A Short Introduction to Formal Methods

Manuel Carro
mcarro@fi.upm.es
Technical University of Madrid (Spain)

Formal Methods: General Issues

Rigorous Development

- Rigorous development aims at developing analysis, designs, programs, components and proving interesting properties thereof
- Several approaches at several levels
- Will delve into the so-called *Formal Method* approach
- **Formal methods** in fact encompasses several techniques, tools, specification languages, proof theories, . . .
- We will use however a well-known approach (VDM, the *Vienna Development Method*) to highlight several points
- Plan:
  1. General considerations on formal methods
  2. The VDM approach: syntax, semantics, tools, examples
  3. Other approaches

Formal Methods: Pointers

Some of them used to prepare this set of slides:

A Specifier's Introduction to Formal Methods J. M. Wing, Carnegie Mellon University, IEEE Computer, September 1990

Seven Myths of Formal Methods Anthony Hall, Praxis Systems, IEEE Computer, September 1990

Systematic Software Development Using VDM Cliff B. Jones, Prentice-Hall, 1986

Formal Specification of Software John Fitzgerald, Center for Software Reliability

A Guide to Reading VDM Specifications Bob Fields University of Manchester

Programs from Specifications A. Herranz, J. J. Moreno, June 1999 (talk given at the Institut für Wirtschaftsinformatik, Universität Münster)

Formal Specifications: a Roadmap Axel van Lamsweerde, Université Catholique de Louvain

Understanding the differences between VDM and Z, I. J. Hayes, C. B. Jones and J. E. Nicholls, University of Manchester

Modeling Systems: Practical Tools and Techniques in Software Development Fitzgerald & Larsen,Cambridge University Press, 1998

Formal Methods

- Mathematically based techniques for describing system properties (in a **very** broad sense)
- Turing (late 1940s): annotation of programs makes reasoning with them easier
- Mathematical basis usually given by a formal specification language
- However, formal methods usually include:
  - Indications of fields where it can be applied
  - Guidelines to be successfully used
  - Sometimes, associated tools
- **Tools do not necessarily exist**: a FM is a FM, and not a computer language (compare with maths or physics)
- However, associated computer languages often exist
- Specification language always present

An Example of a FM

- Backus-Naur form for grammars is a specification language

  A := aBb | λ
  B := AA
- Any reasoning over a *schema* of a grammar is valid for any grammar represented by the scheme
- Formal method associated include equations over strings and automata
- Domain of application clearly delimited (module translation of other problems into strings)
- This is usual in FM: normally domain-oriented

What is Formal Specification

The expression in some formal language and at some level of abstraction of a collection of properties some system should satisfy

- *Properties* denote a wide variety of targets:
  - Functional requirements
  - Non-functional requirements (complexity, timing, ...)
  - Services provided by components
  - Protocols of interaction among such components
  - ...
- A formal specification include:
  - Rules to determine well formed sentences (syntax),
  - Rules to interpret sentences (semantics),
  - Rules to infer useful information (proof theory)

Good Specifications

- Specification languages often more expressive than computer languages
- Hence, specifications more concise than computer programs
- Good specifications:
  - **Adequate** for the problem at hand
  - **Internally consistent** (single interpretation makes true all properties)
  - **Unambiguous** (only one *interesting* interpretation makes the specification true)
  - **Complete** (the set of specified properties must be enough)
- Probably as difficult as writing a good computer program

Why Formally?

- Lack of ambiguity (present in, e.g., natural language)
- Even computer languages can show some degree of ambiguity!

```
if P1 if P2 C1; else C2;

a := b++c;
```

- Formality helps to check and derive further properties
- Automatically or, at least, systematically:

derive logical consequences through theorem proving; confirm that operational specifications satisfy abstract specifications; generate counterexamples otherwise; infer specifications from scenarios; animate the specification to check adequacy; generate invariants or liveness conditions; refine specifications and produce proof obligations; generate automatically test cases and oracles; support reuse and matching of components; ensure liveness and security

For Whom and When?

Useful at many levels:

- Consumers may approve specifications (not usual)
- Programmers use the specification as a reference guide
- Analyzers use the specification to discover incompleteness and inconsistencies in the original requirements
- Designers can use it to decompose and refine a software system
- Verification needs a previous specification
- Validation and debugging can take advantage of test cases and expected results generated by means of the specification
- Specifications can be used to document the path from requirements to implementation

Formal Methods and CBSE

- Developed models composed *after* inception
- Some may need to be extended (even dynamically reconfigured)
- Reuse is key: reasoning based on *compositional* properties (and not in global properties particular to a model)
- Lack of referential transparency in many languages an issue!
- Lack of *global vision* and architecture specification a problem
- Should be coupled with component specifications themselves

Pitfalls

Formal specification is not without problems:

- Specifications are never totally formal: an initial, informal definition of, e.g., properties, is always needed
- A translation from “informal” to “formal” is not enough
- Hard to develop and assess
- Modeling choices usually not documented (“fox syndrome”)
- Importance of byproducts usually neglected
- More useful when application domain is reduced

A Taxonomy

- Traditionally: model-based vs. property based
- Somewhat incomplete / confusing (intersection not empty, even without forcing the language)
- Alternative classification:
  - **History-based** state the set of admissible histories; interpreted over time
  - **State-based** express the set of valid states at any arbitrary snapshot; use invariants and pre/post conditions
  - **Transition-based** characterize transitions between states; preconditions guard the transition
  - **Functional** classified as algebraic (capture data type behavior as equations) or higher-order
  - **Operational** rely on the definition of an (abstract) machine
- Will review VDM, a state-based well-known formal method

VDM Basics: Types, Functions, Operations

VDM in a Nutshell

- *Vienna Development Method*: IBM laboratory, Vienna
- Roughly and inaccurately:

ALGOL-60 Compiler	→	PL/I Compiler	→	UDL-3 Oper. Semantics	→	Denot. Semantics	VDM Funct. Part

- State-based language (several variants exist)
- Data types, invariants, preconditions, postconditions
- Type checking and proof obligations
- Logic of Partial Functions
- Implicit and explicit specifications

