.

Concurrent RSL Now we can also make things concurrent.

Modules Large specifications, like large programs, need to be modular.

Method The method for creating and developing specifications into software.

Harbour example A simple information system.

Lift example A harder problem with safety concerns and concurrency.

Communication example A larger example showing a design decomposition.

System example An example of a vary large system.

RAISE resources

- Home page: `http://www.iist.unu.edu/www/raise`
- RAISE tools: `http://www.iist.unu.edu/newrh/III/1/page.html`
- ftp site: `ftp://ftp.iist.unu.edu/pub/RAISE`

  rsltc Tools

  method_book Method book

  case_studies Case studies book

  course_material This course

  Chinese Tutorial in Chinese

Introduction to Formal Methods

Chris George

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

Macao

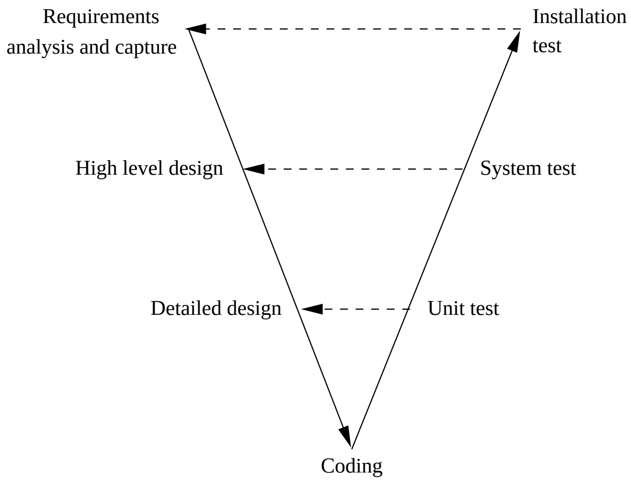
UNU-IIST

The problem is ...

The problem is we find that there is no way to describe the system based on the customer's requirements.

So we must describe the system itself.

V-diagram model of software life cycle



Another actual quotation

The trouble with formal methods is that you have to think too hard at the beginning.

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*

Aims

To produce software that is

- more likely to meet its requirements
- less likely to contain errors
- more reliable
- better documented
- easier to maintain

Formality

Formal specification language:

- precise syntax
- mathematical meaning (semantics)
- abstraction

Formality =>

- unambiguous
- formal reasoning (prove properties)

Abstraction =>

- high level view: ignore implementation details

RAISE?

Radical Alternative for Inadequate Software Engineers

Rambling Around In Search of Enlightenment

Rigorous Approach to Industrial Software Engineering

RAISE is a product consisting of:

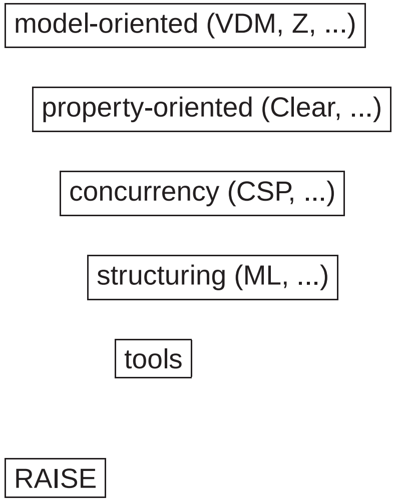
- a method for software development
- a formal specification language: RSL
- computer based tools

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
$\times$	`><`	*	`-list`	$^{\omega}$	`-inflist`	$\rightarrow$	`->`
$\overset{\sim}{\rightarrow}$	`-~->`	$\underset{m}{\rightarrow}$	`-m->`	$\overset{\sim}{\underset{m}{\rightarrow}}$	`-~m->`	$\leftrightarrow$	`<->`
$\wedge$	`/\`	$\vee$	`\/`	$\Rightarrow$	`=>`	$\forall$	`all`
$\exists$	`exists`	$\bullet$	`:-`	$\Box$	`always`	$\equiv$	`is`
$\neq$	`~=`	$\leq$	`<=`	$\geq$	`>=`	$\uparrow$	`**`
$\in$	`isin`	$\notin$	`~isin`	$\subset$	`<<`	$\subseteq$	`<<=`
$\supset$	`>>`	$\supseteq$	`>>=`	$\cup$	`union`	$\cap$	`inter`
$\dagger$	`!!`	$\langle$	`<.`	$\rangle$	`.>`	$\mapsto$	`+>`
$\parallel$	`||`	╫	`++`	▯	`|=|`	$\sqcap$	`|^|`
		λ	`-\`	$\circ$	`#`		

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 $\underset{m}{\rightarrow}$ 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 : **Int** • 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:	$T \times U,\ T \times U \times V,\ \ldots$	(t,u), (t,u,v)
Set:	T-**set**, T-**infset**	$\{\},\ \{t_1,t_2\}$
List:	$T^*,\ T^\omega$	$\langle\rangle,\ \langle t_1,t_2\rangle$
Map:	$T \underset{m}{\rightarrow} U,\ T \overset{\sim}{\underset{m}{\rightarrow}} U$	$[\,],\ [\,t_1 \mapsto u_1,\ t_2 \mapsto u_2\,]$
Function:	$T \rightarrow U,\ T \overset{\sim}{\rightarrow} U$	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
$\{\ 2{*}x{+}1 \mid x : \textbf{Int} \bullet x \in \{0..4\}\ \}$,
$\langle\ x \mid x$ **in** $\langle 0..10\rangle \bullet$ is_odd(x)$\rangle$, and
$[\,f(x) \mapsto g(x) \mid x : \textbf{Int} \bullet p(x)\ ]$

Products

A product is

an ordered finite collection
of
values of possibly different types

Examples:

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

Product Type Expressions

$\text{type\_expr}_1 \times \ldots \times \text{type\_expr}_n,\ n \geq 2$

Values:

$(v_1,\ldots,v_n),\ v_i : \text{type\_expr}_i$

Operators:

$=$
$\neq$

Example: A System of Coordinates I

```
scheme 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

```
scheme 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

```
scheme 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 $\overset{\sim}{\rightarrow}$ Val

Only variant type definitions may be recursive.

Case expressions

```
value
    will_die_before_the_end : Cowboy → Bool
    will_die_before_the_end(c) ≡
        case c of
            good → false,
            _ → true
        end,

    traverse : Tree → Val*
    traverse(t) ≡
        case t of
            nil → ⟨⟩,
            node(l, v, r) → traverse(l) ⌒ ⟨v⟩ ⌒ traverse(r)
        end
```

Partial functions: example

```
value
    factorial : Nat $\overset{\sim}{\to}$ Nat
    factorial(n) ≡
        if n = 1 then 1 else n ∗ factorial(n − 1) end
    pre n > 0
```

A partial function has $\overset{\sim}{\to}$ in its signature and **pre** in its definition.

Implicit functions: example

```
value
    square_root : Real $\overset{\sim}{\to}$ 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 $\overset{\sim}{\to}$ 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.

Subtypes and preconditions

Chris George

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

Contents

Subtypes

- subtype expressions
- maximal types and type checking

Preconditions

- relation to subtypes
- what they mean
- when to use them

Subtype Expressions

Examples:

$\{|$ l : **Int*** • **len** l > 0 $|\}$
$\{|$ t : Tree • is_ordered_tree(t) $|\}$

General form:

$\{|$ binding : type_expr • *logical*-value_expr $|\}$

Maximal types

The maximal types are

- **Bool**, **Int**, **Real**, **Char**, **Unit**
- Sorts
- Type expressions composed from maximal types and the type constructors $\times$, **-infset**, $^\omega$, $\overset{\sim}{\underset{m}{\rightarrow}}$, $\overset{\sim}{\rightarrow}$
- Type identifiers defined as abbreviations for maximal types.

Examples of non-maximal types:

- **Nat**, **Text** ($=$ **Char***)
- Type expressions involving the type constructors **-set**, *, $\underset{m}{\rightarrow}$, $\rightarrow$
- Subtypes (unless the type_expr is maximal and the *logical*-value_expr is **true**)

Type checking only checks maximal types.

Example

If f is defined

```
value
   f : Nat → Nat
   f(n) ≡
      if n = 0 ∨ n = 1 then 1
      else n ∗ f(n−1) end
```

then f(-1) is not a type error.

Preconditions and subtypes

Subtypes in argument types are equivalent to preconditions:

```
f : Nat → Int
f(x) ≡ ...
```

is equivalent to

```
f : Int ⥲ Int
f(x) ≡ ...
pre x ≥ 0
```

Semantics of preconditions

```
f : Int ⥲ Int
f(x) ≡ ...
pre x ≥ 0
```

means that ... is the meaning of f(x) when $x \geq 0$.

Nothing is known about the meaning of f(x) otherwise (except that if it has a value it must be an **Int** value).

Preconditions

- A precondition may be assumed to be true inside the function body.
- Checking a precondition is the responsibility of the caller.
- Preconditions in top level functions should be checked at the user interface.
- Preconditions are used
  - because a partial function or operator is used inside the function body, or to ensure termination, or
  - to prompt a check elsewhere

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

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

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

Overloading of hd

Theory:

$\mathbf{hd} : \text{T-}\mathbf{infset} \overset{\sim}{\to} \text{T}$
$\mathbf{hd}(s)\ \mathbf{as}\ x\ \mathbf{post}\ x \in s$
$\mathbf{pre}\ s \neq \{\}$

Example:

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

i.e.

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

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

Set Value Expressions

Enumerated: $\{\text{expr}_1,\ldots,\text{expr}_n\}$

$\{1,2\}$
$\{1,2,1\}$

Ranged: $\{\textit{integer}\text{-expr}_1 \ .. \ \textit{integer}\text{-expr}_2\}$

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

Comprehended: $\{\text{expr}_1 \mid \text{typing}_1,\ldots,\text{typing}_n \bullet \textit{logical}\text{-expr}_2\}$

$\{2{*}n \mid n : \mathbf{Nat} \bullet n \leq 3\}$

```
scheme 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:

$\langle 1,3,3,1,5 \rangle$
$\langle$**true**,**false**,**true**$\rangle$

List Type Expressions

- type_expr*

  $\langle v_1, \ldots, v_n \rangle$

  where $n \geq 0$, v_i : type_expr

- type_expr$^\omega$

  $\langle v_1, \ldots, v_n \rangle$,
  $\langle v_1, \ldots, v_n, \ldots \rangle$

  where $n \geq 0$, v_i : type_expr

List Value Expressions

Enumerated: $\langle \text{expr}_1,\ldots,\text{expr}_n \rangle$

$\langle 1,3,3,1,5 \rangle$
$\langle$**true**,**false**,**true**$\rangle$

Ranged: $\langle \textit{integer}\text{-expr}_1 \mathrel{..} \textit{integer}\text{-expr}_2 \rangle$

$\langle 3 \mathrel{..} 7 \rangle = \langle 3,4,5,6,7 \rangle$
$\langle 3 \mathrel{..} 3 \rangle = \langle 3 \rangle$
$\langle 3 \mathrel{..} 2 \rangle = \langle\rangle$

Comprehended: $\langle \text{expr}_1 \mid \text{binding } \mathbf{in}\ \textit{list}\text{-expr}_2 \bullet \textit{logical}\text{-expr}_3 \rangle$

$\langle 2{*}n \mid n$ **in** $\langle 0 \mathrel{..} 3 \rangle\rangle$
$\langle n \mid n$ **in** $\langle 0 \mathrel{..} 100 \rangle \bullet$ is_even(n)$\rangle$

List Indexing

Basic form:

$\textit{list}\text{-expr}(\textit{integer}\text{-expr}_1)$

Example:

$\langle 2,5,3 \rangle(2) = 5$

Associated Built-in Operators

$\widehat{}\ : T^* \times T^\omega \to T^\omega$	$\langle e_1,\ldots,e_n \rangle \widehat{}\ \langle e_{n+1},\ldots \rangle = \langle e_1,\ldots,e_n,e_{n+1},\ldots \rangle$
hd : $T^\omega \overset{\sim}{\to} T$	**hd** $\langle e_1,e_2,\ldots \rangle = e_1$
tl : $T^\omega \overset{\sim}{\to} T^\omega$	**tl** $\langle e_1,e_2,\ldots \rangle = \langle e_2,\ldots \rangle$
len : $T^\omega \overset{\sim}{\to}$ **Nat**	**len** $\langle e_1,\ldots,e_n \rangle = n$ **len** il $\equiv$ **chaos**
elems : $T^\omega \to$ T-**infset**	**elems** $\langle e_1,e_2,\ldots \rangle = \{e_1,e_2,\ldots\}$

inds : $T^\omega \to$ **Nat-infset**	**inds** fl = {1 .. **len** fl}
	inds il = {idx | idx : **Nat** • idx $\geq$ 1}

Overloading of ∈ and ∉

$\in : T \times T^\omega \to$ **Bool**	$'d' \in \langle 'a', 'b', 'c' \rangle =$ **false**
$\notin : T \times T^\omega \to$ **Bool**	$'a' \notin \langle 'a', 'b', 'c' \rangle =$ **false**

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

