MICEI EDFS-0405

A Brief Introduction to VDM-SL

Nuno Rodrigues – nfr@di.uminho.pt

Grupo de Lógica e Métodos Formais
Dept. Informática, Universidade do Minho
Braga, Portugal

Tel.: +351.253.60 44 44 ; Fax.: +351.253.600 44 71; E-Mail: nfr@di.uminho.pt ;
URL: http://wiki.di.uminho.pt/twiki/bin/view/Nuno

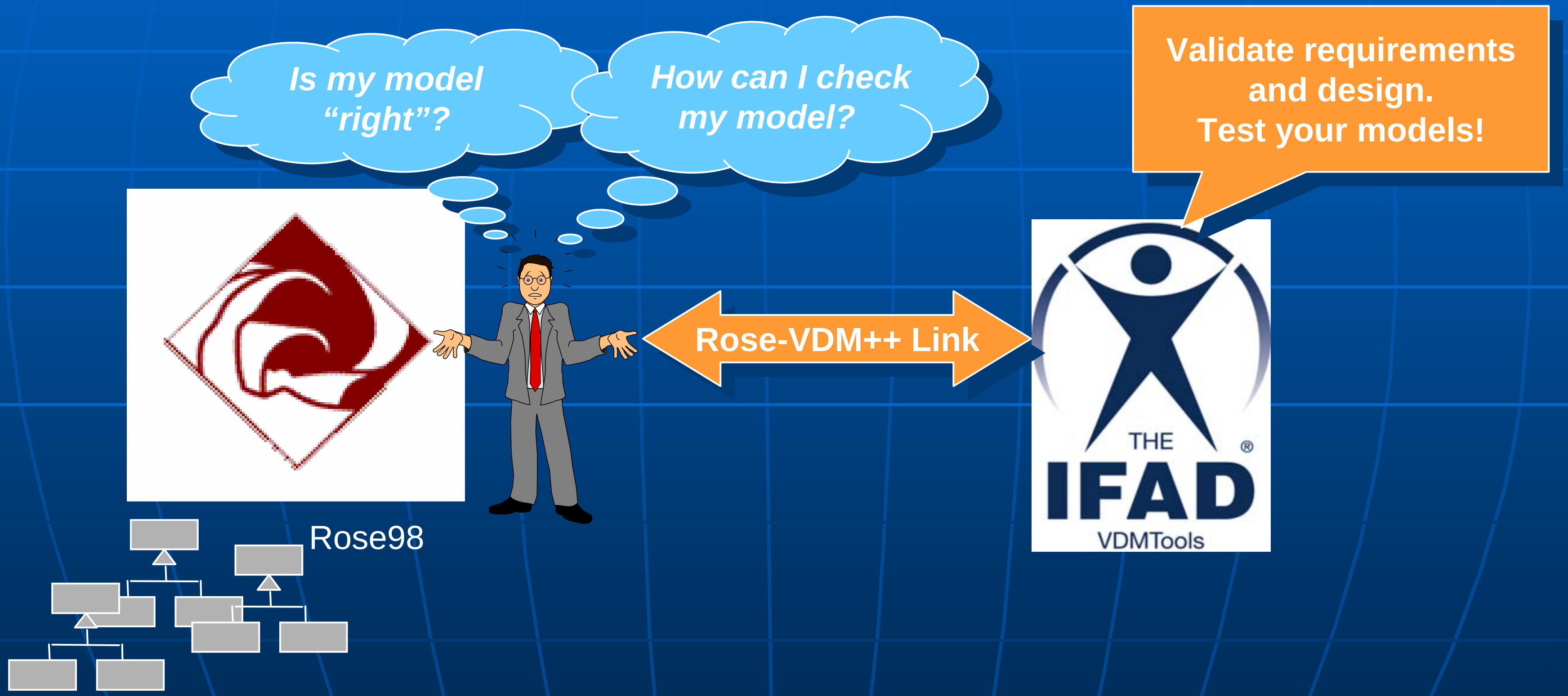
Introduction to VDM-SL

- Overview of VDMTools®
- Formal Development with VDMTools®
  - Types
  - Functions
  - Expressions