The Overall Picture

- A formal model in VDM is composed of:
  - Basic types,
  - Defined types (with many useful constructors)
  - *Invariants* for those types,
  - Explicit function definitions (including preconditions),
  - Implicit definitions (postconditions),
  - Not referentially transparent constructs,
  - Very possibly grouped into abstract data types (standard VDM-SL) or classes (VDM-PP)
- Not all of them have to be present in a given model
- Heavy use of (first-order[a]) logic
- Explicit function definitions using a relatively standard language
- Mathematical and computer-oriented syntax

[a] More on that later

Basic Types

Type Symbol	Values	Example Values	Operators
`nat`	*Natural numbers*	`0,1,...`	`+, -, *, ...`
`nat1`	`nat` *excluding 0*	`1,2,...`	`+, -, *, ...`
`int`	*integers*	`...,-1,0,1,...`	`+, -, *, ...`
`real`	*Real Numbers*	`3.1415`	`+, -, *, ...`
`char`	*Characters*	`'a', 'F', '$'`	`=, <>`
`bool`	*Booleans*	`true, false`	`and, or, ...`
`token`[a]	*Not applicable*	*Not applicable*	`= , <>`
`quote`	*Named values*	*<Red>, <Bio>*	`= , <>`

- `Token`: used to represent any unknown / yet not define type
- Signatures: $+_{\texttt{nat}} : \texttt{nat} \times \texttt{nat} \rightarrow \texttt{nat}$
- What is the signature of $=$, $<>$?

[a] Special type

Explicit Function Definitions

- VDM features a (functional/procedural) programming language
- Function definitions include a signature and the expression defining the function:

$$f : X_1 \times \ldots \times X_n \rightarrow R$$
$$f(x_1, \ldots, x_n) \triangleq e(x_1, \ldots, x_n)$$

- Several arrows available
- Using computer notation:

```
f: X1 * ... * Xn -> R
f(x1, ..., xn) == ...
```

- E.g.: define multiplication based on addition

```
mult: nat * nat -> nat
mult(x, y) == if y = 1
              then x
              else mult(x, y - 1) + y
```

Implicit Function Definitions

- Sometimes one does not want / know how to define a function
- *Implicit* function definitions allow to express *what* is to be computed, not *how*

$f(x_1 : X_1, \ldots, x_n : X_n)\ r : R$
pre $P(x_1, \ldots, x_n)$
post $Q(x_1, \ldots, x_n, r)$

pre: what has to be true before calling; **post**: what will be true after calling

- Computer notation:

```
f (x1: X1, ..., xn: Xn) res: R
pre P(x1, ..., xn)
post Q(x1, ..., xn, res)
```

- Example:

```
mult(x: nat, y: nat) res: R
pre true
post res = x * y
```

- Implementations are required to be deterministic (e.g., $x \in T$)

Proof Obligations

- **pre** and **post**conditions impose *formulas to be met* by the function definition

$$pre\text{-}f(x_1, \ldots, x_n) \to post\text{-}f(x_1, \ldots, x_n, f(x_1, \ldots, x_n))$$

- These formulas have to be *discharged* (proved)
- By proving them we:
  - ensure that the model is consistent and that the functions implement the desired properties,
  - can find inconsistencies in the requirements
- Proofs:
  - Classically (by hand)
  - Automated prover (often proofs are trivial)
- Hard-to-prove proof obligations often pinpoint weak parts of the model / requirements

Implicit + Explicit

- Both can be used at the same time

$f : X_1 \times \ldots \times X_n \to R$
$f(x_1, \ldots, x_n) \triangleq e(x_1, \ldots, x_n)$
pre $P(x_1, \ldots, x_n)$
post $Q(x_1, \ldots, x_n, res)$

- Computer notation:

```
f: X1 * ... * Xn -> R
f (x1, ..., xn) == ...
pre P(x1, ..., xn)
post Q(x1, ..., xn, RESULT)
```

- Example:

```
mult: nat * nat -> nat
mult(x, y) == if y = 1
              then x
              else mult(x, y - 1) + y
pre true
post RESULT = x * y
```

- RESULT implicit identifier to express the result of the function

Operations

- VDM can also model changes to a global state
- Operations which do so have to explicitly declare that

$op(x_1 : X_1 \times \ldots \times x_n : X_n)\ r : R$
ext rd $i : I$
wr $io : IO$
pre $P(x_1, \ldots, x_n, i, io)$
post $Q(x_1, \ldots, x_n, i, io, \overleftarrow{io}, res)$

- External state: i and io
- Decorated $\overleftarrow{io}$: value of io after the operation executes

What Now?

- Express software system as a model
- Check Internal consistency:
  - Types (type system has rules)
  - Proof obligations (using LPF and proof theory, preconditions, postconditions, invariants)
- Check consistency with other modules (used or users)
- Reference for requirements analysis
- Reference for design and implementation:
  - Automatic (e.g., IFAD Tools)
  - Manual (refinement steps)

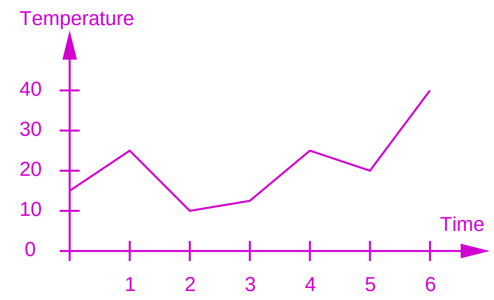
Logic

Logic(s)

Our ability to state invariants, record preconditions and post-conditions, and the ability to reason about a formal model depend on the logic on which the modeling language is based.