```
scheme 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
```

```
scheme 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
```

Case expressions

Lists are often analysed by case expressions, as in:

```
value
   reverse : T* → T*
   reverse(l) ≡
      case l of
         ⟨⟩ → ⟨⟩,
         ⟨h⟩⌒t → reverse(t)⌒⟨h⟩
      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:

$[''\texttt{Klaus}'' \mapsto 7, ''\texttt{John}'' \mapsto 2, ''\texttt{Mary}'' \mapsto 7]$
$[1 \mapsto 2, 5 \mapsto 10]$

Maps may be:

- infinite
- partial
- non-deterministic

Map Type Expressions

- $\mathrm{type\_expr}_1 \underset{m}{\rightarrow} \mathrm{type\_expr}_2$

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

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

- $\mathrm{type\_expr}_1 \overset{\sim}{\underset{m}{\rightarrow}} \mathrm{type\_expr}_2$

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

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

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

Examples

Nat $\xrightarrow[m]{}$ **Bool**

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

Nat $\overset{\sim}{\xrightarrow[m]{}}$ **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$_1 \mapsto$ expr$_1{}'$,. . . ,expr$_n \mapsto$ expr$_n{}'$]

$[\,3 \mapsto$ **true**, $5 \mapsto$ **false** $]$
$[\,{}''$`Klaus`$'' \mapsto 7,\ ''$`John`$'' \mapsto 2,\ ''$`Mary`$'' \mapsto 7\,]$

Comprehended: [expr$_1 \mapsto$ expr$_2 \mid$ typing$_1$,...,typing$_n$ • *logical*-expr$_3$]

[n $\mapsto$ 2$*$n $\mid$ n : **Nat** • n $\leq$ 2] $=$ [0 $\mapsto$ 0, 1 $\mapsto$ 2, 2 $\mapsto$ 4]
[n $\mapsto$ 2$*$n $\mid$ n : **Nat**] $=$ [0 $\mapsto$ 0, 1 $\mapsto$ 2, 2 $\mapsto$ 4, . . .]

Map Application

Basic form:

map-expr(expr$_1$)

Examples:

[$''$`Klaus`$'' \mapsto 7,\ ''$`John`$'' \mapsto 2,\ ''$`Mary`$'' \mapsto 7$]($''$`John`$''$) $= 2$

[3 $\mapsto$ **true**, 3 $\mapsto$ **false**](3) $=$ **true** $\lceil\rceil$ **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$_1$ $\overset{\sim}{\xrightarrow[m]{}}$ T$_2$) $\rightarrow$ T$_1$**-infset**	**dom** $[\,3 \mapsto$ **true**, $5 \mapsto$ **false** $] = \{3, 5\}$ **dom** $[\,3 \mapsto$ **true**, $5 \mapsto$ **false**, $5 \mapsto$ **true** $] = \{3, 5\}$
rng : (T$_1$ $\overset{\sim}{\xrightarrow[m]{}}$ T$_2$) $\rightarrow$ T$_2$**-infset**	**rng** $[\,3 \mapsto$ **false**, $5 \mapsto$ **false** $] = \{$**false**$\}$ **rng** $[\,3 \mapsto$ **false**, $5 \mapsto$ **false**, $5 \mapsto$ **true** $] = \{$**false**, **true**$\}$
$\dagger$: (T$_1$ $\overset{\sim}{\xrightarrow[m]{}}$ T$_2$) $\times$ (T$_1$ $\overset{\sim}{\xrightarrow[m]{}}$ T$_2$) $\rightarrow$ (T$_1$ $\overset{\sim}{\xrightarrow[m]{}}$ T$_2$)	$[\,3 \mapsto$ **true**, $5 \mapsto$ **false** $]\ \dagger\ [\,5 \mapsto$ **true** $] = [\,3 \mapsto$ **true**, $5 \mapsto$ **true** $]$
$\cup$: (T$_1$ $\overset{\sim}{\xrightarrow[m]{}}$ T$_2$) $\times$ (T$_1$ $\overset{\sim}{\xrightarrow[m]{}}$ T$_2$) $\rightarrow$ (T$_1$ $\overset{\sim}{\xrightarrow[m]{}}$ T$_2$)	$[\,3 \mapsto$ **true**, $5 \mapsto$ **false** $] \cup [\,5 \mapsto$ **true** $] = [\,3 \mapsto$ **true**, $5 \mapsto$ **false**, $5 \mapsto$ **true** $]$

$\backslash : (T_1 \xrightarrow[m]{\sim} T_2) \times T_1\textbf{-infset} \rightarrow$ $(T_1 \xrightarrow[m]{\sim} T_2)$	$m \backslash s =$ $[\, d \mapsto m(d) \mid d : T_1 \bullet d \in \textbf{dom}\ m \wedge d \notin s \,]$ $[\, 3 \mapsto \textbf{true}, 5 \mapsto \textbf{false} \,] \backslash \{5, 7\} = [\, 3 \mapsto \textbf{true} \,]$
$/ : (T_1 \xrightarrow[m]{\sim} T_2) \times T_1\textbf{-infset} \rightarrow$ $(T_1 \xrightarrow[m]{\sim} T_2)$	$m\ /\ s =$ $[\, d \mapsto m(d) \mid d : T_1 \bullet d \in \textbf{dom}\ m \wedge d \in s \,]$ $[\, 3 \mapsto \textbf{true}, 5 \mapsto \textbf{false} \,]\ /\ \{5, 7\} = [\, 5 \mapsto \textbf{false} \,]$
$\circ : (T_2 \xrightarrow[m]{\sim} T_3) \times (T_1 \xrightarrow[m]{\sim} T_2) \rightarrow$ $(T_1 \xrightarrow[m]{\sim} T_3)$	$m_1 \circ m_2 =$ $[\, x \mapsto m_1(m_2(x)) \mid x : T_1 \bullet$ $x \in \textbf{dom}\ m_2 \wedge m_2(x) \in \textbf{dom}\ m_1 \,]$ $[\, 3 \mapsto \textbf{true}, 5 \mapsto \textbf{false} \,] \circ$ $[\, ''\texttt{Klaus}'' \mapsto 3, ''\texttt{John}'' \mapsto 7 \,]$ $= [\, ''\texttt{Klaus}'' \mapsto \textbf{true} \,]$
$\in : T_1 \times (T_1 \xrightarrow[m]{\sim} T_2) \rightarrow \textbf{Bool}$	$3 \in [\, 3 \mapsto \textbf{true} \,] = \textbf{true}$

```
scheme DATABASE =
   class
      type
         Database = Key $\underset{m}{\rightarrow}$ Data,
         Key, Data
      value
         empty : Database = [ ],

         insert : Key $\times$ Data $\times$ Database $\rightarrow$ Database
         insert(k,d,db) $\equiv$ db $\dagger$ [ k $\mapsto$ d ],

         remove : Key $\times$ Database $\rightarrow$ Database
         remove(k,db) $\equiv$ db $\backslash$ {k},
```

```
         defined : Key $\times$ Database $\rightarrow$ Bool
         defined(k,db) $\equiv$ k $\in$ db,

         lookup : Key $\times$ Database $\overset{\sim}{\rightarrow}$ Data
         lookup(k,db) $\equiv$ 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** $\overset{\sim}{\rightarrow}$ **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** 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:

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

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

	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

context ⊢ value_expr :≼ opt_access_desc_string **Bool**
context ⊢ **read-only-convergent** value_expr

---

context ⊢
    if value_expr **then** value_expr **else true end** ≃
        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]
$\mathsf{es} \subseteq \mathsf{es}' \setminus \mathsf{es}'' \simeq \textbf{true}$
$\quad \textbf{when convergent}(\mathsf{es}) \wedge \textbf{readonly}(\mathsf{es}) \wedge$
$\quad\quad \textbf{convergent}(\mathsf{es}') \wedge \textbf{readonly}(\mathsf{es}') \wedge$
$\quad\quad \textbf{convergent}(\mathsf{es}'') \wedge \textbf{readonly}(\mathsf{es}'') \wedge \mathsf{es} \subseteq \mathsf{es}' \wedge \mathsf{es} \cap \mathsf{es}'' = \{\}$

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

More soundness tests

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

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

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

```
scheme 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**:

$$\begin{aligned}
\langle x := 1 ; x \; , x := 2 ; x \rangle &\equiv x := 2 ; \langle 1,2 \rangle \\
\langle x := 2 ; x \; , x := 1 ; x \rangle &\equiv x := 1 ; \langle 2,1 \rangle \\
\\
x \; + (x := x + 1 ; x) &\equiv x := x + 1 ; 2 * x - 1 \\
(x := x + 1 ; x \,) + x &\equiv x := x + 1 ; 2 * x
\end{aligned}$$

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**
$1 \lceil\rceil 2 = 1 \lceil\rceil 2$	**true** $\lceil\rceil$ **false**
$1 \lceil\rceil 2 \equiv 1 \lceil\rceil 2$	**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

- Insert an object “A” instantiating the applicative module, and hide it.
- Add variable “v” with type “A.T” where “T” is the type of interest and hide it. (Can use several variables if the type is a product or record, and adapt below accordingly.)
- Copy constants and functions to be visible from applicative module.
- Remove type of interest from function signatures; fill holes with **Unit**.
- Give type “**Unit** → **write** v **Unit**” to each constant “c” of type of interest, and make the definition “c() ≡ v := A.c”
- Insert “**write** v” access in generator signatures.

- Insert “**read** v” access to observer signatures.
- Replace instances “U” of types defined in the applicative module with “A.U”. (Or add type definition “U = A.U”.)
- Remove formal parameters representing type of interest.
- For each generator “g” make its body “v := A.g(...)” where “...” is the formal parameters plus “v”.
- For each observer “f” make its body “A.f(...)” where “...” is the formal parameters plus “v”.
- In preconditions: remove type of interest arguments from applicative function calls; replace any other occurrences of the type of interest parameter with “v”.

Optionally, the type and value definitions from the applicative module can be “unfolded”. This may make the object “A” redundant.

Imperative example

```
scheme I_DATABASE = hide A, database in
   class
      object A : DATABASE
      variable database : A.Database
      value
         empty : Unit → write database Unit
         empty() ≡ database := A.empty,

         insert : A.Key × A.Data → write database Unit
         insert(k,d) ≡ database := A.insert(k, d, database),

         remove : A.Key → write database Unit
         remove(k) ≡ database := A.remove(k, database),
```

```
         defined : A.Key → read database Bool
         defined(k) ≡ A.defined(k, database),

         lookup : A.Key ⥲ read database A.Data
         lookup(k) ≡ A.lookup(k, database)
         pre defined(k)
   end
```

This is for “leaf” modules in a hierarchy. Non-leaf modules involve a similar but simpler transformation:

- Replace applicative scheme instantiations with the corresponding imperative ones.
- Remove the type of interest definition, and its occurrences in signatures, and the corresponding formal parameters.
- Add “write O.any” in generator signatures for each object “O”.
- Add “read O.any” in observer signatures for each object “O”.
- Adapt function bodies to use the imperative functions from the objects instead of the previous applicative ones.