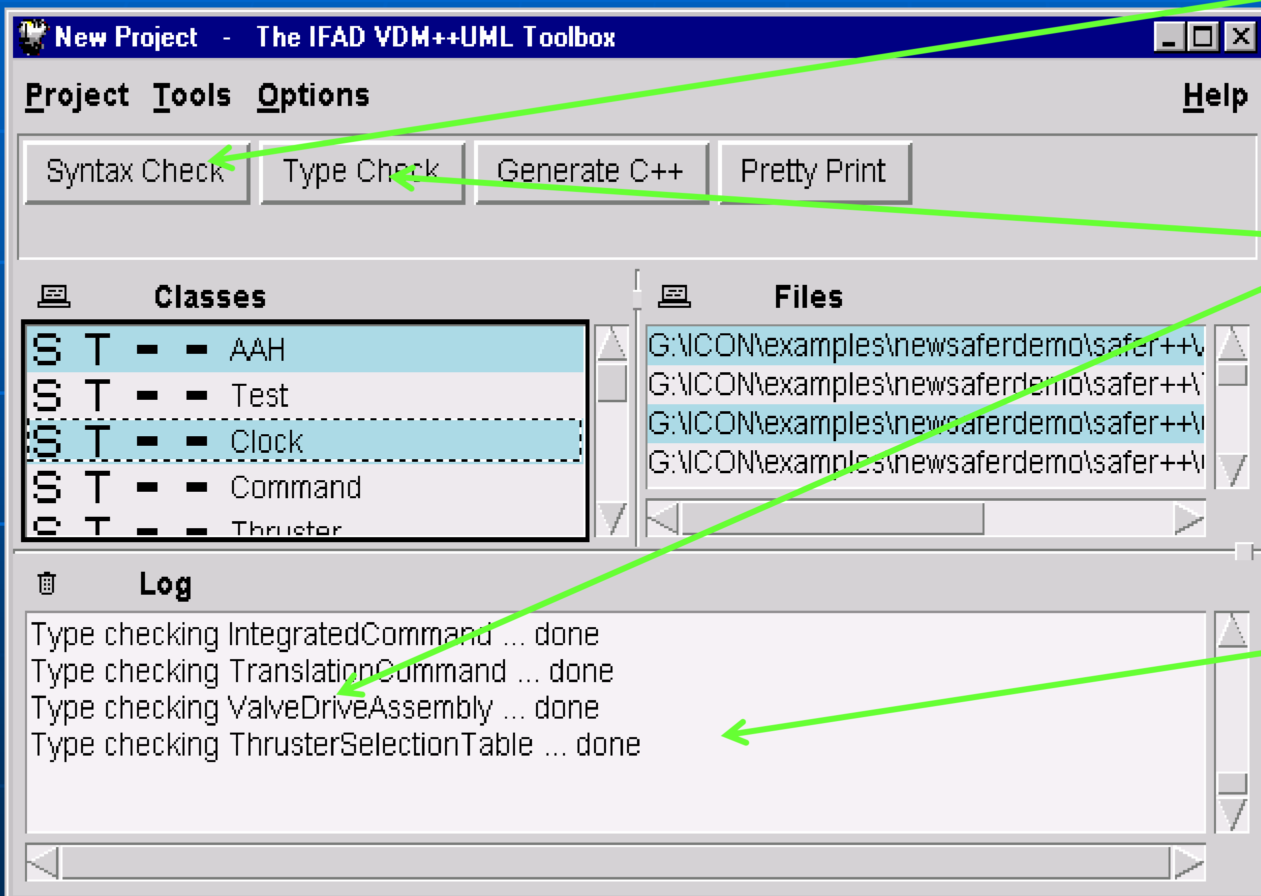
VDMTools® Overview...

- Syntax & Type Checker
- The Rose-VDM++ Link
- Interpreter (Debugger)
- THE IFAD® VDMTools
- Document Generator
- API (Corba), DL Facility
- C++/Java Code Generator