- Need to state invariants, record preconditions and post-conditions
- Reasoning about a formal model depends on the logic on which the modeling language is based
- Classical logical propositions and predicates
- Connectives
- Quantifiers
- Handling undefinedness: the logic of partial functions

The Temperature Monitor Example



The monitor records the last five temperature readings

25	10	5	5	10

The Temperature Monitor Example

- The following conditions are to be detected by the monitor:
  - **Rising:** the last reading in the sample is greater than the first
  - **Over limit:** there is a reading in the sample in excess of 400 C
  - **Continually over limit:** all the readings in the sample exceed 400 C
  - **Safe:** If readings do not exceed 400 C by the middle of the sample, the reactor is safe. If readings exceed 400 C by the middle of the sample, the reactor is still safe provided that the reading at the end of the sample is less than 400 C.
  - **Alarm:** The alarm is to be raised if and only if the reactor is not safe

Predicates and Propositions

- Predicates are logical expressions
- The simplest kind of logical predicate is a proposition
- Proposition: a logical assertion about a **particular** value or values
- Usually involving some operator to compare the values:

$$3 < 27$$

$$5 = 9$$

- Propositions are normally either true or false (classical logic)
- VDM handles also undefined values

First Order Predicates

- A logical expression that contains variables which can stand for one of a range of possible values, e.g.

$$x < 27$$

$$x^2 + x - 6 = 0$$

- The truth or falsehood of a predicate depends on the value taken by the variables

Predicates in the Monitor Example

- We will advance some data structures:
  - `Monitor` is an array of integers[a]

```
Monitor = seq of int
```

- Consider a monitor `m`
- First reading in `m`: `m(1)`; last reading: `m(5)`
- State that the first reading in `m` is strictly less than the last reading: `m(1)` $<$ `m(5)`
- The truth of the predicate depends on the value of `m`.

[a] Approximately; VDM sequences have properties not present in arrays

Predicates: The Rising Condition

- The last reading in the sample is greater than the first
- We can express the rising condition as a Boolean function:

```
Rising: Monitor -> bool
Rising(m) == m(1) < m(5)
```

- For any monitor m , the expression `Rising(m)` evaluates to true iff the last reading in the sample in m is higher than the first, e.g.

```
Rising([233,45,677,650,900], true)
Rising([433,45,677,650,298], false)
```


Basic logical operators

- We build more complex logical expressions out of simple ones using logical connectives
- A and B truth values (*true* or *false*)

Traditional	VDM	Name
$\neg A$	`not A`	Negation
$A \wedge B$	`A and B`	Conjunction
$A \vee B$	`A or B`	Disjunction
$A \rightarrow B$	`A => B`	Implication
$A \leftrightarrow B$	`A <=> B`	Biimplication

- Interpretation of expressions usually done using truth tables

Basic Logical Operators

- **Negation:** the opposite of some logical expression is true

A	$\neg A$
true	*false*
false	*true*

- E.g., the reading does not raise: `not Rising(mon)`
- **Disjunction:** alternatives that are not necessarily exclusive

A	B	$A \vee B$
false	*false*	*false*
false	*true*	*true*
true	*false*	*true*
true	*true*	*true*

- E.g., ***Over limit***: *There is a reading in the sample in excess of 400 C*

```
OverLimit: Monitor -> bool

OverLimit(m) ==
```


Basic Logical Operators

- **Conjunction:** all of a collection of predicates are true

A	B	$A \wedge B$
false	*false*	*false*
false	*true*	*false*
true	*false*	*false*
true	*true*	*true*

- ***Continually over limit***: *all readings in the sample exceed 400 C*

```
COverLimit: Monitor -> bool

COverLimit(m) ==
```

- De Morgan law: $\neg(A \vee B) \equiv \neg A \wedge \neg B$

Basic Logical Operators

- **Implication:** predicates which must be true under certain conditions

A	B	$A \rightarrow B$
false	*false*	*true*
false	*true*	*true*
true	*false*	*false*
true	*true*	*true*

- $A \rightarrow B \equiv \neg A \vee B$
- ***Safe:*** *If readings do not exceed 400 C by the middle of the sample, the reactor is safe. If readings exceed 400 C by the middle of the sample, the reactor is still safe provided that the reading at the end of the sample is less than 400 C.*

```
Safe: Monitor -> bool

Safe(m) ==
```


Basic Logical Operators

- Biimplication allows us to express equivalence

A	B	$A \leftrightarrow B$
false	*false*	*true*
false	*true*	*false*
true	*false*	*false*
true	*true*	*true*

- $A \leftrightarrow B \equiv (A \rightarrow B) \wedge (B \rightarrow A)$
- ***Alarm*** *is true if and only if the reactor is not safe*

```
Alarm(m) =
```

- This can also be recorded as an invariant property (more on that later)

Quantifiers

- For large collections of values, using a variable makes more sense than dealing with each case separately.
- `inds m` represents indices (1-5) of the sample
- The "over limit" condition can then be expressed more economically as: *There is an index whose reading is over 400*
- "Continually over limit" condition can be expressed more succinctly
- **Existential** quantifier:

Logic	VDM notation
$\exists x \bullet P(x)$	`exists Binding & Predicate`

- **Universal** quantifier:

Logic	VDM notation
$\forall x \bullet P(x)$	`forall Binding & Predicate`

Quantifiers in VDM

- Bindings *restrict* the set of value a variable ranges over
  - Type bindings:

    `x: nat` $x \in \mathbb{N}$

    `n: seq of char` $n \in$ *seq of char*
  - Set bindings:

    `i in set inds m` $i \in$ *inds m*

    `x in set {1 ... 20}` $x \in \{1, \ldots, 20\}$
- Type binding: the bound variable ranges over a type (a possibly infinite collection of values); improves type information
- Set binding: the bound variable ranges over a **finite** set of values
- **Type: set of values**
- Unneeded in classical, type free, logic —no notion of "erroneous" or "undefined" values
- But there are type-aware logics (many-sorted logics)

Quantifiers

- Several variables may be bound at once by a single quantifier:

$$\forall \mathtt{x}, \mathtt{y} \in \{1, \ldots, 5\} \bullet \neg(\mathtt{m}(\mathtt{x}) = \mathtt{m}(\mathtt{y}))$$

  or, in VDM notation,

  `forall x, y in set {1 ... 5} & not m(x) = m(y)`
- Would this predicate be true for the following value of `m` ?