While expressions

```
scheme 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

```
scheme 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

```
scheme FRACTION_SUM =
  class
    variable
      result : Real
    value
      fraction_sum : Nat $\overset{\sim}{\to}$ 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
```

```
scheme FRACTION_SUM = class
    value
      fraction_sum : Nat $\overset{\sim}{\to}$ Real
      fraction_sum(n) ≡
        local
          variable
            result : Real := 0.0
        in
          for i in ⟨1 .. n⟩ do
            result := result + 1.0/(real i)
          end ;
          result
        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:

  value_expr$_1$ $\|$ 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

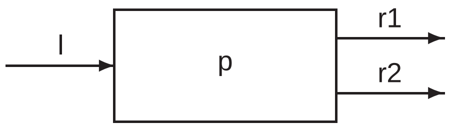
channel id : type_expr

Communication expressions:

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

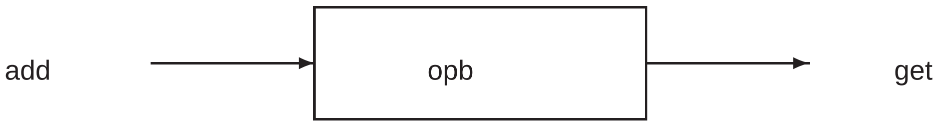
Input expressions have the same type as the channel.
Output expressions have type **Unit**.

Example

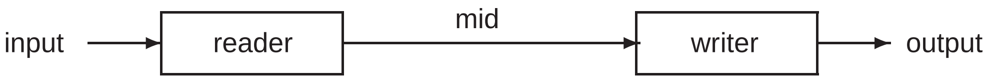


```
channel
   l, r1, r2: Int

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

Another example



```
scheme 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
```

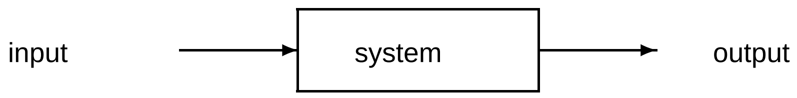


```
scheme READER_WRITER =
   class
      type Elem
      channel input, output, mid : Elem
      value
         reader : Unit → in input out mid Unit
         reader() ≡
            let v = input? in mid ! v end ; reader(),
         writer : Unit → in mid out output Unit
         writer() ≡
       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

```
scheme 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.



input

system

output

```
scheme 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

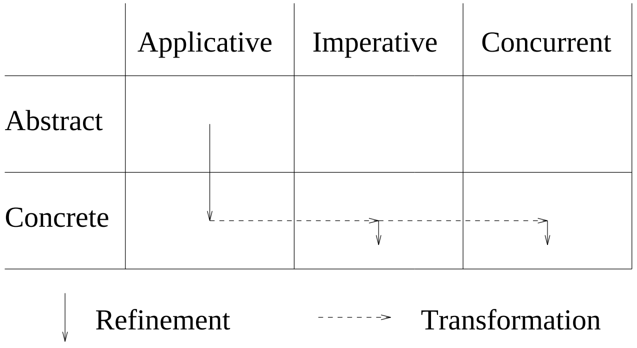
v:=c1? ⌈⌉ c2!e

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



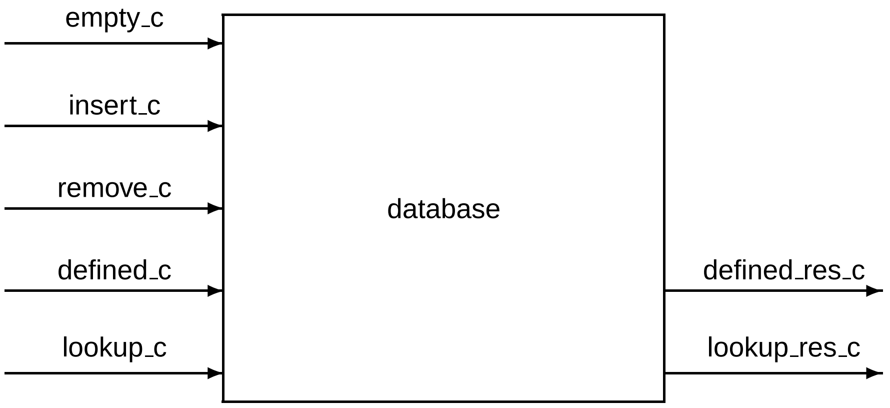
Applicative Imperative Concurrent

Abstract

Concrete

Refinement

Transformation



empty_c

insert_c

remove_c

defined_c

lookup_c

database

defined_res_c

lookup_res_c

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*.

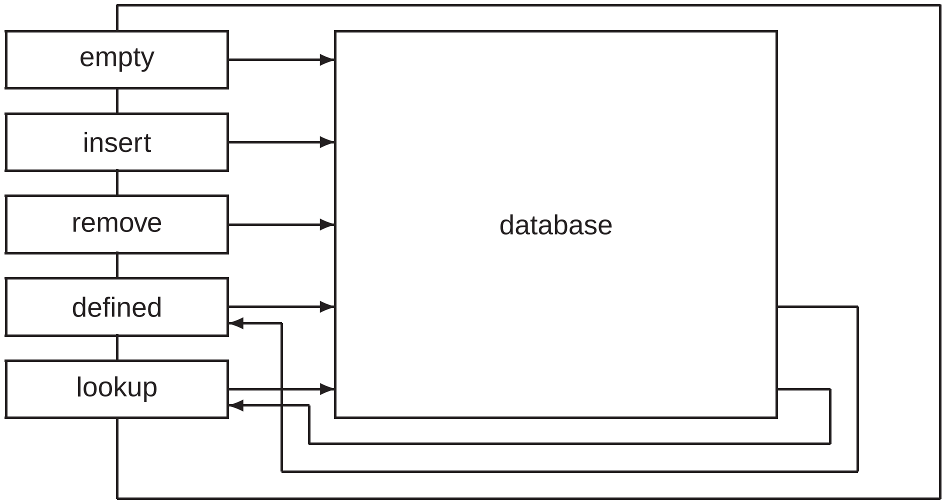
This is for “leaf” modules in a hierarchy. Non-loeaf modules are similar but easier.

```
scheme 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



```
scheme 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

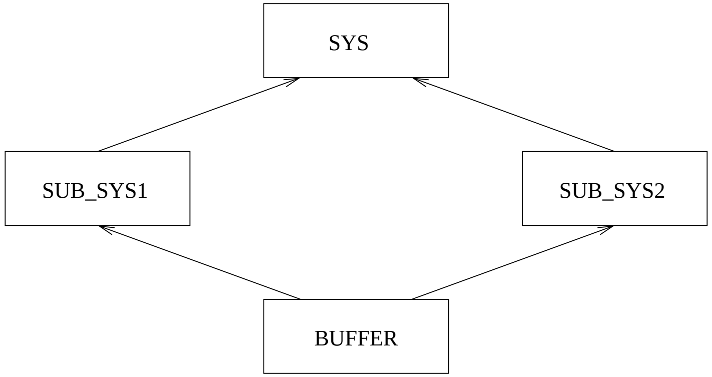
```
scheme DATABASE = ...

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

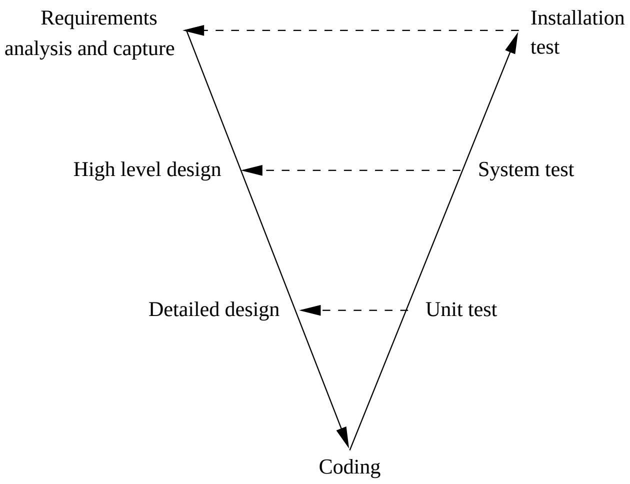
http://catless.ncl.ac.uk/Risks/

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



Requirements analysis and capture

Installation test

High level design

System test

Detailed design

Unit test

Coding

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 (⇒ implementation conditions)

Example 1

```
scheme S0 =
   class
      value x : Int
      axiom x ≥ 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

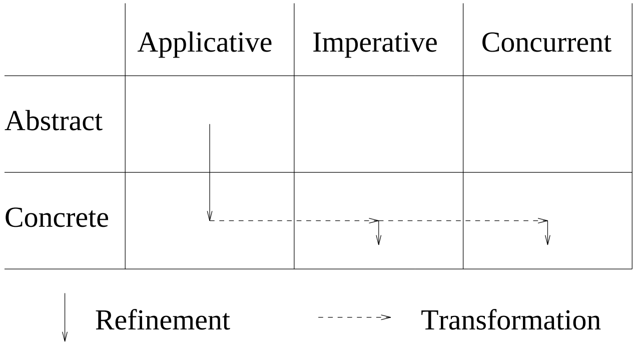
```
scheme S0 =
   hide z in class
      value x, y, z : Int
      axiom x > z ∧ z > y
   end

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



	Applicative	Imperative	Concurrent
Abstract			
Concrete			

Refinement

Transformation

Translation

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

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

Harbour example

Chris George

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

Requirements

Ships arriving at a harbour have to be allocated berths in the harbour which are vacant and which they will fit, or wait in a “pool” until a suitable berth is available.

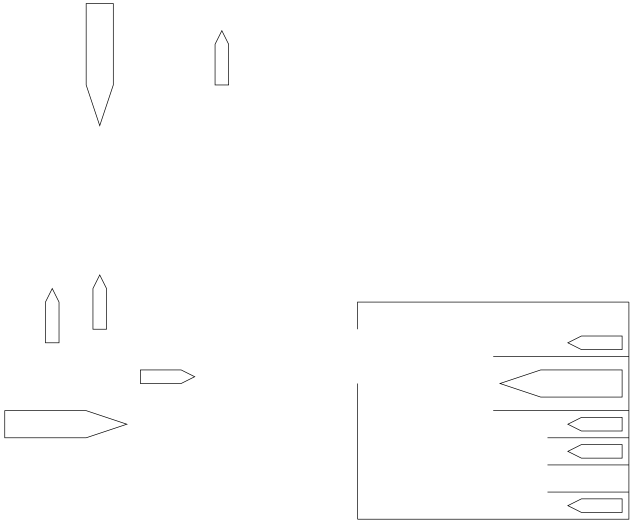
Develop a system providing the following functions to allow the harbour master to control the movement of ships in and out of the harbour:

arrive: to register the arrival of a ship

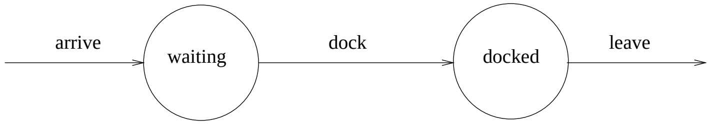
dock: to register a ship docking in a berth

leave: to register a ship leaving a berth

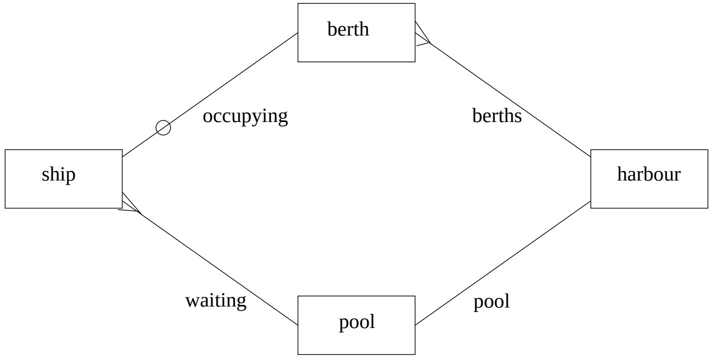
The harbour



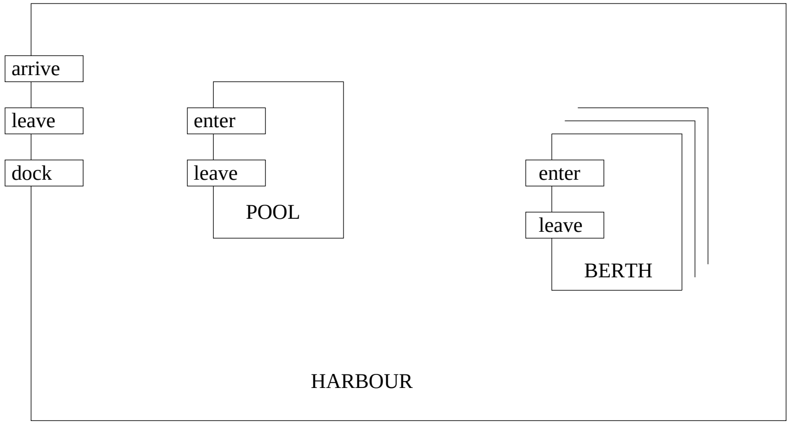
State transitions for ships



Entity relationship diagram



Harbour objects



Possible attributes

- Harbour
  - Pool (S)
  - (Set of) berths (S)
- Pool
  - (Set of) ships (D)
- Berth
  - Occupancy (D)
  - Size (S)
- Ship
  - Location (D)
  - Name (S)
  - Size (S)