The Rose-VDM++ Link



Syntax & Type checking

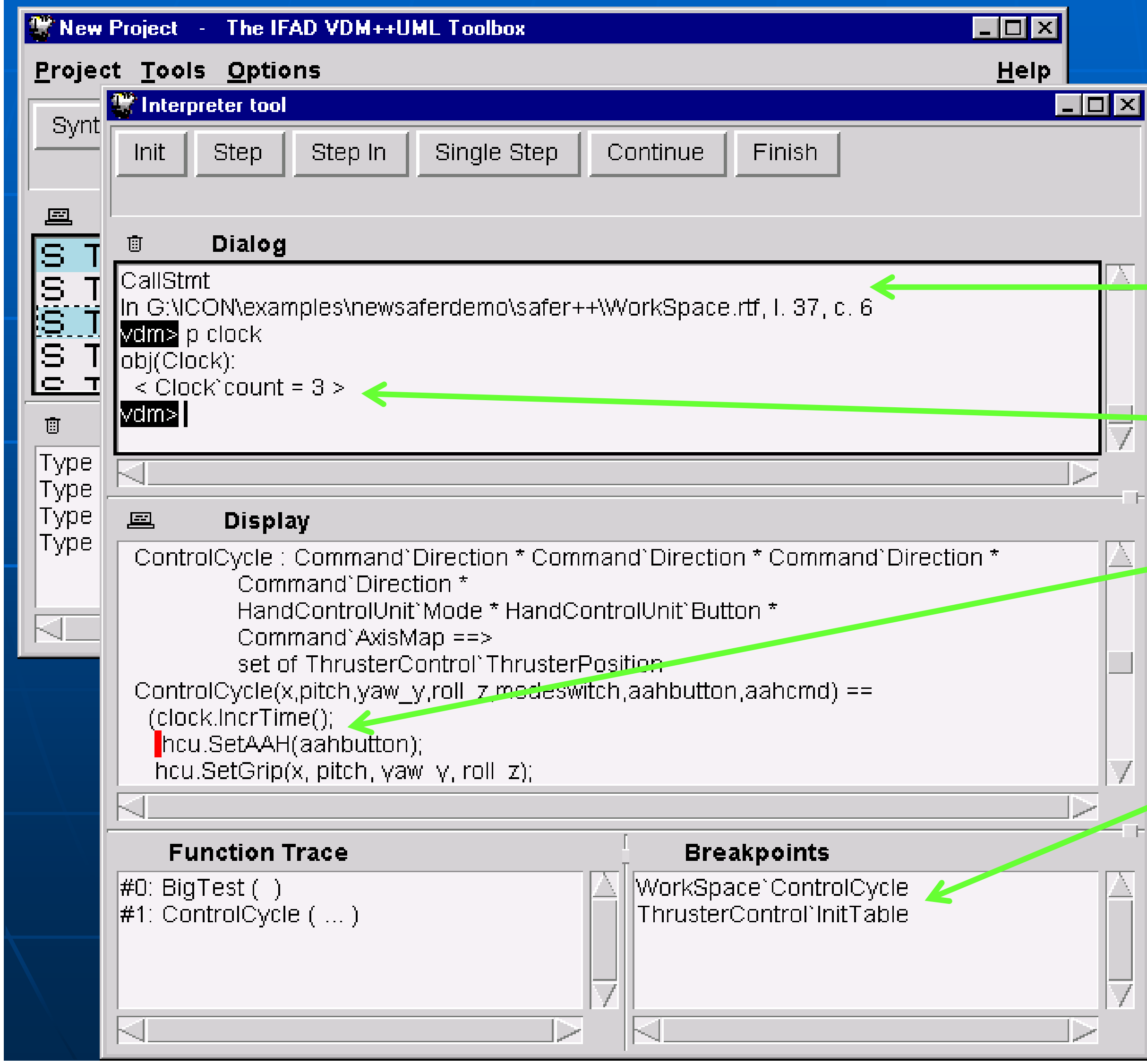


Syntax checking

Type checking

Log messages

Debugging with VDMTools®



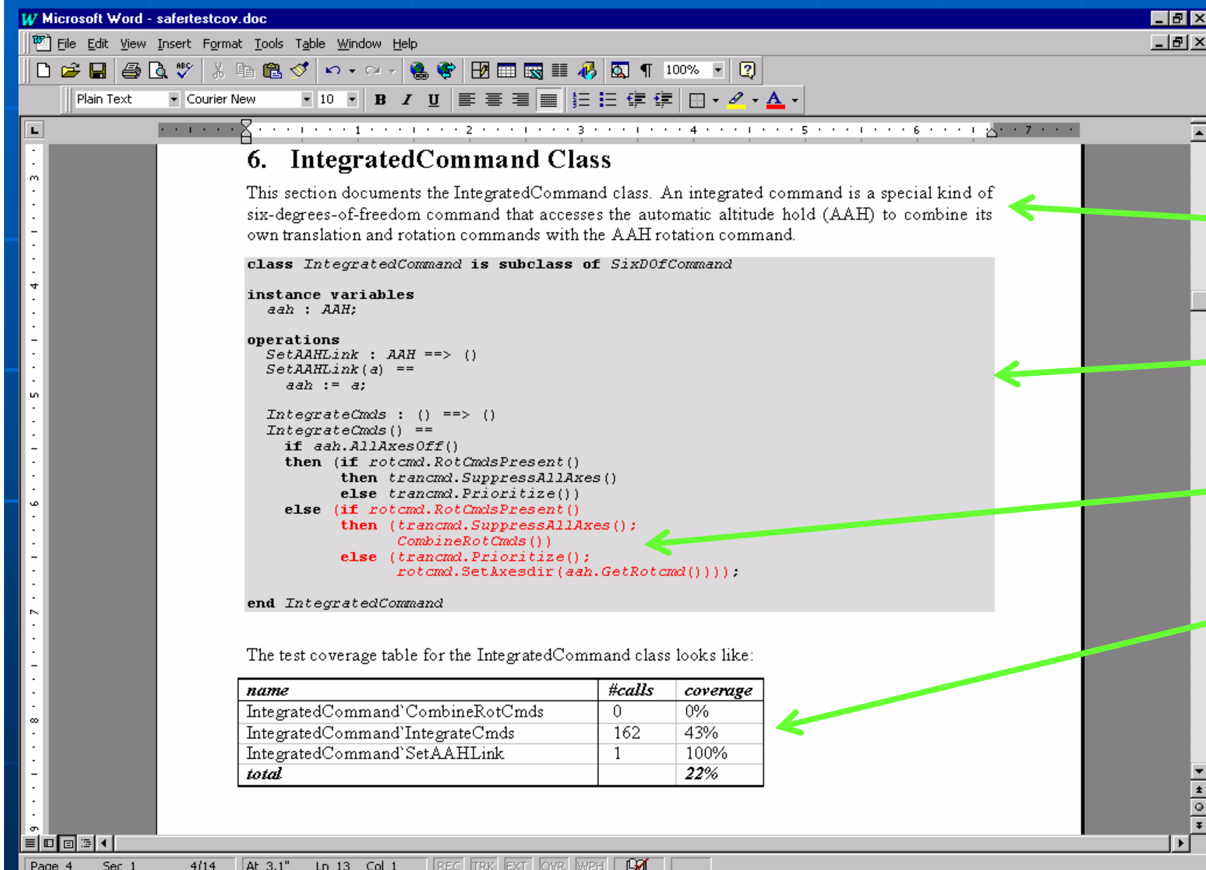
Execution

Value inspection

Single stepping

Breakpoints

Documentation in MS Word


Microsoft Word - safertestcov.doc

6. IntegratedCommand Class

This section documents the IntegratedCommand class. An integrated command is a special kind of six-degrees-of-freedom command that accesses the automatic altitude hold (AAH) to combine its own translation and rotation commands with the AAH rotation command.

```
class IntegratedCommand is subclass of SixDOfCommand

instance variables
  aah : AAH;

operations
  SetAAHLink : AAH ==> ()
  SetAAHLink(a) ==
    aah := a;

  IntegrateCmds : () ==> ()
  IntegrateCmds() ==
    if aah.AllAxesOff()
    then (if rotcmd.RotCmdsPresent()
          then trancmd.SuppressAllAxes()
          else trancmd.Prioritize())
    else (if rotcmd.RotCmdsPresent()
          then (trancmd.SuppressAllAxes();
                CombineRotCmds())
          else (trancmd.Prioritize();
                rotcmd.SetAxesdir(aah.GetRotcmd())));

end IntegratedCommand
```