  `[320, 220, 105, 119, 150]`

Quantifiers: Exercises

1. All the readings in the sample are less than 400 and greater than 50
2. Each reading in the sample is up to 10 greater than its predecessor
3. There are two distinct readings in the sample which are over 400
4. There is a "single minimum" in the sequence of readings, i.e., there is a reading which is strictly smaller than any of the other readings
5. Reverse the order of the quantifiers in the previous example and give it a meaning

Deduction Rules

Valid derivations in propositional / predicate calculus are represented using inference rules, e.g.

$$\vee - I \quad \frac{E_i}{E_1 \vee E_2} \; (1 \leq i \leq 2) \qquad \forall - \mathit{defn} \quad \frac{\neg\exists x \in X \bullet \neg E(x)}{\forall x \in X \bullet E(x)}$$

$$\neg\neg - I \quad \frac{E}{\neg\neg E} \qquad \forall - E \quad \frac{\forall x \in X \bullet E(x);\, s \in X}{E(s/x)}$$

$$\vdots \qquad\qquad \vdots$$

$$\mathit{contr} \quad \frac{E_1; \neg E_1}{E}$$

- Any good book on classical logic should include a detailed discussion on them.
- VDM completes them with rules for types and equality

Coping with Undefinedness

- **LPF**: *Logic of Partial Functions*
- $f : X_1 \times \ldots \times X_n \to R$ **total** if for any $c_1 : X_1, \ldots, c_n : X_n$ the expression $f(c_1, \ldots, c_n)$ is defined, and **partial** otherwise
- What if a function yields no (suitable) value for some element in the domain?

```
subp: int * int -> int
subp(x, y) ==
    if x = y
       then 0
       else subp(x, y + 1) + 1
pre y =< x
post RESULT = x - y
```

No value ever returned if $x < y$, e.g., `subp(0, 1)`

- Proof obligation:

$$\forall x, y \in \mathbb{N} \bullet y \leq x \to \mathit{subp}(x, y) \in \mathbb{N} \wedge \mathit{subp}(x, y) = x - y$$

Logic of Partial Functions

- When antecedent *false*, whole formula is *true*
- However `subp` will not denote a natural number
- How can we *determine* the truth value of `subp(0, 1) = 1`?
- What values have to be assigned to expressions where terms fail to denote values?
- Logic in VDM is equipped with facilities for handling undefined

$$\forall x : \mathbb{N} \bullet x = 0 \vee \frac{x}{x} = 1$$

- **Can't evaluate** disjunction when $x = 0$
- Even if order-sensitive operators (*cand*, *cor*) are used
- However, it is a **key** property of numbers

Basic LPF Operators

Disjunction: If one disjunct is true, the whole disjunction is true

A	B	$A \vee B$
false	*false*	*false*
*	*false*	*false*
false	*	*false*
false	*true*	*true*
*	*true*	*true*
true	*	*true*
*	*	*
true	*true*	*true*

Conjunction: If one conjunct is false, the whole conjunction is false

A	B	$A \wedge B$
false	*false*	*false*
*	*false*	*false*
false	*	*false*
false	*true*	*false*
*	*true*	*
true	*	*
*	*	*
true	*true*	*true*

A	$\neg A$
true	*false*
false	*true*
*	*

Negation: negating the undefined is undefined

Last Operators and Some Properties

- Tables for $\rightarrow$ and $\leftrightarrow$ can be deduced from their definitions (do it)
- Does De Morgan law hold? (test it)
- Existential: $\exists x \bullet P(x) \equiv P(c_0) \vee P(c_1) \vee \cdots$
- Universal: $\forall x \bullet P(x) \equiv P(c_0) \wedge P(c_1) \wedge \cdots$
- Notably, excluded middle ($E \vee \neg E$) does not hold!
- Some proofs more involved than in classical logic
- VDM includes specific *proof rules* for all implicit operations

Points to Take into Account

It should be noted that:

- Propositional (no variables) calculus is always decidable
  - But computationally hard
- Pure predicate calculus is semi-decidable
  - An algorithm can prove that a sentence is a theorem (provable) when it is a theorem
  - Do not mix *being provable in a formal system* with *being true in a model*!
- Predicate calculus with equality axioms and interpreted functions is not decidable
  - There are *true sentences* which are not provable, and whose negation is not provable either

More Types and Constructions: Sequences, Sets, Mappings, Records, . . .

Non-Basic Types in VDM

- VDM is equipped with structured types
- Will review them very shortly:
  - Sets,
  - Mappings,
  - Sequences,
  - Records,
  - Cartesian and union types,
  - Type definitions and invariants
- Mathematical script counterparts will be given when reasonably well known and appropriate

Sets

- Finite, non-indexed, collection of values, with no repetition, order immaterial
- Type constructor:
  `T1 = set of T2` $T_1 = T_2\text{−set}$
- `T1`: class of all possible finite sets with elements drawn from `T2`
- Examples:
  `Coins = set of nat1` *Coins* $= \mathbb{N}_1\text{−set}$
  `Alphabet = set of char` *Alphabet* $= \text{nat−set}$
- Values:
  `{'a', 'g', 'K'}`
  `{{-2, -3, 1}, {}, {3, 0}}`

Defining Sets

- Enumeration: `{}`, `{4.3, 5.6}`
- Integer subrange: `{3, ..., 11}`
- Comprehension: `{expression | binding & predicate}`
- Set of values of `expression` under each assignment of variables in `binding` which satisfy `predicate`
- Examples:
  `{x | x: nat & x < 5}` $\{x | x \in \mathbb{N} \bullet x < 5\}$
  `{y | y: nat & y < 0}` $\{y | y \in \mathbb{N} \bullet y < 0\}$
  `{x+y | x, y: nat & x<3 and y<4}`
  $\{x + y | x, y \in \mathbb{N} \bullet x < 3 \wedge y < 4\}$
  `{x*y | x, y: nat1 & (x > 1 or y > 1) and x*y < k}`
  $\{x * y | x, y \in \mathbb{N}_1 \bullet (x > 1 \vee y > 1) \wedge x * y < k\}$
- What is the meaning of the last one?