“S” indicates a static attribute
“D” indicates a dynamic (state-dependent) attribute

Design decisions

- Don’t know components of “size” — length, width, depth/draught etc. So define

  value
  fits : Ship × Berth → **Bool**

  and leave underspecified.
- Name of ship unnecessary
- Location of ship can be calculated (to avoid duplication)

TYPES module

```
scheme TYPES =
   class
      type
         Ship, Berth,
         Occupancy == vacant | occupied_by(occupant : Ship)
      value
         fits : Ship × Berth → Bool
   end
```

We then make a global object from TYPES:

```
object T : TYPES
```

Consistency

1. a ship can't be in two places at once
2. at most one ship can be in any one berth
3. a ship can only be in a berth it fits

Two possibilities:

- build into model
- express as a predicate

2nd consistency condition in *Occupancy*; for 1st and 3rd we will use a predicate.

Design of state

Typically, especially for an information system, we start with the state, the information we need to hold:

- a collection of ships (the pool)
- a collection of berths

For the pool we will use a set.

For the berths we could use a map, but an array may be better, as the domain is fixed.

```
type
   Harbour ::
      pool : T.Ship-set
      berths : Berth_array
```

ARRAY_INIT_PARM

```
scheme ARRAY_INIT_PARM =
class
   type Elem
   value
      min, max : Int,
      init : Elem
   axiom [array_not_empty] max ≥ min
end
```

Instantiation

We can use *Occupancy* for *Elem*, *vacant* for *init*, but we need an integer index as an attribute (static) of *Berth*.

We extend *TYPES* with

```
type
   Berth_index = {| i : Int • i ≥ min ∧ max ≥ i |}
value
   min, max : Int,
   indx : Berth → Berth_index
axiom
   [ index_not_empty ] max ≥ min,
   [ berths_indexable ]
   ∀ b1, b2 : Berth •
      indx(b1) = indx(b2) ⇒ b1 = b2
```

A_ARRAY_INIT

```
scheme A_ARRAY_INIT(P : ARRAY_INIT_PARM) =
class
   type
      Array,
      Index = {| i : Int • i ≥ P.min ∧ P.max ≥ i |}

   value
      /∗ generators ∗/
      init : Array,
      change : Index × P.Elem × Array → Array,

      /∗ observer ∗/
      apply : Index × Array → P.Elem
```

```
   axiom
      [ apply_init ]
      ∀ i : Index • apply(i, init) ≡ P.init,

      [ apply_change ]
         ∀ i, i′ : Index, e : P.Elem, a : Array •
            apply(i′, change(i, e, a)) ≡
               if i = i′ then e else apply(i′, a) end
end
```

Design of functions

We start with the generators and observers. First things to decide are

- name
- parameter and result types
- whether partial or total

These three things form the *signature* of a function.

Generators

These are straightforward:

```
value
   arrive : T.Ship × Harbour →̃ Harbour,
   dock : T.Ship × T.Berth × Harbour →̃ Harbour,
   leave : T.Ship × T.Berth × Harbour →̃ Harbour
```

Observers 1

pool and *berths* are defined by the type *Harbour*. What else do we need? First, functions for preconditions.

```
value
   can_arrive : T.Ship × Harbour → Bool,
   can_dock : T.Ship × T.Berth × Harbour → Bool,
   can_leave : T.Ship × T.Berth × Harbour → Bool
```

Observers 2

Second, we try to define *consistent*:

```
value
   consistent : Harbour → Bool
   consistent(h) ≡
      (∀ s : T.Ship •
         ∼ (waiting(s, h) ∧ is_docked(s, h)) ∧
         (∀ b1, b2 : T.Berth •
            occupancy(b1, h) = T.occupied_by(s) ∧
            occupancy(b2, h) = T.occupied_by(s) ⇒
               b1 = b2) ∧
         (∀ b : T.Berth • occupancy(b, h) = T.occupied_by(s) ⇒ T.fits(s, b)))
```

Observers 3

To define *consistent* we have used some more observers:

```
value
   waiting : T.Ship × Harbour → Bool,
   is_docked : T.Ship × Harbour → Bool,
   occupancy : T.Berth × Harbour → T.Occupancy
```

We try to make observers total.

Now we are ready to write the first specification.

A_HARBOUR1

```
scheme A_HARBOUR1 =
hide B in class
   object
      B : A_ARRAY_INIT(T{Occupancy for Elem, vacant for init})

   type
      Harbour ::
         pool: T.Ship-set ↔ update_pool
         berths : Berth_array ↔ update_berths,
      Berth_array = B.Array
```

```
   value
      /∗ generators ∗/
      arrive : T.Ship × Harbour →̃ Harbour
      arrive(s, h) ≡
         let new_pool = pool(h) ∪ {s}
         in
            update_pool(new_pool, h)
         end
      pre can_arrive(s, h),
```

```
      dock : T.Ship × T.Berth × Harbour →̃ Harbour
      dock(s, b, h) ≡
         let
            new_pool = pool(h) \ {s},
            new_berths = B.change(T.indx(b), T.occupied_by(s), berths(h))
         in
            mk_Harbour(new_pool, new_berths)
         end
      pre can_dock(s, b, h),
```

```
      leave : T.Ship × T.Berth × Harbour →̃ Harbour
      leave(s, b, h) ≡
         let new_berths = B.change(T.indx(b), T.vacant, berths(h))
         in
            update_berths(new_berths, h)
         end
      pre can_leave(s, b, h),
    /∗ observers ∗/
    waiting : T.Ship × Harbour → Bool
    waiting(s, h) ≡ s ∈ pool(h),

    occupancy : T.Berth × Harbour → T.Occupancy
    occupancy(b, h) ≡ B.apply(T.indx(b), berths(h)),

    is_docked : T.Ship × Harbour → Bool
    is_docked(s, h) ≡
       (∃ b : T.Berth •
          occupancy(b, h) = T.occupied_by(s)),
```

```
    /∗ guards ∗/
    can_arrive : T.Ship × Harbour → Bool
    can_arrive(s, h) ≡
       ∼ waiting(s, h) ∧ ∼ is_docked(s, h),

    can_dock : T.Ship × T.Berth × Harbour → Bool
    can_dock(s, b, h) ≡
       waiting(s, h) ∧ ∼ is_docked(s, h) ∧
       occupancy(b, h) = T.vacant ∧
       T.fits(s, b),

    can_leave : T.Ship × T.Berth × Harbour → Bool
    can_leave(s, b, h) ≡
       occupancy(b, h) = T.occupied_by(s)
end
```

Validation 1

Have we met the main requirements?

1. Ships can arrive and will be registered
2. Ships can be docked when a suitable berth is free
3. Docked ships can leave
4. Ships can only be allocated to berths they fit
5. Any ship will eventually get a berth
6. Any ship waiting more than 2 days will be flagged
7. ...

Validation 2

Requirements might

- be met
- be deferred; be met later
- be removed; not be met
- make us rework the specification

Finished applicative development?

- All functions explicit (though *is_docked* not translatable)
- Standard module A_ARRAY_INIT can be ignored

So ready for next step — to imperative style.

I_HARBOUR1

```
scheme I_HARBOUR1 =
hide B in
class
   object
      B : I_ARRAY_INIT(T{Occupancy for Elem, vacant for init})

   variable
      pool : T.Ship-set := {}
```

```
value
   /∗ generators ∗/
   arrive : T.Ship ⥲ write any Unit
   arrive(s) ≡
      pool := pool ∪ {s}
   pre can_arrive(s),

   dock : T.Ship × T.Berth ⥲ write any Unit
   dock(s, b) ≡
      pool := pool \ {s};
      B.change(T.indx(b), T.occupied_by(s))
   pre can_dock(s, b),

   leave : T.Ship × T.Berth ⥲ write any Unit
   leave(s, b) ≡ B.change(T.indx(b), T.vacant)
   pre can_leave(s, b),
```

```
   /∗ observers ∗/
   waiting : T.Ship → read any Bool
   waiting(s) ≡ s ∈ pool,

   occupancy : T.Berth → read any T.Occupancy
   occupancy(b) ≡ B.apply(T.indx(b)),

   is_docked : T.Ship → read any Bool
   is_docked(s) ≡
      (∃ b : T.Berth •
         occupancy(b) = T.occupied_by(s)),
    /∗ guards ∗/
    can_arrive : T.Ship → read any Bool
    can_arrive(s) ≡
       ∼ waiting(s) ∧ ∼ is_docked(s),

    can_dock : T.Ship × T.Berth → read any Bool
    can_dock(s, b) ≡
       waiting(s) ∧ ∼ is_docked(s) ∧
       occupancy(b) = T.vacant ∧ T.fits(s, b),

    can_leave : T.Ship × T.Berth → read any Bool
    can_leave(s, b) ≡
       occupancy(b) = T.occupied_by(s)
end
```

Validation and verification

Validation :

- Have we taken any more requirements into account?
- If so, are they satisfied?

Verification :

Idea of method is that this is not done; we have no abstract imperative version for which to show implementation. Instead we argue for “correctness by construction”.

Consistency

```
value
   consistent : Unit → read any Bool
   consistent() ≡
      (∀ s : T.Ship •
         ∼ (waiting(s) ∧ is_docked(s)) ∧
         (∀ b1, b2 : T.Berth •
            occupancy(b1) = T.occupied_by(s) ∧
            occupancy(b2) = T.occupied_by(s) ⇒
               b1 = b2) ∧
         (∀ b : T.Berth • occupancy(b) = T.occupied_by(s) ⇒ T.fits(s, b)))
```

Approaches to consistency 1

Include *consistent* in the applicative specification and prove some theorems:

1. The initial state is consistent

```
in A_HARBOUR1 ⊢ consistent(mk_Harbour({}, B.init))
```

2. Each generator preserves consistency, e.g.

```
in A_HARBOUR1 ⊢
   ∀ s : T.Ship, b : T.Berth, h : Harbour •
      consistent(h) ∧ can_dock(s, b, h) ⇒ consistent(dock(s, b, h))
```

Approaches to consistency 2

Include *consistent* in the imperative as well as the applicative specification, add it as a precondition to each generator, and include it as a postcondition as well, e.g.

```
value
   dock : T.Ship × T.Berth ∼→ write any Unit
   dock(s, b) ≡
      pool := pool \ {s};
      B.change(T.indx(b), T.occupied_by(s))
   post consistent()
   pre consistent() ∧ can_dock(s, b),
```

The translators will now include (optional) code for checking consistency before and after each call of a generator.