The test coverage table for the IntegratedCommand class looks like:

name	*#calls*	*coverage*
IntegratedCommand`CombineRotCmds	0	0%
IntegratedCommand`IntegrateCmds	162	43%
IntegratedCommand`SetAAHLink	1	100%
total		*22%*

One compound document:

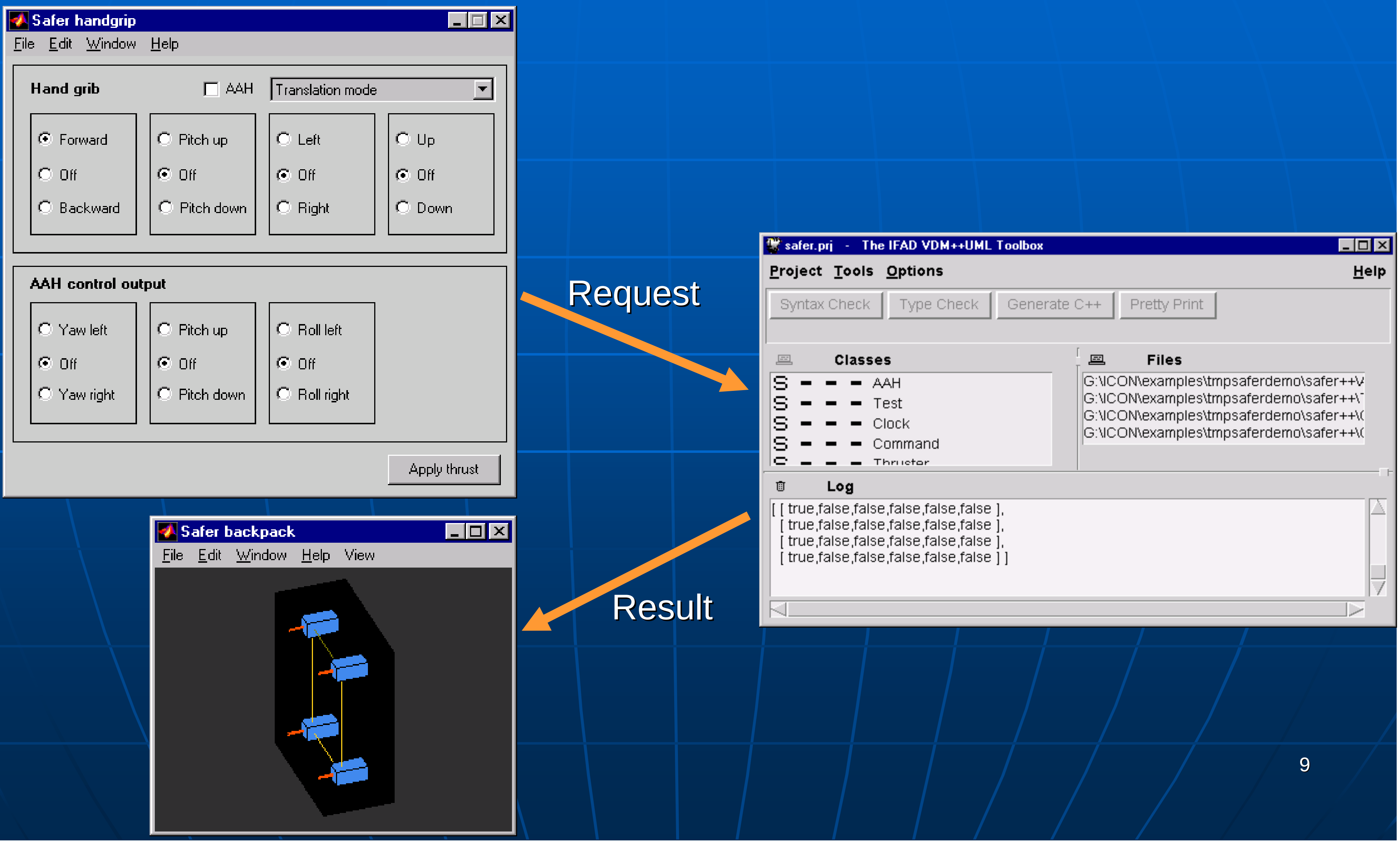
- Documentation
- Specification
- Test coverage
- Test coverage statistics