Set Operations

- Counterparts of the usual mathematical constructions
- Obey to usual foundations
- Recall that, e.g., Pascal already had some set operations
- Assume: $\mathtt{T}_X$ `= set of X`

`_ union _:` $\mathtt{T}_X$ `*` $\mathtt{T}_X$ `->` $\mathtt{T}_X$	*Set union*	$A \bigcup B$
`_ inter _:` $\mathtt{T}_X$ `*` $\mathtt{T}_X$ `->` $\mathtt{T}_X$	*Set intersection*	$A \bigcap B$
`_ \ _:` $\mathtt{T}_X$ `*` $\mathtt{T}_X$ `->` $\mathtt{T}_X$	*Set difference*	$A - B$
`card:` $\mathtt{T}_X$ `-> nat`	*Cardinality*	$\lvert A \rvert$
`_ in set _:` `X *` $\mathtt{T}_X$ `-> bool`	*Membership*	$x \in A$
`_ subset _:` $\mathtt{T}_X$ `*` $\mathtt{T}_X$ `-> bool`	*Subset testing*	$A \subset B$

- Note: all of them are total (modulo well-typedness)

Mappings

- Partial applications between two arbitrary sets
- **Very** expressive: mappings can represent sequences, hash tables, functions, ...
- Not available in most languages![a]
- One-to-one or many-to-one, never many-to-*
- This is an adequate basis for many other types:
  - Arrays: `inds s` $\mapsto$ `T`,
  - Bank accounts: `BankNumber` $\mapsto$ `Owner`,
  - (Hash) Tables, ...
- Mappings have to be finite to be well defined

[a] They resemble extensible hash/associative arrays, though

Mapping Constructors

- Type constructor:
  `T1 = map T2 to T3` $T_1 = T_2 \mapsto T_3$
- E.g.: `map nat to real`
- Mapping enumeration: finite set of *maplets*
  `{0 |-> 1, 1 |-> 1, 2 |-> 2, 3 |-> 6}`
  $\{0 \mapsto 1, 1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 6\}$
  `{0 |-> 0!, 1 |-> 1!, 2 |-> 2!, 3 |-> 3!}`
  $\{0 \mapsto 0!, 1 \mapsto 1!, 2 \mapsto 2!, 3 \mapsto 6!\}$
- Mapping comprehension:
  `{expression |-> expression | binding & predicate}`
- Examples:
  `{x|->x**2 | x:nat1 & x**2<3}` $\{x \mapsto x^2 | x \in \mathbb{N}_1 \bullet x^2 < 3\}$
  $\{x \mapsto y | x \in \{0, \ldots, 9\}, \quad y \in \mathbb{N} \bullet \lfloor 10^y \pi \rfloor \bmod 10 = x \wedge \nexists z \bullet z < y \wedge \lfloor 10^z \pi \rfloor \bmod 10 = x\}$

Operators on Mappings

$\mathtt{T}_{X,Y}$ `= map X to Y`

`dom:` $\mathtt{T}_{X,Y}$ `-> set of X`	*Domain*
`rng:` $\mathtt{T}_{X,Y}$ `-> set of Y`	*Range*
`_(_):` $\mathtt{T}_{X,Y}$ `* X -> Y`	*Lookup; partial*
`_ munion _:` $\mathtt{T}_{X,Y}$ `*` $\mathtt{T}_{X,Y}$ `->` $\mathtt{T}_{X,Y}$	*Mapping union; partial*
`_ ++ _:` $\mathtt{T}_{X,Y}$ `*` $\mathtt{T}_{X,Y}$ `->` $\mathtt{T}_{X,Y}$	*Overriding mapping union*

- Note that the lookup operator has the same syntax as indexing in sequences
- Other operators available to restrict mappings

Sequences

- Finite, indexed, collection of values (of any type)
- Order matters, repetitions allowed (unlike sets)
- Type constructor:
  `T1 = seq of T2` $T_1 = T_2^*$
- T1: class of all possible finite sequences with elements drawn from T2
- Examples:
  `Naturals = seq of nat` *Naturals* $= \mathbb{N}^*$
  `Matrix = seq of (seq of real)` *Matrix* $= (\mathbb{R}^*)^*$
- Values (write the corresponding type!)

```
[1, 2, -6, 7]
[{4.5, 7.6}, {-5, -0.9}]
```


Operators on Sequences

- Assume: T_X = `seq of X`

`hd:` T_X `-> X`	*First element; partial*
`tl:` T_X `->` T_X	*Tail; partial*
`len:` T_X `-> nat`	*Length of sequence*
`elems:` T_X `-> set of X`	*Set of elements in sequence*
`inds:` T_X `-> set of nat`	*Set of elements in sequence*
`_ ^ _:` T_X `*` T_X `->` T_X	*Concatenation*
`_ (_):` T_X `* nat -> X`	*N-th element; partial*
`_(_..._):` T_X`*nat*nat-> X`	*Subsequence; partial*

- `len` and `_(_)` obey to

$$s \in T_X \vdash \forall i \bullet 1 \leq i \leq \texttt{len}\ s \rightarrow s(i) \in X$$

- Behavior of the rest of the operators can be derived
- E.g., $x \in \texttt{elems}\ s \leftrightarrow \exists i \in \{1, \ldots, \texttt{len}\ s\} \bullet s(i) = x$

Sequence Example

- Alternatively merging two sequences

```
Merge: S * S -> S
Merge(s1, s2) ==
    if s1 = [] then s2 else
    if s2 = [] then s1 else
       [ hd s1, hd s2 ] ^ Merge(tl s1, tl s2)
```

- Write down the corresponding postcondition
- Note that the algorithm

```
Merge(s1, s2) ==
    if s1 = [] then s2
    else [hd s1] ^ Merge(tl s2, tl s1)
```

  should correspond to the same specification

Records

- Combine items of different types in a single unit
- Type constructor:

```
RecType :: FieldName1: Type1
           FieldName2: Type2
           ...
```

- Similar to C / C++ structures or Pascal / Ada records
- Example:

```
CarDef :: Plate:  nat
          Engine: seq of char
```

- Records also called *composites*

Constructing and Consulting Records

- Record definitions induce a construction function:

$$\textit{mk-RecType} : \textit{Type}_1 \times \textit{Type}_2 \times \cdots \rightarrow \textit{RecType}$$