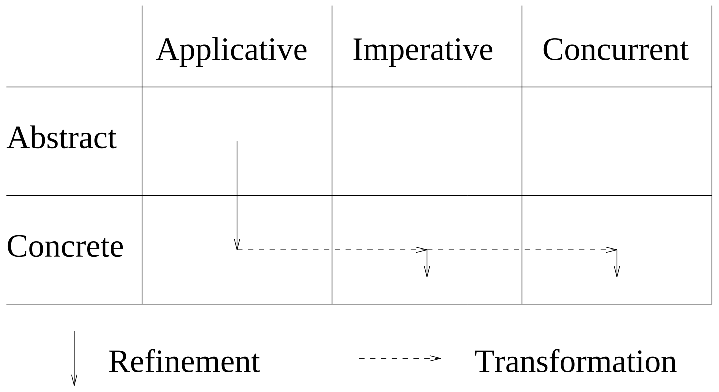
This is a currently undocumented extension to RSL.

Approaches to consistency 3

Define the theory suggested in approach 1 but perform the proofs only mentally.

These three approaches give gradually less effort and less assurance. Which you choose is a tradeoff, and depends on the effort available and the degree of assurance of correctness you want.

Typical Development



	Applicative	Imperative	Concurrent
Abstract			
Concrete			

(Apart from *consistent*), only non-translatable function is *is_docked*. Develop to I_HARBOUR2 with *is_docked* defined by

```
value
   is_docked : T.Ship → read any Bool
   is_docked(s) ≡
      (∃ i : Int • i ∈ {T.min .. T.max} ∧ B.apply(i) = T.occupied_by(s))
```

Verify implementation by showing this satisfies the definition in I_HARBOUR1.

Completion

- translation of I_HARBOUR2
- translation (if necessary) of standard module I_ARRAY_INIT
- testing
- installation
- testing

Lift example

Chris George

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

Requirements

Provide the control software for a single lift for a building of 9 floors (LG, G, 1, 2, ... 8) with automatic doors and ...

It is important that the system is

- safe
- reliable
- effective

What is a lift?

- Intrinsics
- Technology
- People
- Hardware

Intrinsics

- Cage
- Door
- Button
- Indicators
- Floor
- Motor

Technology

- Cage
  - front or front/back doors?
  - single or double height?
- Motor
  - move, halt
  - go to floor (any? restricted?)
- Door
  - double, automatic - open/close
  - manual - lock/unlock
- Indicator
  - electronic - on/off
  - analogue - set
- Button
  - up/down
  - go to floor

People

- users: press buttons; enter/leave
- maintainers: ?
- inspectors: ?

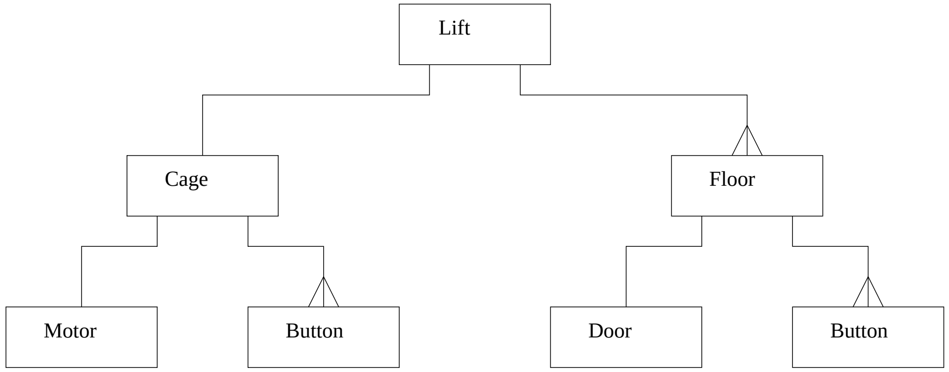
Hardware

- processor
- programming language
- connections/communications
- hardware interfaces

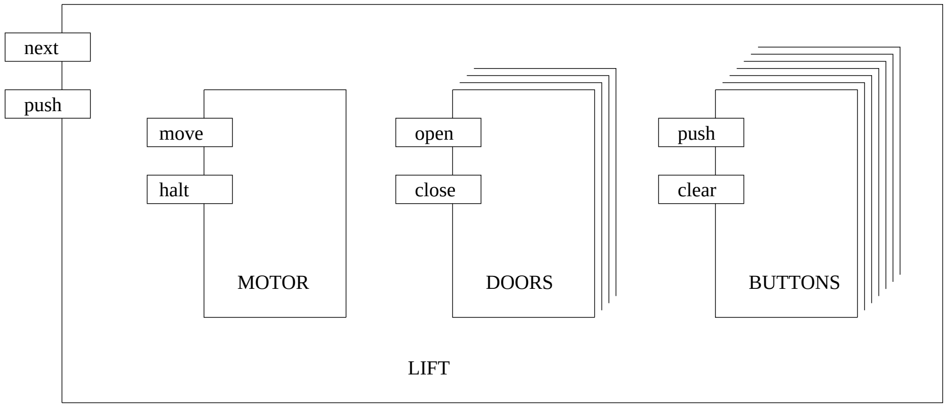
Assumptions

- Doors mostly hardware — open, close — and lift doors (if any) operated with floor doors
- Motor — up to next, down to next, halt at next
- Real time of little importance (mechanisms much slower than processor) but relative time of some events may be critical
- Indicators are hardware triggered by the cage — can ignore
- Hardware failures and need for maintenance ignored
- Floors numbered consecutively
- Cage is single doored, single height

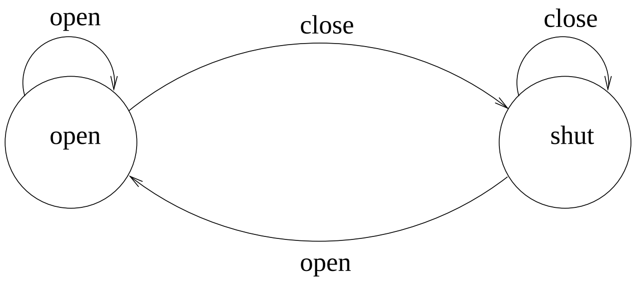
Entity relationships



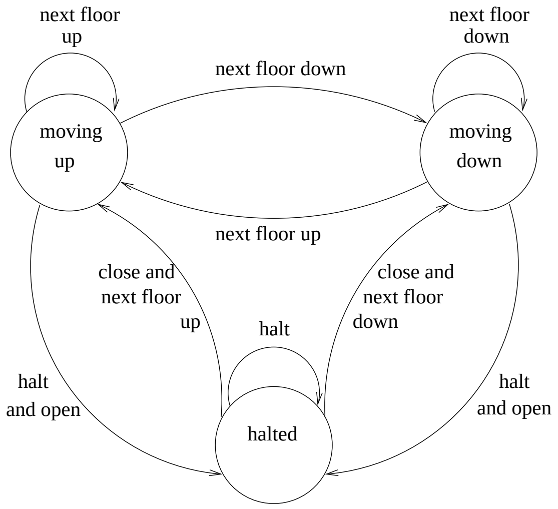
Components



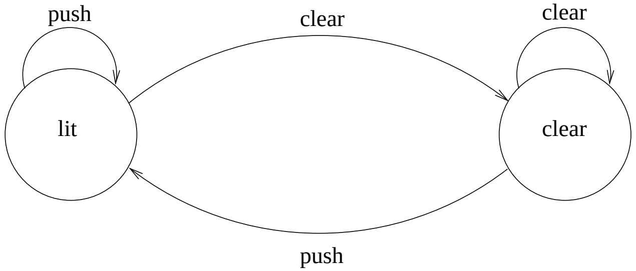
State transitions for doors



State transitions for lift



State transitions for buttons



TYPES module

```
scheme TYPES = class
    value
      min_floor, max_floor : Int,
      is_floor : Int → Bool
      is_floor(f) ≡ f ≥ min_floor ∧ f ≤ max_floor
    axiom [some_floors] max_floor > min_floor
    type
      Floor = {| n : Int • is_floor(n) |},
      Lower_floor = {| f : Floor • f < max_floor |},
      Upper_floor = {| f : Floor • f > min_floor |},
      Door_state == open | shut,
      Button_state == lit | clear,
      Direction == up | down,
      Movement == halted | moving,
      Requirement :: here : Bool  after : Bool  before : Bool
```

```
    value
      next_floor : Direction × Floor ~→ Floor
      next_floor(d, f) ≡
        if d = up then f + 1 else f − 1 end
        pre is_next_floor(d, f),

      is_next_floor : Direction × Floor → Bool
      is_next_floor(d, f) ≡
        if d = up then f < max_floor else f > min_floor end,

      invert : Direction → Direction
      invert(d) ≡ if d = up then down else up end
  end
```

Hazard analysis

What damage can a lift system do to

- people
- itself
- other things
- your finances

The wrong answer

USE THIS LIFT
AT YOUR OWN RISK

No liability accepted — Ethical Lift Co.

Safety and Liveness

Safety property : An event will never happen

$\Box\ \forall\ \mathrm{f} : \mathrm{Floor} \bullet$
$\quad \sim(\mathrm{movement} = \mathrm{moving} \wedge \mathrm{floor} = \mathrm{f} \wedge$
$\quad\quad \mathrm{door\_state(f)} = \mathrm{open})$

Can generally be stated in RSL: □ means "in all states".

Liveness property : An event will eventually happen

$\Box\ \forall\ \mathrm{f} : \mathrm{Floor} \bullet$
$\quad \mathrm{lift\_button(f)} = \mathrm{lit} \Rightarrow$
$\quad\quad \Diamond\ \mathrm{floor} = \mathrm{f} \wedge \mathrm{movement} = \mathrm{halted} \wedge$
$\quad\quad \mathrm{door\_state(f)} = \mathrm{open}$

Cannot in general be stated in RSL: no ◇

Control processes

Typical imperative structure is

```
while true do
  read sensors ;
  take next appropriate action
end
```

In applicative specification:

"read sensors" will be a generator of type

$\mathrm{State} \rightarrow \mathrm{Messages} \times \mathrm{State}$

It should be a generator not an observer, otherwise result would be predetermined (and constant for a constant state).

"take next appropriate action" will be a generator of type

$\mathrm{State} \times \mathrm{Messages} \rightarrow \mathrm{State}$

Lift generators

value
$\quad \mathrm{check\_buttons} : \mathrm{Lift} \rightarrow \mathrm{T.Requirement} \times \mathrm{Lift}$

$\quad \mathrm{next} : \mathrm{T.Requirement} \times \mathrm{Lift} \xrightarrow{\sim} \mathrm{Lift}$

T is a an object, an instance of TYPES.

Lift observers

```
value
  movement : Lift → T.Movement,

  door_state : Lift → T.Floor → T.Door_state,

  floor : Lift → T.Floor,

  direction : Lift → T.Direction
```

Safety

- No falls into open lift shaft

  (door_state(s)(f) = T.open) ⇒ (movement(s) = T.halted ∧ floor(s) = f)

- No (unintentionally) starving occupants

  Lift eventually stops at requested floor and

  (movement(s) = T.halted ∧ floor(s) = f) ⇒ (door_state(s)(f) = T.open)

  Note this is partly a liveness property.

```
value
  safe : Lift → Bool
  safe(s) ≡
   (∀ f : T.Floor •
     (door_state(s)(f) = T.open) = (movement(s) = T.halted ∧ floor(s) = f))
```

Safety and liveness

Relation over three lift states:

$$s \xrightarrow{check\_buttons} s' \xrightarrow{next} s''$$

- If s is safe, so are s' and s''.
- If the lift is halted in state s'', the lift was wanted either “here” or nowhere else, and the floor of s'' is the same as the floor of s.
- If the lift is moving in state s'', the lift was wanted either “after” or “before”, and the floor it is moving towards is next to the floor in state s and a valid floor.
- If the lift has changed direction between states s and s'', “after” must be false.