C++ Code Generator

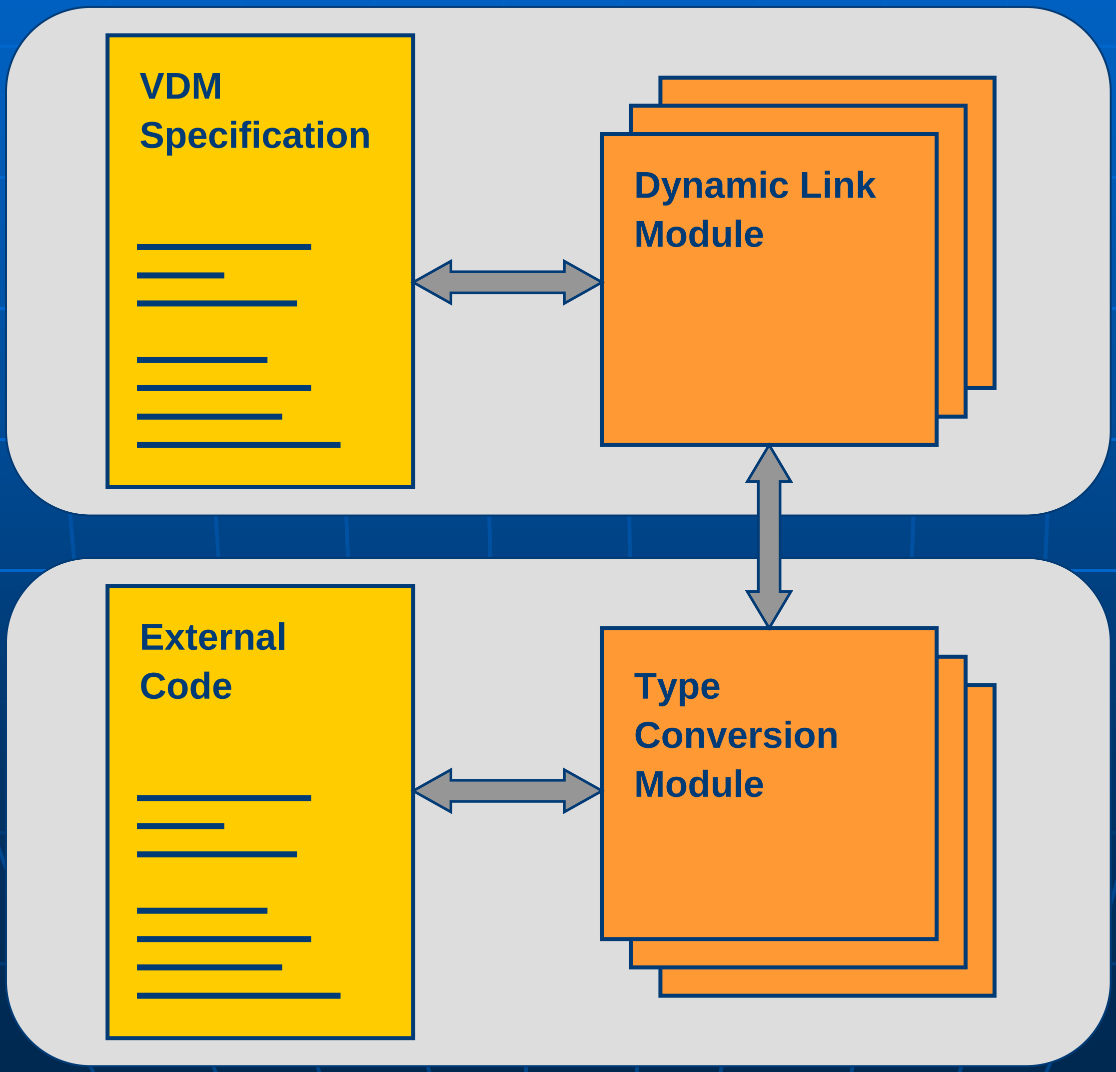
- Platforms and Compilers
  - GNU egcs version 1.1:
    - Sun SPARC SunOS running 4.1.x or Solaris 2.6
    - HP9000/700 running HP-UX 10
    - PC's running Linux
  - Visual C++ version 5.0 or higher:
    - Windows NT
    - Windows 95/98

```
(...)

Int vdm_Conta::vdm_Levantamento
(const TYPE_Conta_String &vdm_dt,

const Int &vdm_lev) {
 if (((Bool)
((vdm_saldo.GetValue()) >=
(vdm_lev.GetValue()))).GetValue()
) {
    vdm_saldo = vdm_saldo -
vdm_lev;

vdm_movimentos.ImpModify(vdm_dt,
-vdm_lev);
    return (Generic) vdm_saldo;
 }
 else
   return (Generic) (-(Int) 1);
}

(...)
```

Toolbox API


Safer handgrip
File Edit Window Help
Hand grib
AAH
Translation mode
Forward
Off
Backward
Pitch up
Off
Pitch down
Left
Off
Right
Up
Off
Down
AAH control output
Yaw left
Off
Yaw right
Pitch up
Off
Pitch down
Roll left
Off
Roll right
Apply thrust
Request
Result
Safer backpack
File Edit Window Help View
safer.prj - The IFAD VDM++UML Toolbox
Project Tools Options
Help
Syntax Check
Type Check
Generate C++
Pretty Print
Classes
AAH
Test
Clock
Command
Files
G:\ICON\examples\tmpsaferdemo\safer++\
Log
[[true,false,false,false,false,false],
[true,false,false,false,false,false],
[true,false,false,false,false,false],
[true,false,false,false,false,false]]

Dynamic Link Facility



Introduction to VDM-SL

- ✓ Overview of VDMTools®
- ➢ Formal Development with VDMTools®
  - Types
  - Functions
  - Expressions

Mathematical Foundations

Abstract Notation	C/C++	Description

A x B	`struct {`
	`A fst;`
	`B snd;`
	`};`
		Products
		(records)

A + B	`Struct {`
	`int tag; /* 1,2 */`
	`union {`
	`A ifA;`
	`B ifB;`
	`} data;`
	`};`
		Coproducts
		(variant records)

B^A	`B ... [A];`	Exponentials (arrays)
1 + A	`A *...;`	Pointers (null alternative)

Introduction to VDM-SL

- ✓ Overview of VDMTools®
- ✓ Formal Development with VDMTools®
  - ➢ Types
  - • Functions
  - • Expressions

Type Definitions

- Basic Data Types
  - Boolean
  - Numeric
  - Tokens
  - Characters
  - Quotations

- Derived Data Types
  - Set types
  - Sequence types
  - Map types

- Primitive Data Types
  - Product (record) types
  - Coproduct (union) types
  - Function types
  - Optional types

Data Type invariants must also be added!