- E.g., `mk_CarDef(345, "XFD88767DD")`
- Also, for each field a consulting function is created:

$$\textit{FieldName}_n : \textit{RecType} \rightarrow \textit{Type}_n$$

- E.g.,

```
Plate(mk_CarDef(345, "XFD88767DD")) = 345
Engine(mk_CarDef(345, "XFD88767DD")) = "XFD88767DD"
```

- Updating: μ function changes a single field
- Assume `Car = mk_CarDef(345, "XFD88767DD")`

```
mu(Car, Plate |-> 256) = mk_CarDef(256, "XFD88767DD")
```


Product, Union, Optional Components

- Cartesian product: tuple construction
  `T = T1 * T2 * ...` $T = T_1 \times T_2 \times \ldots$
- Values are tuples, assumed right associative, with selectors `fst` and `snd`
- Union of types:
  `T = T1 | T2 | ...` $T = T_1 | T_2 | \ldots$
- Any of the values in $T_1, T_2, \ldots$ is a value of T
- If $T_1, \ldots, T_n$ are disjoint, a function can discern the case at hand
- Optional component: `T = [T1]`
- Also as part of products, records
- If missing, value is `nil`

Invariants

- Restricting attention to some elements in the type is often convenient (types traditionally checkable at compile time)
- E.g., polar coordinate system or search trees
- In general, invariants help to have a *normal form*: each *object* has a *canonical representative*
- This makes equality testing easier
- VDM allows to associate an *invariant* (a predicate) to each new data type
- This invariant has to:
  - Be true (in addition to any precondition) before function application
  - Be true (in addition to any postcondition) upon function exit

An Invariant Example

- Polar coordinate system: (r, θ)
- We want rotate points (construction comes for free)

```
PolPoint = Polar :: Radius: real
                    Angle : real

Rotate: PolPoint * real -> PolPoint
Rotate (P, R) == ...
pre true
post RESULT = mu(P, Angle |-> Angle(P) + R)
```

- Invariant belongs to the data type, not to the function

```
PolPoint = Polar :: Radius: real
                    Angle : real
```

inv P == $(\mathit{Radius(P)} > 0 \land 0 \leq \mathit{Angle(P)} < 2\pi) \lor$
$(\mathit{Radius(P)} = 0 \land \mathit{Angle(P)} = 0)$

- Postcondition and function definitions have to be changed to respect invariant *inv-Polar*

Extended Examples

Extended Examples

- Will develop three longer examples:
  - Sequence-based standard stack
  - Record-based standard stack
  - Insertion in a sorted sequence
- We will try them with a set of tools (IFAD VDM TollBox)
- We will then study:
  - Generated proof obligations
  - Generated code
- IFAD VDM files include: module name and keyword to separate types, functions, etc.
- Will not show them here

VDM Model: Stack

- Using a sequence
- Type definition:
  `IStck = seq of int`
- Operations naturally use the corresponding sequence operations:

```
Empty: () -> IStck
Empty () == []
```
pre *true*
post $\mathit{RESULT} = []$

```
Top: IStck -> int
Top (S) == hd S
```
pre $S \neq []$
post $\mathit{RESULT} = hd\ S$

```
Pop: IStck -> IStck
Pop (S) == tl S
```
pre $S \neq []$
post $\mathit{RESULT} = tl\ S$

```
Push: IStck * int +> IStck
Push (S, E) == [E] ^ S
```
pre *true*
post $E = hd\ \mathit{RESULT} \land$
$S = tl\ \mathit{RESULT}$

Stack: Proof Obligations

- Different obligations if only implicit, explicit, or both definitiones are used
- We will have a look at some proof obligations
- `Pop`, `Top`: Need to ensure precondition

$$\forall S : \mathit{IStck} \bullet S \neq []$$

- Impossible to ensure in isolation: every call to `Pop`, `Top` has to guarantee it
- `Push`: need to ensure that algorithm really implements postcondition if precondition is assumed

$$\forall S \in \mathit{IStck}, E \in \mathbb{Z} \bullet \mathit{pre\text{-}Push}(S, E) \rightarrow \mathit{post\text{-}Push}(S, E, [E] \frown S))$$

- Trivial in this case

Proof Obligations: What For?

- They should be proved (discharged), or else they remain *pending* to prove:
  - Very difficult
  - Not true in general
- IFAD Toolbox points them out (besides making syntax and type checks)
- Theorem provers can help with the simpler ones (e.g., B tools, LARCH provers, *perfect*, Boyer-Moore, NuPrl, SETHEO, Stalmark's method, ...)
- If discharging a proof is hard (impossible?), we should worry

Code Generation: How?

- Specification → code is in general in the programmer's hands
- Specification provides a detailed, consistent, account of what is required
- Several tools available for different methods, however
- In particular: VDM-SL explicit specifications relatively easy to execute / translate
- Implicit specifications harder to translate, but more expressive
- Ually a computation method can be *read* after several *reifi cation* steps
- IFAD Tools can generate code to:
  - Implement functional specification
  - Test implicit specification
- Code relies on libraries to implement ADTs (e.g., sequences)

Stack: Type Defi nition

- Type based on a sequence (`SEQ`) template instantiated with `Int`

```
#define TYPE_IStck type_iL

class type_iL : public SEQ<Int> {
public:
  type_iL () : SEQ<Int>() {}
  type_iL (const SEQ<Int> &c) : SEQ<Int>(c) {}
  type_iL (const Generic &c) : SEQ<Int>(c) {}
};
```

- Interface given by (generic) sequence used to implement the operations

Stack: Code for Operations

```
TYPE_IStck vdm_Pop (const TYPE_IStck &vdm_S) {
    return (Generic)vdm_S.Tl();
}

Bool vdm_pre_Pop (const TYPE_IStck &vdm_S) {
    return (Generic)(Bool)!(vdm_S == Sequence());
}

Bool vdm_post_Pop (const TYPE_IStck &vdm_S,
                   const TYPE_IStck &vdm_RESULT) {
     return (Generic)(Bool)(vdm_RESULT == vdm_S.Tl());
}
```

- Code quite clear in this example (apart from type juggling —it should have been correctly generated)
- Note: separate generation and testing
- Will see other languages which remove this distinction