Can then argue that the lift will eventually reach and stop at a floor for which a button is lit.

```
scheme A_LIFT0 = class
  type Lift
  value
    /∗ generators ∗/
    next : T.Requirement × Lift ~→ Lift,
    check_buttons : Lift → T.Requirement × Lift,
    /∗ observers ∗/
    movement : Lift → T.Movement,
    door_state : Lift → T.Floor → T.Door_state,
    floor : Lift → T.Floor,
    direction : Lift → T.Direction,
    /∗ derived ∗/
    safe : Lift → Bool
    safe(s) ≡
      (∀ f : T.Floor •
        (door_state(s)(f) = T.open) = (movement(s) = T.halted ∧ floor(s) = f))
```

```
axiom
  [ safe_and_useful ]
  ∀ s : Lift •
    safe(s) ⇒
    let (r, s′) = check_buttons(s) in
      safe(s′) ∧
      let s″ = next(r, s′) in
        safe(s″) ∧
        (movement(s″) = T.halted ⇒
          (T.here(r) ∨ (∼ T.after(r) ∧ ∼ T.before(r))) ∧ floor(s) = floor(s″)) ∧
        (movement(s″) = T.moving ⇒
          (T.after(r) ∨ T.before(r)) ∧
          T.is_next_floor(direction(s″), floor(s)) ∧
          floor(s″) = T.next_floor(direction(s″), floor(s))) ∧
        (direction(s) ≠ direction(s″) ⇒ ∼ T.after(r))
      end end end
```

Validation

Have we met the main requirements for the lift?

Next steps

- Provide algorithm for *next*
- Justify it satisfies *safe_and_useful*

Method:

- Define a new version A_LIFT1:
  - Define *next* in terms of two new generators *move* and *halt*.
  - *next* needs a precondition; define *check_buttons* by a postcondition stating that it
    1. provides precondition for *next*
    2. does not change direction, movement, floor or door_state attributes
- Justify A_LIFT1 $\preceq$ A_LIFT0

A_LIFT1

```
scheme A_LIFT1 = class
  type Lift
  value
    /∗ generators ∗/
    move : T.Direction × T.Movement × Lift ∼→ Lift,
    halt : Lift → Lift,
    check_buttons : Lift → T.Requirement × Lift,
    /∗ observers ∗/
    movement : Lift → T.Movement,
    door_state : Lift → T.Floor → T.Door_state,
    floor : Lift → T.Floor,
    direction : Lift → T.Direction,
    /∗ derived ∗/
    next : T.Requirement × Lift ∼→ Lift
```

```
next(r, s) ≡ let d = direction(s) in
    case movement(s) of
      T.halted →
        case r of
          T.mk_Requirement(_, true, _) → move(d, T.halted, s),
          T.mk_Requirement(_, _, true) → move(T.invert(d), T.halted, s),
          _ → s
        end,
      T.moving →
        case r of
          T.mk_Requirement(true, _, _) → halt(s),
          T.mk_Requirement(_, false, false) → halt(s),
          T.mk_Requirement(_, true, _) → move(d, T.moving, s),
          T.mk_Requirement(_, _, true) → move(T.invert(d), T.moving, s)
        end end end
  pre (T.after(r) ⇒ T.is_next_floor(direction(s), floor(s))) ∧
      (T.before(r) ⇒ T.is_next_floor(T.invert(direction(s)), floor(s))),
```

```
safe : Lift → Bool
safe(s) ≡
  (∀ f : T.Floor •
    (door_state(s)(f) = T.open) =
    (movement(s) = T.halted ∧ floor(s) = f))
```

```
axiom
  [ movement_move ]
    ∀ s : Lift, d : T.Direction, m : T.Movement •
      movement(move(d, m, s)) ≡ T.moving
        pre T.is_next_floor(d, floor(s)),
  [ door_state_move ]
    ∀ s : Lift, d : T.Direction, m : T.Movement, f : T.Floor •
      door_state(move(d, m, s))(f) ≡
        if m = T.halted ∧ floor(s) = f then T.shut
        else door_state(s)(f) end
        pre T.is_next_floor(d, floor(s)),
  [ floor_move ]
    ∀ s : Lift, d : T.Direction, m : T.Movement •
      floor(move(d, m, s)) ≡ T.next_floor(d, floor(s))
        pre T.is_next_floor(d, floor(s)),
```

```
[ direction_move ]
  ∀ s : Lift, d : T.Direction, m : T.Movement •
    direction(move(d, m, s)) ≡ d
      pre T.is_next_floor(d, floor(s)),
[ movement_halt ]
  ∀ s : Lift • movement(halt(s)) ≡ T.halted,
[ door_state_halt ]
  ∀ s : Lift, f : T.Floor •
    door_state(halt(s))(f) ≡
      if floor(s) = f then T.open else door_state(s)(f) end,
[ floor_halt ] ∀ s : Lift • floor(halt(s)) ≡ floor(s),
[ direction_halt ]
  ∀ s : Lift • direction(halt(s)) ≡ direction(s),
[ check_buttons_ax ]
  ∀ s : Lift •
    check_buttons(s) as (r, s′)
  post
    movement(s′) = movement(s) ∧
    door_state(s′) = door_state(s) ∧
    floor(s′) = floor(s) ∧
    direction(s′) = direction(s) ∧
    (T.after(r) ⇒
      T.is_next_floor(direction(s′), floor(s′))) ∧
    (T.before(r) ⇒
      T.is_next_floor(T.invert(direction(s′)), floor(s′)))
end
```

Validation and verification

Validation :

- Have we taken any more requirements into account?
- If so, are they satisfied?

Verification :

- Justification that A_LIFT1 $\preceq$ A_LIFT0

Next step

Decompose the state: motor, doors and buttons.

Method:

- Define A_MOTOR0, A_DOORS0 and A_BUTTONS0 modules
- Define A_LIFT2 using these modules
- Justify A_LIFT2 $\preceq$ A_LIFT1

A_MOTOR0

```
scheme A_MOTOR0 =
class
  type Motor
  value
    /* generators */
    move : T.Direction × Motor ~→ Motor,
    halt : Motor → Motor,
    /* observers */
    direction : Motor → T.Direction,
    movement : Motor → T.Movement,
    floor : Motor → T.Floor
```

```
axiom
  [ direction_move ]
    ∀ s : Motor, d : T.Direction •
      direction(move(d, s)) ≡ d
      pre T.is_next_floor(d, floor(s)),
  [ movement_move ]
    ∀ s : Motor, d : T.Direction •
      movement(move(d, s)) ≡ T.moving
      pre T.is_next_floor(d, floor(s)),
  [ floor_move ]
    ∀ s : Motor, d : T.Direction •
      floor(move(d, s)) ≡ T.next_floor(d, floor(s))
      pre T.is_next_floor(d, floor(s)),
```

```
  [ direction_halt ]
    ∀ s : Motor • direction(halt(s)) ≡ direction(s),
  [ movement_halt ]
    ∀ s : Motor • movement(halt(s)) ≡ T.halted,
  [ floor_halt ]
    ∀ s : Motor • floor(halt(s)) ≡ floor(s)
end
```

A_BUTTONS0

```
scheme A_BUTTONS0 =
class
  type Buttons
  value
    /∗ generators ∗/
    clear : T.Floor × Buttons → Buttons,

    check : T.Direction × T.Floor × Buttons → T.Requirement × Buttons
    check(d, f, s) as (r, s′) post
      (T.after(r) ⇒ T.is_next_floor(d, f)) ∧
      (T.before(r) ⇒ T.is_next_floor(T.invert(d), f))
end
```

```
scheme A_DOORS0 = class
  type Doors
  value
    /∗ generators ∗/
    open : T.Floor × Doors → Doors,
    close : T.Floor × Doors → Doors,
    /∗ observer ∗/
    door_state : Doors → T.Floor → T.Door_state
  axiom
    [ door_state_open ]
      ∀ f, f′ : T.Floor, s : Doors •
        door_state(open(f, s))(f′) ≡ if f = f′ then T.open else door_state(s)(f′) end,
    [ door_state_close ]
      ∀ f, f′ : T.Floor, s : Doors •
        door_state(close(f, s))(f′) ≡ if f = f′ then T.shut else door_state(s)(f′) end
end
```

A_LIFT2

```
scheme A_LIFT2 =
  hide M, DS, BS in
  class
    object
      /∗ motor ∗/
      M : A_MOTOR0,
      /∗ doors ∗/
      DS : A_DOORS0,
      /∗ buttons ∗/
      BS : A_BUTTONS0
    type Lift = M.Motor × DS.Doors × BS.Buttons
```

```
value
  /∗ generators ∗/
  move : T.Direction × T.Movement × Lift ⥲ Lift
  move(d, m, (ms, ds, bs)) ≡
    (M.move(d, ms),
     if m = T.halted then DS.close(M.floor(ms), ds)
     else ds end,
     bs)
    pre T.is_next_floor(d, M.floor(ms)),

  halt : Lift → Lift
  halt((ms, ds, bs)) ≡
    (M.halt(ms),
     DS.open(M.floor(ms), ds),
     BS.clear(M.floor(ms), bs)),
```

```
check_buttons : Lift → T.Requirement × Lift
check_buttons((ms, ds, bs)) ≡
  let (r, bs′) = BS.check(M.direction(ms), M.floor(ms), bs) in
    (r, (ms, ds, bs′))
  end,

/∗ derived ∗/
next : T.Requirement × Lift ⥲ Lift
```

```
next(r, (ms, ds, bs)) ≡
  let d = M.direction(ms) in case M.movement(ms) of
      T.halted →
        case r of
          T.mk_Requirement(_, true, _) → move(d, T.halted, (ms, ds, bs)),
          T.mk_Requirement(_, _, true) → move(T.invert(d), T.halted, (ms, ds, bs)),
          _ → (ms, ds, bs)
        end,
      T.moving →
        case r of
          T.mk_Requirement(true, _, _) → halt((ms, ds, bs)),
          T.mk_Requirement(_, false, false) → halt((ms, ds, bs)),
          T.mk_Requirement(_, true, _) → move(d, T.moving, (ms, ds, bs)),
          T.mk_Requirement(_, _, true) → move(T.invert(d), T.moving, (ms, ds, bs))
        end end end
  pre (T.after(r) ⇒ T.is_next_floor(M.direction(ms), M.floor(ms))) ∧
      (T.before(r) ⇒ T.is_next_floor(T.invert(M.direction(ms)), M.floor(ms))),
  safe : Lift → Bool
  safe((ms, ds, bs)) ≡
    (∀ f : T.Floor •
      (DS.door_state(ds)(f) = T.open) =
      (M.movement(ms) = T.halted ∧ M.floor(ms) = f))
end
```

Validation and verification

Validation :

- Have we taken any more requirements into account?
- If so, are they satisfied?

Verification :

- Justification that A_LIFT2 $\preceq$ A_LIFT1

Next step

We have a concrete state for A_LIFT2, but abstract states for the component modules A_MOTOR0, A_DOORS0 and A_BUTTONS0.