Basic Data Types

▪ Boolean:	`bool`

▪ Numeric:	`real`
	`rat`
	`int`
	`nat`
	`nat1`

▪ Tokens:	`token`
▪ Characters:	`char`
▪ Quotations:	<RED> (e.g.)

Primitive Data Types

- Product (record) types:

```
/* cartesian product */  T = A * B * … * Z
/* labeled tuples */     T :: a : A
                              b :
                              …
                              z : Z
```

- Coproduct (union) types:

```
/* disjoint union */     T = A | B | … | Z
```

- Function types:

```
/* partial functions */  T = A -> B
/* total functions */    T = A +> B
```

- Optional types:

```
/* pointers */           T = [A]
/* T | nil */
```

Derived Data Types

- Set types:

```
/* sets */                        T = set of A
```

- Sequence types:

```
/* sequences */                   T = seq of A
/* non-empty sequences */         T = seq1 of A
```

- Map types:

```
/* general maps */                T = map A to B
/* injective maps */              T = inmap A to B
```

Data Type Operators

- The different data types (basic, primitive and derived) have operations specific to those types:
  - `not x`
  - `x < y`
  - `s1 union s2`
  - `head l`
  - `dom f`
  - Etc.
- For each such type we'll list these operations

Boolean Data Type Operators

`not b`	Negation	`bool -> bool`
`a and b`	Conjunction	`bool * bool -> bool`
`a or b`	Disjunction	`bool * bool -> bool`
`a => b`	Implication	`bool * bool -> bool`
`a <=> b`	Biimplication	`bool * bool -> bool`
`a = b`	Equality	`bool * bool -> bool`
`a <> b`	Inequality	`bool * bool -> bool`

Numeric Data Types Operators

`-x`	Unary minus	`real -> real`
`abs x`	Absolute value	`real -> real`
`floor x`	Floor	`real -> int`
`x + y`	Sum	`real * real -> real`
`x – y`	Difference	`real * real -> real`
`x * y`	Product	`real * real -> real`
`x / y`	Division	`real * real -> real`
`x ** y`	Power	`real * real -> real`
`x < y`	Less than	`real * real -> bool`
`x > y`	Greater than	`real * real -> bool`
`x <= y`	Less or equal	`real * real -> bool`
`x >= y`	Greater or equal	`real * real -> bool`
`x = y`	Equal	`real * real -> bool`
`x <> y`	Not equal	`real * real -> bool`
`x div y`	Integer division	`int * int -> int`
`x rem y`	Remainder	`int * int -> int`
`x mod y`	Modulus	`int * int -> int`

Product Data Type Operators

- Product type definition:

```
A1 * A2 * … * An
```

  Construction of a tuple:

```
mk_(a1,a2,…,an)
```

- Record type definition:

```
A :: fst : A1
     snd : A2
       …
     nth : An
```

  Construction of a record:

```
mk_A(a1,a2,...,an)
```

Set Operators

e in set s1	Membership	A * set of A -> bool
e not in set s1	Not membership	A * set of A -> bool
s1 union s2	Union	set of A * set of A -> set of A
s1 inter s2	Intersection	set of A * set of A -> set of A
s1 \ s2	Difference	set of A * set of A -> set of A
s1 subset s2	Subset	set of A * set of A -> bool
s1 psubset s2	Proper subset	set of A * set of A -> bool
s1 = s2	Equality	set of A * set of A -> bool
s1 <> s2	Inequality	set of A * set of A -> bool
card s1	Cardinality	set of A -> nat
dunion s1	Distr. union	set of set of A -> set of A
dinter s1	Distr. intersection	set of set of A -> set of A
power s1	Finite power set	set of A -> set of set of A

Sequence Operators

`hd l`	Head	`seq1 of A -> A`
`tl l`	Tail	`seq1 of A -> seq of A`
`len l`	Length	`seq of A -> nat`
`elems l`	Elements	`seq of A -> set of A`
`inds l`	Indexes	`seq of A -> set of nat1`
`l1 ^ l2`	Concatenation	`seq of A * seq of A -> seq of A`
`conc ll`	Distr. conc.	`seq of seq of A -> seq of A`
`l(i)`	Seq. application	`seq1 of A * nat1 -> A`
`l ++ m`	Seq. modification	`seq of A * map nat1 to A -> seq of A`
`l1 = l2`	Equality	`seq of A * seq of A -> bool`
`l1 <> l2`	Inequality	`seq of A * seq of A -> bool`

Map Operators

`dom m`	Domain	`(map A to B) -> set of A`
`rng m`	Range	`(map A to B) -> set of B`
`m1 munion m2`	Merge	`(map A to B) * (map A to B) -> map A to B`
`m1 ++ m2`	Override	`(map A to B) * (map A to B) -> map A to B`
`merge ms`	Distr. merge	`set of (map A to B) -> map A to B`
`s <: m`	Dom. restr. to	`set of A * (map A to B) -> map A to B`
`s <-: m`	Dom. restr. by	`set of A * (map A to B) -> map A to B`
`m :> s`	Rng. restr. to	`(map A to B) * set of A -> map A to B`
`m :-> s`	Rng. restr. by	`(map A to B) * set of A -> map A to B`
`m(d)`	Map apply	`(map A to B) * A -> B`
`inverse m`	Map inverse	`inmap A to B -> inmap B to A`
`m1 = m2`	Equality	`(map A to B) * (map A to B) -> bool`
`m1 <> m2`	Inequality	`(map A to B) * (map A to B) -> bool`