Stack Two: Using Records

- Non-linear data structures (e.g., trees) are awkward to implement with sequences
- Composites can be used to simulate *algebraic types*
- Types:

```
IStck = [ IStckNode ];
IStckNode :: Content: int
                Next: IStck;
```

- Note the *optional* type (implicit constant `nil` appears)
- Recal that records generate automatically functions to construct consult
- Other possibility:
  `IStck = int` $\times$ `[ IStck ]`
- And use functions `fst`, `snd` to access components

Stack Two: Operations

```
Empty: () +> IStck
Empty () == nil
pre true
post RESULT = nil;

Pop: IStck -> IStck
Pop (S) ==  S.Next
pre S ≠ nil
post ∃Head ∈ ℤ • S =
 mk_IStckNode(Head, RESULT);

Push: IStck * int +> IStck
Push (S, E) ==
    mk_IStckNode(E, S)
pre true
post  RESULT =
       mk_IStckNode(E, S);
```

```
Top: IStck -> int
Top (S) == S.Content
pre S ≠ nil
post  ∃Tail ∈ IStck • S =
  mk_IStckNode(RESULT, Tail);
```

Stack Two: Type Implementation

- Type a little more involved
- Custom record definition

```
enum  {
  vdm_IStckNode = TAG_TYPE_IStckNode,
  length_IStckNode = 2,
  pos_IStckNode_Content = 1,
  pos_IStckNode_Next = 2
};

class TYPE_IStckNode: public Record {
public:
  TYPE_IStckNode (): Record(TAG_TYPE_IStckNode, 2) {}
  TYPE_IStckNode &Init (Int p2, TYPE_IStack p3);
  TYPE_IStckNode (const Generic &c): Record(c) {}
}
```


Stack Two: Sample Code

```
TYPE_IStck vdm_Push (const TYPE_IStck &vdm_S,
                     const Int &vdm_E) {

    Record varRes_3(vdm_IStckNode, length_IStckNode);

    varRes_3.SetField(1, vdm_E);
    varRes_3.SetField(2, vdm_S);
    return (Generic) varRes_3;
}
```


Sorted Sequence

- Items (integers) are sorted in ascending order

SortedSeq = **seq of int**
inv S **==** $S = [] \vee \forall I, J \in \textit{inds}\ S \bullet I > J \rightarrow S(I) \geq S(J)$

- Invariant: restricts which elements of the type are admissible
- Why $S = [] \vee \ldots$? How could it be interpreted if logic is not LPF?
- It must hold on entry and upon exit of every operation
- It will therefore be part of the proof obligations
- Will model only two operation: creation (easy) and insertion (more difficult)

  ```
  Empty: () +> SortedSeq
  Empty () == []
  pre true
  post RESULT = [];
  ```


Sorted Sequence: Insertion

- Implicit definition (might have used dichotomy as well):

```
Insert: SortedSeq * int +> SortedSeq
Insert (S, E) ==
  cases true:
    (S = [])     -> [ E ],
    (E <= hd S) -> [ E ] ^ S,
    (E >  hd S) -> [ hd S ] ^ Insert(tl S, E)
  end
```

- Pre- and postconditions:

pre $\textit{inv-SortedSeq}(S)$
post $\textit{len}\ S + 1 = \textit{len}\ \textit{RESULT} \wedge \textit{inv-SortedSeq}(\textit{RESULT}) \wedge$
$\textit{let}\ S1 = [E]^\frown S\ \textit{in}$
$\forall X \in (\textit{elems}\ \textit{RESULT} \bigcup \textit{elems}\ S1) \bullet$
$|\{I | I \in \textit{inds}\ \textit{RESULT} \bullet \textit{RESULT}(I) = X\}| =$
$|\{I | I \in \textit{inds}\ S1 \bullet S1(I) = X\}|$

Proof Obligations

- More interesting (and more involved)
- Exhaustive matching:

$$\begin{aligned}\forall S \in \textit{SortedSeq}, E \in \mathbb{Z} \bullet\ & \textit{inv-SortedSeq}(S) \rightarrow \\ & \textit{true} = (S = []) \vee \\ & \textit{true} = (E <= \textit{hd}(S)) \vee \\ & \textit{true} = (E > \textit{hd}(S))\end{aligned}$$

- Unneeded if `if-then-else` had been used
- Note the *true* $= \ldots$ to work around possible undefinedness

Proof Obligations

- Proof obligation for the recursive call

$$\begin{aligned}&\forall S \in \textit{SortedSeq}, E \in \mathbb{Z} \bullet \textit{inv-SortedSeq}(S) \rightarrow \\ &\quad \textit{true} \neq (S = []) \rightarrow \\ &\qquad \textit{true} \neq (E <= \textit{hd}(S)) \rightarrow \\ &\qquad\quad \textit{true} = (E > \textit{hd}(S)) \rightarrow \\ &\qquad\qquad \textit{pre-Insert}(\textit{tl}(S), E))\end{aligned}$$

- I.e., when `Insert` is recursively called, its precondition (which includes the type invariant) is met
- **Code:** long and complicated —based on sequences, includes:
  - Runtime error checks
  - Code to test invariants and postconditions

Validating Formal Models

The Idea of Validation

- Prove that a formal model describes the system the customer wanted
- Requirements often incomplete, incorrect, ambiguous: modelers have to resolve these
- However, a formal model can be approved by a customer
- Validation
  - Checking internal consistency of a model (always needed!)
  - Checking that the model describes the required behavior
- *Verifi cation* deals with ensuring that the system satisfies its specification
  - Unneeded if system automatically generated by another system verified and validated

Internal Consistency

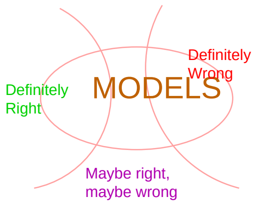
- In a formal language we should have:
  - A formal, unambiguous syntax
  - A formal semantics: rules to determine the meaning of every sentence
- Formal syntax → can be checked with an automatic tool
- Formal semantics → some properties (but **not all**) can be checked with an automatic tool (e.g., a type checker)
- **Type checking** and **proof obligations**