A_MOTOR1:

```
type Motor = T.Direction × T.Movement × T.Floor
```

A_DOORS1:

```
type Doors = T.Floor → T.Door_state
```

A_BUTTONS1:

```
type
  Buttons =
    (T.Floor -m-> T.Button_state) ×          -- lift buttons
    (T.Lower_floor -m-> T.Button_state) ×    -- up buttons
    (T.Upper_floor -m-> T.Button_state)      -- down buttons
```

A_MOTOR1

```
scheme A_MOTOR1 =
class
  type
    Motor = T.Direction × T.Movement × T.Floor
  value
    /∗ generators ∗/
    move : T.Direction × Motor -~-> Motor
    move(d′, (d, m, f)) ≡
      (d′, T.moving, T.next_floor(d′, f))
      pre T.is_next_floor(d′, f),

    halt : Motor → Motor
    halt((d, m, f)) ≡ (d, T.halted, f),
```

```
    /∗ observers ∗/
    direction : Motor → T.Direction
    direction((d, m, f)) ≡ d,

    movement : Motor → T.Movement
    movement((d, m, f)) ≡ m,

    floor : Motor → T.Floor
    floor((d, m, f)) ≡ f
end
```

```
scheme A_DOORS1 = class
  type Doors = T.Floor → T.Door_state

  value
    /∗ generators ∗/
    open : T.Floor × Doors → T.Floor → T.Door_state
    open(f, s)(f′) ≡
      if f = f′ then T.open else s(f′) end,

    close : T.Floor × Doors → T.Floor → T.Door_state
    close(f, s)(f′) ≡
      if f = f′ then T.shut else s(f′) end,

    /∗ observer ∗/
    door_state : Doors → T.Floor → T.Door_state
    door_state(s) ≡ s
end
```

A_BUTTONS1

```
scheme A_BUTTONS1 =
hide required_here, required_beyond in
class
  type
    Buttons =
      (T.Floor -m-> T.Button_state) ×         -- lift buttons
      (T.Lower_floor -m-> T.Button_state) ×   -- up buttons
      (T.Upper_floor -m-> T.Button_state)     -- down buttons
  value
    /∗ generators ∗/
    clear : T.Floor × Buttons → Buttons
    clear(f, (l, u, d)) ≡
    (l † [ f ↦ T.clear ],
     if f < T.max_floor then u † [ f ↦ T.clear ] else u end,
     if f > T.min_floor then d † [ f ↦ T.clear ] else d end),
/∗ observers ∗/
required_here : T.Direction × T.Floor × Buttons → Bool
required_here(d, f, (lift, up, down)) ≡
  lift(f) = T.lit ∨
  d = T.up ∧
  (f < T.max_floor ∧ up(f) = T.lit ∨
   f > T.min_floor ∧
   down(f) = T.lit ∧
   ∼ required_beyond(d, f, (lift, up, down))) ∨
  d = T.down ∧
  (f > T.min_floor ∧ down(f) = T.lit ∨
   f < T.max_floor ∧
   up(f) = T.lit ∧
   ∼ required_beyond(d, f, (lift, up, down))),
```

```
required_beyond : T.Direction × T.Floor × Buttons → Bool
required_beyond(d, f, s) ≡
  T.is_next_floor(d, f) ∧
  let f′ = T.next_floor(d, f) in
    required_here(d, f′, s) ∨ required_beyond(d, f′, s)
  end
```

```
check : T.Direction × T.Floor × Buttons → T.Requirement × Buttons,
check(d, f, s) as (r, s′)
  post
    r =
    T.mk_Requirement
      (required_here(d, f, s),
       required_beyond(d, f, s),
       required_beyond(T.invert(d), f, s))
end
```

Validation and verification

Validation :

- Have we taken any more requirements into account?
- If so, are they satisfied?

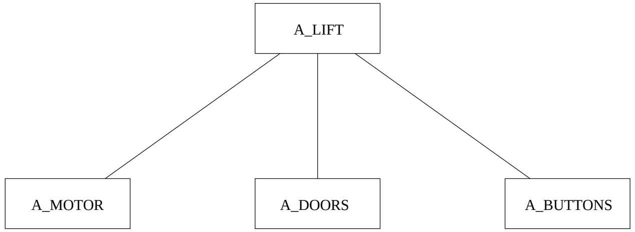
Verification :

- Justification that A_MOTOR1 $\preceq$ A_MOTOR0 etc.

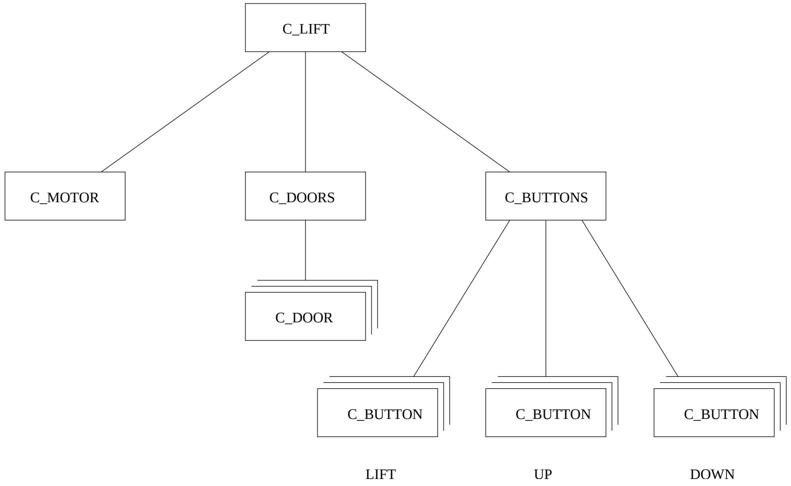
Next step

Create concurrent versions from applicative ones.

Applicative structure:



Concurrent structure:



Concurrent design paradigm

- “leaf” modules and “branch” modules
- only leaf modules have “state”; either variables or embedded sequential imperative modules (with variables or further embedded sequential imperative modules)
- all modules have “init” processes
- leaf modules have “main” (server) processes; after initialisation only these can access the state and they do not (normally) terminate
- init processes in branch modules call the init processes of their descendants in parallel
- init processes in leaf modules initialise the state and then call their main process

- leaf modules have channels
- generators and observers in a leaf module become “interface” processes that communicate with the main process and terminate
- generators and observers in a branch module are sequential or parallel combinations of calls of the generators or observers of their module or its descendants

Consequences

- call of init process at top level initialises all states and starts all main processes running
- call of a generator or observer at top level results in one or more (possibly concurrent) state changes or observations in the leaf modules, with any results passed back to top
- states of leaf modules are all independent
- there is no interference between imperative components; calls of interface functions of leaf modules may be arbitrarily ordered or interleaved
- BUT beware of “interference” in the real world

Applicative to concurrent transformation: branch

- Replace applicative objects with concurrent ones
- Remove type of interest from generator and observer function types; include **in any out any**; make total
- Adapt bodies to use imperative versions of functions
- Add init function

C_LIFT2

```
scheme C_LIFT2 =
hide M, DS, BS, move, halt in
class
  object
    /∗ motor ∗/
    M : C_MOTOR1,
    /∗ doors ∗/
    DS : C_DOORS1,
    /∗ buttons ∗/
    BS : C_BUTTONS1
```

```
value
  /∗ generators ∗/
  move :
   T.Direction × T.Movement → in any out any Unit
  move(d, m) ≡
    if m = T.halted then DS.close(M.floor()) end ;
    M.move(d),

  halt : Unit → in any out any Unit
  halt() ≡
    let f = M.floor() in BS.clear(f) ; M.halt() ; DS.open(f) end,

  check_buttons : Unit → in any out any T.Requirement
  check_buttons() ≡ BS.check(M.direction(), M.floor()),

  next : T.Requirement → in any out any Unit
```

```
next(r) ≡
  let d = M.direction() in
    case M.movement() of
      T.halted →
        case r of
          T.mk_Requirement(_, true, _) → move(d, T.halted),
          T.mk_Requirement(_, _, true) → move(T.invert(d), T.halted),
          _ → skip
        end,
      T.moving →
        case r of
          T.mk_Requirement(true, _, _) → halt(),
          T.mk_Requirement(_, false, false) → halt(),
          T.mk_Requirement(_, true, _) → move(d, T.moving),
          T.mk_Requirement(_, _, true) → move(T.invert(d), T.moving)
        end end end,
```

```
  /∗ initial ∗/
  init : Unit → in any out any write any Unit
  init() ≡ M.init() ‖ DS.init() ‖ BS.init(),

  /∗ control ∗/
  lift : Unit → in any out any Unit
  lift() ≡ while true do next(check_buttons()) end
end
```

Applicative to concurrent transformation: leaf

- Define state variables and/or imperative sequential objects to hold state
- Define channels for non-type of interest arguments and results of generators and observers
- Remove type of interest from generator and observer function types; include **in any out any**; make total; define bodies as outputs of arguments and inputs of results

- Define a “main” function with type

  ```
  Unit → in any out any write any Unit
  ```

  and body

  ```
  while true do e1 ▯ e2 ▯ ... end
  ```

  where ei interacts with a generator or observer, writes/reads variables and/or calls functions of imperative sequential objects
- Add init function to initialise variables and objects and then call main function

```
scheme C_MOTOR1 = hide CH, V, motor in class
  object
    CH : class
        channel
          direction : T.Direction,
          floor : T.Floor,
          movement : T.Movement,
          move : T.Direction,
          halt, move_ack, halt_ack : Unit
      end,
    V : class
        variable
          direction : T.Direction,
          movement : T.Movement,
          floor : T.Floor
      end
```

```
value
  /∗ main ∗/
  motor : Unit → in any out any write any Unit
  motor() ≡
    while true do
      let d′ = CH.move? in
        CH.move_ack ! () ; V.direction := d′ ;
        V.movement := T.moving ; V.floor := T.next_floor(d′, V.floor) end
      ▯
      CH.halt? ; CH.halt_ack ! () ; V.movement := T.halted
      ▯
      CH.direction ! V.direction
      ▯
      CH.movement ! V.movement
      ▯
      CH.floor ! V.floor
    end,
```

```
  /∗ initial ∗/
  init : Unit → in any out any write any Unit
  init() ≡ motor(),

  /∗ generators ∗/
  /∗ assumes move only called when
     next floor in direction exists ∗/
  move : T.Direction → in any out any Unit
  move(d) ≡ CH.move ! d ; CH.move_ack?,

  halt : Unit → in any out any Unit
  halt() ≡ CH.halt ! () ; CH.halt_ack?,
```

```
  /∗ observers ∗/
  direction : Unit → in any out any T.Direction
  direction() ≡ CH.direction?,

  floor : Unit → in any out any T.Floor
  floor() ≡ CH.floor?,

  movement : Unit → in any out any T.Movement
  movement() ≡ CH.movement?
end
```

```
scheme C_DOORS1 = hide DS in class
  object DS[ f : T.Floor ] : C_DOOR1
  value
    init : Unit → in any out any write any Unit
    init() ≡ ∥ { DS[ f ].init() | f : T.Floor },

    open : T.Floor → in any out any Unit
    open(f) ≡ DS[ f ].open(),

    close : T.Floor → in any out any Unit
    close(f) ≡ DS[ f ].close(),

    door_state : T.Floor → in any out any T.Door_state
    door_state(f) ≡ DS[ f ].door_state()
end
```

```
scheme C_DOOR1 = hide CH, door_var, door in class
  object CH : class
        channel
          open, close, open_ack, close_ack : Unit,
          door_state : T.Door_state
      end
  variable door_var : T.Door_state
  value
    door : Unit → in any out any write any Unit
    door() ≡
      while true do
        CH.open? ; CH.open_ack ! () ; door_var := T.open
        ▯
        CH.close? ; CH.close_ack ! () ; door_var := T.shut
        ▯
        CH.door_state ! door_var
      end,
```

```
    /∗ initial ∗/
    init : Unit → in any out any write any Unit
    init() ≡ door(),

    /∗ generators ∗/
    close : Unit → in any out any Unit
    close() ≡ CH.close ! () ; CH.close_ack?,

    open : Unit → in any out any Unit
    open() ≡ CH.open ! () ; CH.open_ack?,

    /∗ observer ∗/
    door_state : Unit → in any out any T.Door_state
    door_state() ≡ CH.door_state?
end
```

C_BUTTONS1 and C_BUTTON1 are very similar to C_DOORS1 and C_DOOR1 (but C_BUTTONS1 will contain three object arrays).

Validation and verification

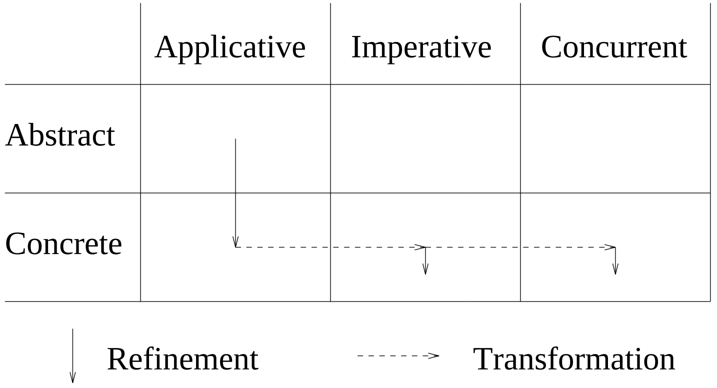
Validation :

- Have we taken any more requirements into account?
- If so, are they satisfied?

Verification :

Idea of method is that this is not done; we have no abstract concurrent version for which to show implementation. Instead we argue for “correctness by construction”.