Comprehension Notation

- Convenient comprehensions exist for sets, sequences and maps:
  - Set-comprehension

    ```
    { elem | bind-list & pred }
    ```

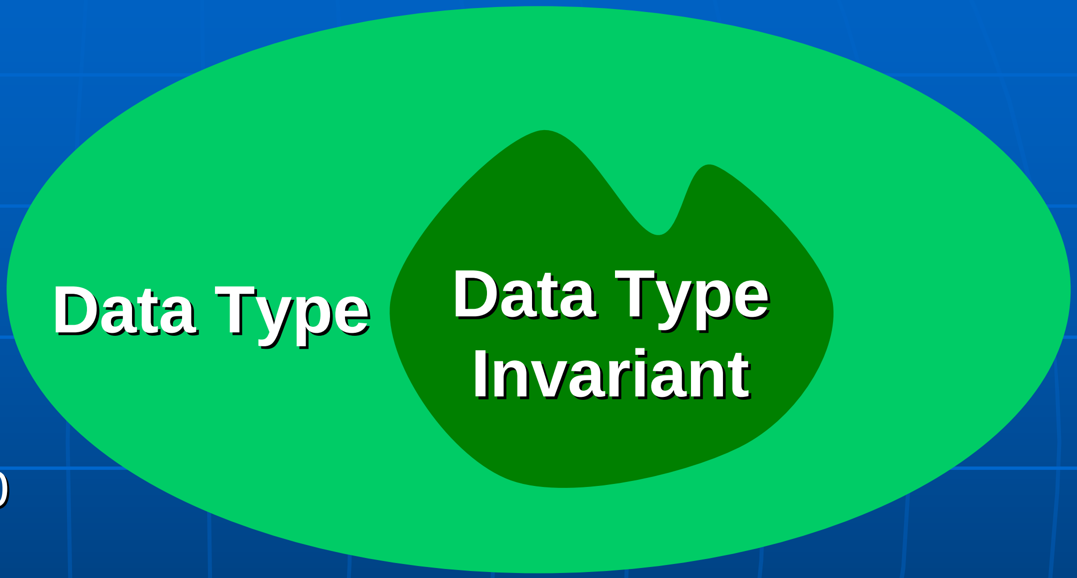
  - Sequence-comprehension

    ```
    [ elem | setbind & pred ]
    ```

    The set binding is restricted to sets of numeric values, which are used to find the order of the resulting sequence
  - Map-comprehension

    ```
    { maplet | bind-list & pred }
    ```

Data Type Invariants




```
Even = nat
inv n == n mod 2 = 0

SpecialPair = nat * real
inv mk_(n,r) == n < r

DisjointSets = set of set of  A
inv ss == forall s1, s2 in set ss &
             s1 <> s2 => s1 inter s2 = {}
```

Introduction to VDM-SL

- ✓ Overview of VDMTools®
- ✓ Formal Development with VDMTools®
  - ✓ Types
  - ➢ Functions
  - • Expressions

Function Definitions

- Explicit Functions:

```
f: A * B * … * Z -> R
f(a,b,…,z) == expression
[pre pre-expression]
[post post-expression]
```

- Implicit Functions:

```
f(a:A, b:B, …, z:Z) r:R
[pre pre-expression]
post post-expression
```

Implicit functions cannot be executed by the VDM interpreter.

Function Examples

- Explicit Function:

```
mapInter: (map nat to nat) * (map nat to nat) -> map nat to nat
mapInter(m1,m2) == (dom m1 inter dom m2) <: m1
pre forall d in set (dom m1 inter dom m2) & m1(d) = m2(d);
```

- Implicit Function:

```
mapInter: (m1,m2: map nat to nat) m: map nat to nat
pre forall d in set (dom m1 inter dom m2) & m1(d) = m2(d)
post dom m = (dom m1 inter dom m2) and
     forall d in set dom m & m(d) = m1(d);
```

Polymorphic Functions

- Generic functions that can be used on values of several different types

```
emptyBag[@elem]: () +> (map @elem to nat1)
emptyBag() == { |-> };

numBag[@elem] : @elem * (map @elem to nat1)
+> nat
numBag(e,m) == if e in set dom m then m(e)
else 0;
```

- Type instantiation

```
emptyInt = emptyBag[int]
numInt = numBag[int]
```

High Order Functions

- Functions that receive other functions as arguments

```
natFilter : (nat -> bool) * seq of nat -> seq of nat
natFilter(p,l) == [ l(i) | i in set inds l & p(l(i)) ];

filter[@elem] : (@elem -> bool) * seq of @elem -> seq of
@elem
filter(p,s) == [ l(i) | i in set inds l & p(l(i)) ];
```

- Example

```
f : nat -> bool
f n == n mod 2 = 0
natFilter (f, [1,2,3,4,5]) = ?
```

Introduction to VDM-SL

- ✓ Overview of VDMTools®
- ✓ Formal Development with VDMTools®
  - ✓ Types
  - ✓ Functions
  - ➢ Expressions

Expressions

- Let-in and Let-be expressions
- If-then-else expressions
- Cases expressions
- Quantified expressions
- Set expressions
- Sequence expressions
- Map expressions
- Tuple expressions
- Record expressions
- Is expressions
- Lambda expressions