Typical Development



	Applicative	Imperative	Concurrent
Abstract			
Concrete			

↓ Refinement ⇢ Transformation

Completion

- translation
- unit testing
- installation
- testing

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]
$\forall$ buff, buff$'$: Buffer, l : T.Message*, m : T.Message $\bullet$
buffered(buff, l) $\wedge$ put(m, buff) $=$ buff$'$ $\Rightarrow$ buffered(buff$'$, l $^\frown$ $\langle$m$\rangle$),

[buffered_get]
$\forall$ buff, buff$'$: Buffer, l : T.Message*, m1, m2 : T.Message $\bullet$
buffered(buff, l) $\wedge$ can_get(buff) $\wedge$ get(buff) $=$ (m1, buff$'$) $\Rightarrow$
($\exists$ l1, l2 : T.Message* $\bullet$
l $=$ l1 $^\frown$ $\langle$m1$\rangle$ $^\frown$ l2 $\wedge$ buffered(buff$'$, l1 $^\frown$ l2) $\wedge$
(m2 $\in$ **elems** l1 $\Rightarrow$ $\sim$ T.leq(m1, m2))),

[no_loss_or_gain]
$\forall$ buff : Buffer, l1, l2 : T.Message* $\bullet$
buffered(buff, l1) $\wedge$ buffered(buff, l2) $\Rightarrow$ permutation(l1, l2)

end

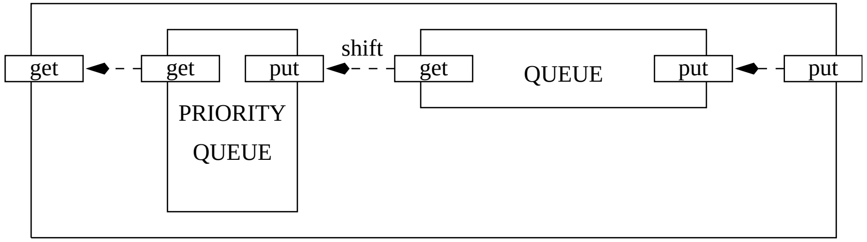
buffered is a relation between the abstract state *Buffer* and the list of messages input but not yet extracted.

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?

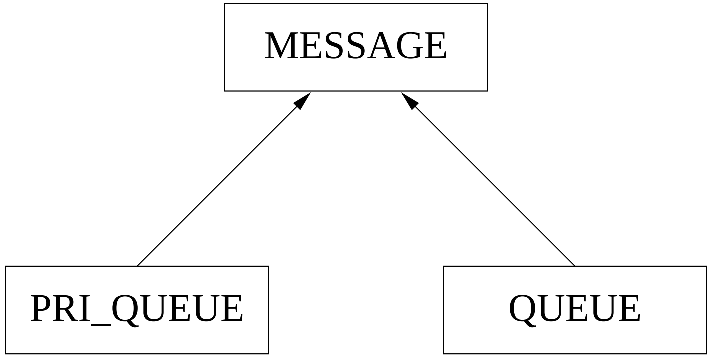
shift is not “invisible”: can change *can_get*, for example.

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.

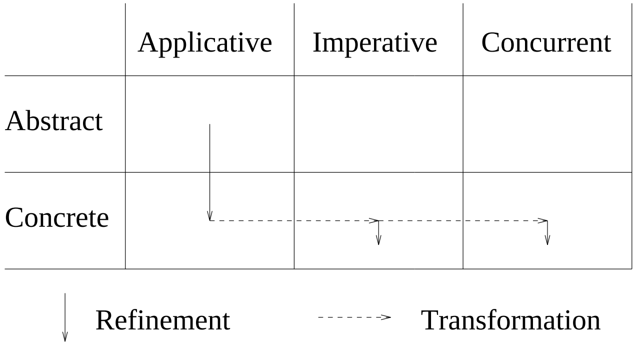
Module dependencies


MESSAGE

PRI_QUEUE

QUEUE

Typical Development



	Applicative	Imperative	Concurrent
Abstract			
Concrete			

↓ Refinement

- - - -> Transformation

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

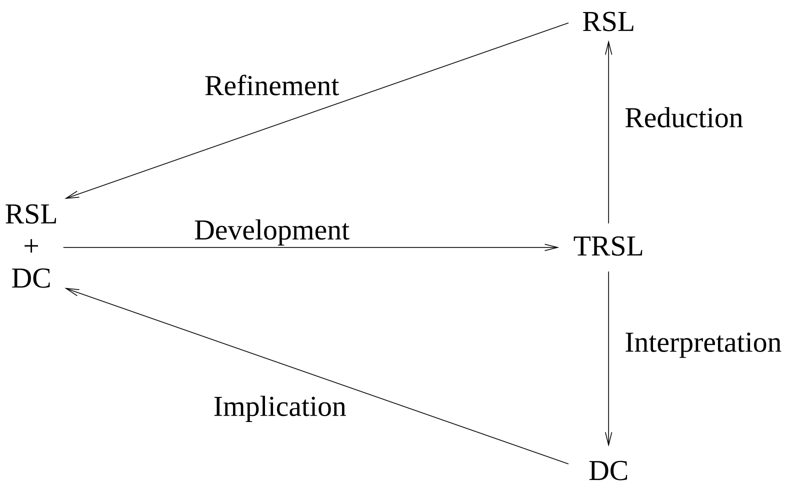
- States of leaf modules are independent: no interference.
- Interface functions of different leaf modules may be called sequentially or concurrently.
- Freedom from deadlock easy to check:
  - all channels hidden
  - all servers started by top-level *init*
  - interface functions and servers match.
- 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

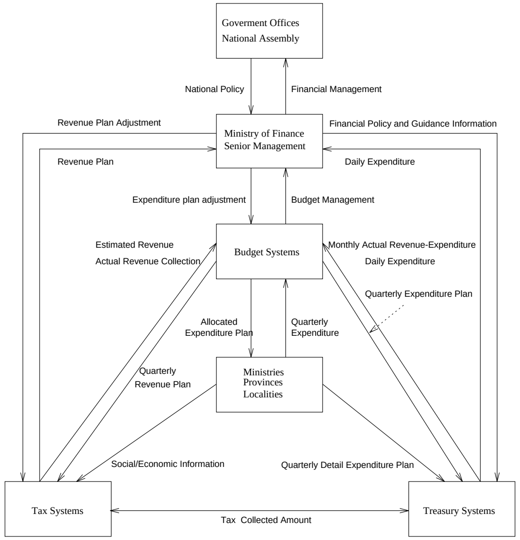
1. We need a national financial information system to collect reliable data, make budgets, assess the affects of possible changes, etc.
2. A small local office will need 8 PCs, ...

Aim of the specification

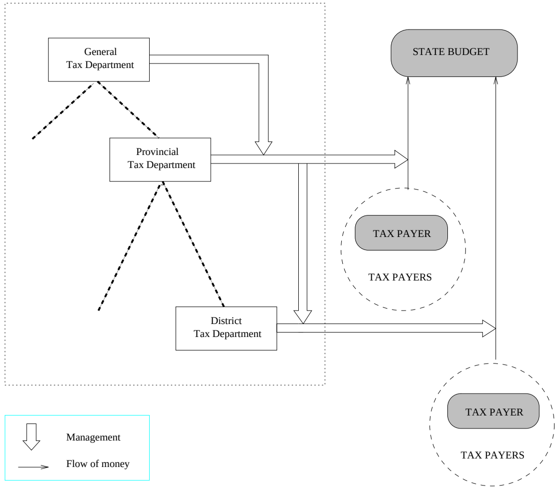
- Act as a high level design: allow design decisions to be explored.
- Define responsibilities for data storage, collection, analysis, reporting etc.
- and so provide detailed requirements for component offices.

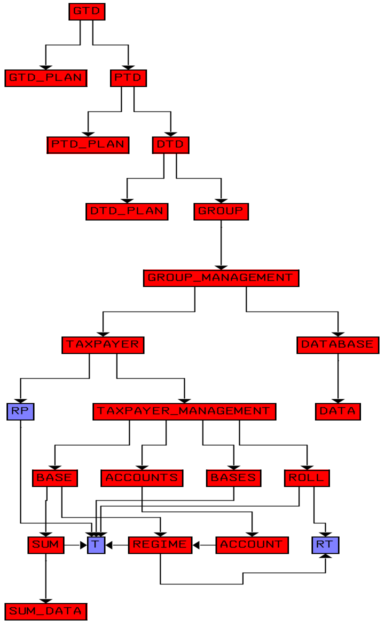
Not intended to be directly implemented! But provides a framework for gradual computerisation.

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) ≡ **true**

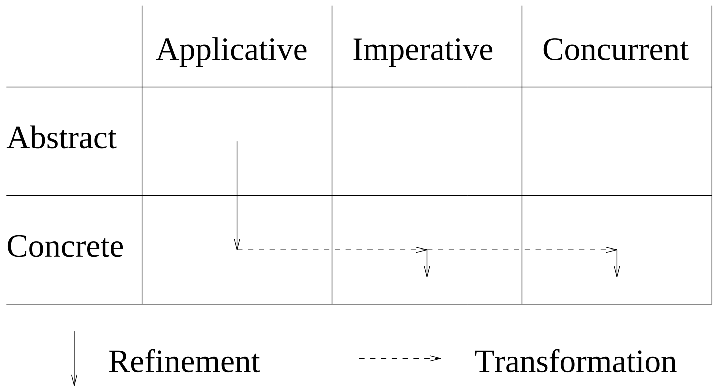
- Imperative sequential

  empty() ; is_empty() ≡ empty() ; **true**

- Imperative concurrent

  ∀ test : **Bool** $\overset{\sim}{\to}$ **Unit** •
  (main() ╫ empty()) ╫ test(is_empty()) ≡
  (main() ╫ empty()) ╫ test(**true**)

Ideal development route



	Applicative	Imperative	Concurrent
Abstract			
Concrete			

Refinement

Transformation

Transformation theorem

$$\begin{tikzcd}
\mathbf{A0} \arrow[r] \arrow[d] & I0 + D \arrow[d] \\
\mathbf{A1} \arrow[r, dashed] & \mathbf{I1}
\end{tikzcd}$$

$\longrightarrow$ Refinement

$\dashrightarrow$ Transformation

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

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

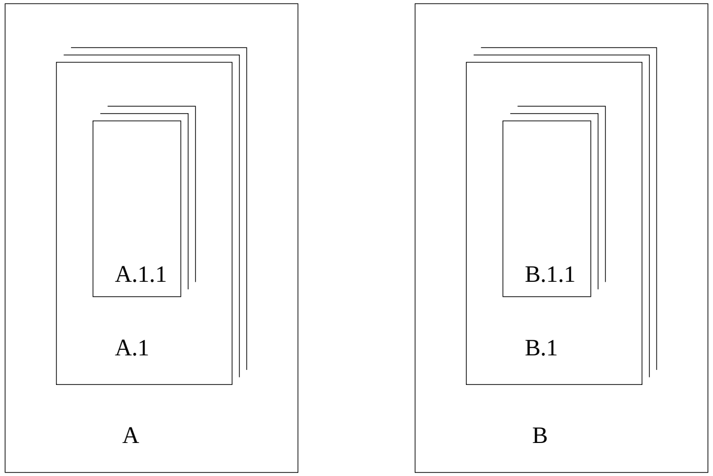
Transformation properties

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

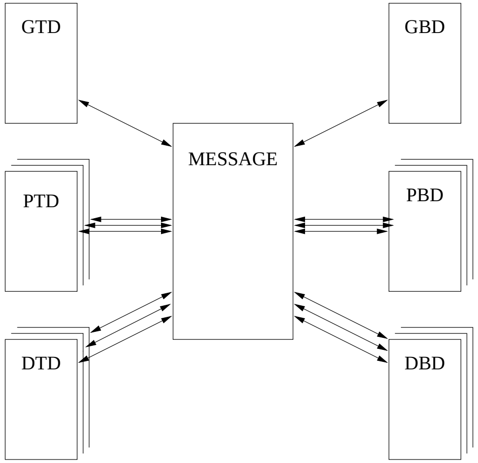
Consequences

- 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.