Let-in and Let-be Expressions

Let-in expressions are used for naming complicated constructs and for decomposing complex structures

```
let cs' = {c |-> cs(c) union {s}},
    ct' = {s |-> ct(s) union {c}}
in … cs' … ct' …
```

Let-be-such-that expressions are even more powerful. A free choice can be expressed:

```
let i in set inds l be st Largest (elems l, l(i))
in … i …
```

and

```
let l in set Permutations(list) be st
    forall i,j in set inds l & i < j => l(i) < l(j)
in … l …
```

If-Then-Else Expressions

- If-Then-Else expressions are similar to those known from programming languages:

```
if c in set dom rq then rq(c)
                   else {}
```

- Nested If-Then-Else expressions are also available:

```
if i = 0 then <Zero>
         elseif 1 <= i and i <= 9
         then <Digit>
         else <Number>
```

Cases Expressions

- Cases expressions are very powerful because of pattern matching:

```
cases com:
    mk_Loan(a,b) -> a^" has borrowed "^b,
    mk_Receive(a,b) -> a^" has returned "^b,
    mk_Status(l) -> l^" are borrowing "^Borrows(l),
    others -> "some other command is used"
end
```

Quantified Expressions

- Quantification can be over sets:

```
forall s1,s2 in set ss &
     s1 <> s2 => s1 inter s2 = {}
```

- Quantification can be over types as well:

```
forall x: int &
     exists1 y: int &
        x ** 2 = y
```

- Quantifications over types cannot be evaluated by the VDM interpreter

Set Expressions

- Set enumeration:
```
{a,3,3,true}
```
- Set comprehension can either use set binding:
```
{a+2 | mk_(a,a) in set
           {mk_(true,1),mk_(1,1)}}
```
  or type binding:
```
{a | a: nat & a<10}
```
- Set range expression:
```
{3,...,10}
```

Sequence Expressions

- Sequence enumeration:
  `[7.7,true,"I",true]`
- Sequence comprehension can only use a set bind with numeric values:
  `[i*i | i in set {1,2,4,6}]`
  and
  `[i | i in set {6,3,2,7} & i mod 2 = 0]`
- Subsequence expression:
  `[4,true,"string",9,4](2,...,4)`

Map Expressions

- Map enumeration:
```
{1|-> true, 7 |-> 6}
```
- Map comprehension can either use type binding:
```
{i|-> mk_(i,true) | i: bool}
```
or set binding:
```
{a+b |-> b-a | a in set {1,2},
                b in set {3,6}}
```
and
```
{i|->i | i in set {1,...,10} &
                   i mod 3 = 0}
```
One must be careful to ensure that every domain element maps uniquely to one range element.

Tuple Expressions

- A tuple expression looks like:
  `mk_(2,7,true,{|->})`
- Remember that tuple values from a tuple type will always
  - have the same length and
  - use the same types (possible union types) at corresponding positions.
- On the other hand the length of a sequence value may vary but the elements of the sequence will always be of the same type.

Record Expression

Given two type definitions like:

```
A :: n: nat
     b: bool
     s: set of nat;
B :: n: nat
     r: real
```

one can write expressions like:

```
mk_A(1,true,{8})
mk_B(3,3)
mu (mk_A(7,false,{1,4}), n|->1, s|->{})
mu (mk_B(3,4), r|->5.5)
```

The `mu` operator is called “the record modifier”.

Apply Expressions

- Map applications

```
let m = {2|->1, 1|->2}
in m(1)
```

- Sequence applications

```
[2,7,3](2)
```

- Field Select applications

```
let r = mk_A(2,false,{6,9})
in r.b
```

Is Expressions

- Basic values and record values can be tested by is- expressions:

  - `is_nat(5)`
  - `is_C(mk_C(5))`
  - `is_A(mk_B(3,7))`
  - `is_A(6)`

Lambda Expressions

- Lambda expressions are an alternative way of defining explicit functions:

```
lambda n: nat & n * n
```

- They can take a type bind list:

```
lambda a: nat, b: bool &
 if b then a else 0
```

- Or use more complex types:

```
lambda mk_(a,b): nat * nat & a + b
```

Exercises

```
npush : int * seq of int -> seq of int
npush(n, s) == [n] ^ [ s(i) | i in set inds s
                                & n <> s(i) ] ;
```

```
npush10 : int * seq of int -> seq of int
npush10(n, s) == [n] ^
      [ s(i) | i in set inds s &
            n <> s(i) and
            if n <> s(i) then i <= 9 else i <= 10 ] ;
```

Chemical Plant Alarm System

A chemical plant is equipped with a number of sensors which are able to raise alarms in response to conditions in the plant. When an alarm is raised, an expert must be called to the scene. Experts have different qualifications for coping with different kinds of alarm.

CPAS Requirements

- R1 – A computer-based system is to be developed to manage the alarms of this plant
- R2 – Four kinds of qualification are needed to cope with the alarms. These are electrical, mechanical, biological, and chemical.
- R3 – There must be experts on duty all periods which have been allocated in the system.
- R4 – Each expert can have a list of qualifications
- R5 – Each alarm reported to the system has a qualification associated with it along with a description of the alarm which can be understood by the expert
- R6 – Whenever an alarm is received by the system an expert with the right qualification should be found so that he or she can be paged.
- R7 – The experts should be able to use the system database to check when they will be on duty
- R8 – It must be possible to assess the number of experts on duty