Validating Behavior

- **Formal proofs**
  - Excellent coverage
  - Not supported by all tools and formal methods
- **Animation**
  - Run the model through an interpreter
  - Good for inexpert users
- **Systematic testing**
  - Assess coverage
  - Quality depends on the tests performed
  - Automatic test generation possible (testing all / most / many paths)

Type Checking

- In general, type systems are designed to be checkable at compile time (VDM-SL's is)
- But some are not, and either human intervention or run-time checks is needed
- Preconditions and invariants are usually expressive enough as to be not (automatically) provable



Much of current research aims at developing languages and tools to reduce the size of the middle area by performing more checks automatically

Proof Obligations

- When checks cannot be performed automatically, mathematical proofs are needed
- Three types:
  - **Domain checking:** Every (partial) function is applied to values inside its domain (preconditions and invariants included)
    - **Protecting postconditions:** Defensive programming; applicability of automatic tools reduced
  - **Satisfi ability of explicit defi nitions:** The result of every function (assuming the preconditions hold) is in the right domain
  - **Satisfi ability of implicit defi nitions:** For every input satisfying the precondition there is an object satisfying the postcondition

Animation

- Execution of the model through an interface
- Dynamic link facility should exists to link the interface code to the model
- E.g., IFAD ToolBox has an interpreter and a C++/Java code generator + CORBA interface
- Increases confidence that a model accurately *reflects* the requirements
- Does not prove! (But problems found definitely problems)
- Customers rarely understand the modeling language —but they appreciate watching the model running

Systematic Testing

- Animation only as good as the choice of scenarios executed
- More systematic testing possible
  - Define a collection of test cases
  - Execute each test case on the formal model
  - Compare with expectation
- Test cases generated by hand or automatically
- Automatic generation can produce a vast number of test cases!
- Techniques for test generation in functional languages carry over to many formal models

Other Formal Specification Languages and Methods

Classical Models

- Date back to Turing
- Hoare logic:

```
{Pre}
  Sentence;
{Post}
```

- *Weakest Precondition* (WP):
  - Basic sentences have `{Pre}` / `{Post}` axioms
  - Sentence composition chain backward the *Weakest Precondition* at each point
  - Until program beginning is reached
  - Gries: *The Science of Programming*
- Impractical in real cases

Z Notation

- Spivey
- A notation, not a method (although application guidelines exist)
- Similar to VDM in many things: state based
- Preconditions *hidden* in postconditions
- Limitation object-oriented systems, concurrency (Z++ extension)
- Used in industrial development

The B Method

- J.R. Abrial
- State-based:
  - Stepwise refinement of *abstract machines*
  - Each step must be proved
  - Auxiliary tools (e.g., theorem provers) available
- Industrial success:
  - Paris underground, automating line 14
  - 100.000 lines of B code; refinement discovered many errors
  - 87.000 lines of Ada automatically generated
  - 27.000 tests
  - *No single error detected when conventional validation tests applied*

Axiomatic Specifications

- Data types as free algebraic structures
- Operation properties as minimal set of equations

```
sorts:  Stack, ℤ, 𝔹

new:                 → Stack
push:   Stack × ℤ    → Stack
pop:    Stack        → Stack ⋃ {error}
top:    Stack        → ℤ  ⋃ {error}
empty:  Stack        → 𝔹
```

```
pop(new())          =  error
pop(push(S,i))      =  S
top(new())          =  error
top(push(S,i))      =  i
empty(new())        =  true
empty(push(S,i))    =  false
```

- Implementations must obey the equations

Process Algebras: CSP

- Designed as a programming language (Hoare)
- Rich and complex algebra
- OCCAM: language based on CSP
- Process as first-order citizens: *STOP*, *RUN*, *SKIP*
- Communication
- Sequential, parallel, and alternative composition

- Π calculus (Milner): simplification of CSP
- More dynamic behavior
- A number of languages based on it: Pict, ELAN, Nepi, Piccola

Axiomatic Specifications

- OBJ, FOOPS
- Maude:
  - Equations evaluated non deterministically
  - Concurrency, reactive systems
  - *Reflexive* language
  - Good performance
  - Specifications with algorithmic flavor
  - Difficult to manage in practical cases

Temporal Logic

- Aims at specifying and validating concurrent and distributed systems
- Pnuelli, 77: time added to propositional logic
- Semantics: State of a program $\equiv$ assignment of values to variables
- Behaviors: List of states a program traverses in time
- Specify / prove existence of some behaviors
- Temporal operators:

$\bigcirc$	*in the next moment in time*
$\Box$	*at every future moment*
$\Diamond$	*at some future moment*

Specifying with Temporal Logics

- Interesting properties can be written very concisely:

  $\Box(\textit{send} \rightarrow \Diamond\ \textit{received})$: *it is always the case that if a message is sent it will be received in the future*

  $\Box(\textit{send} \rightarrow \bigcirc(\textit{received} \vee \textit{send}))$: *it is always the case that, if we send a message then, at the next moment in time, either the message will be received or we will send it again*

  $\Box\textit{send} \wedge \Box \rightarrow \neg\textit{received}$: it is always the case that if a message is received it cannot be sent again

- We should be able to deduce that $\Box\textit{send} \wedge \Box\neg\textit{received}$ is inconsistent (message continually resent, never received)

The Difficulty

- Many different temporal logics exist:
  - Different operators
  - Different idea of time (continuous, discrete, branching, . . .)
- Even *propositional, linear, discrete* temporal logic has high complexity:

  $$\vdash \Box(\varphi \rightarrow \bigcirc\varphi) \rightarrow (\varphi \rightarrow \Box\varphi)$$

  (induction axiom) can be read as

  $$[\forall i \bullet \varphi(i) \rightarrow \varphi(i+1)] \rightarrow [\varphi(0) \rightarrow \forall j \bullet \varphi(j)]$$

- I.e., the FOL induction axiom
- Decision procedure is PSPACE-complete
- Predicate temporal logic: things get even worse

Execution and Applications

- Resolution in temporal clauses: provers for temporal logic (detect inconsistencies, determine if some conclusion holds)
- Temporal logic programming
- Model checking:
  - Finite-state model captures execution of a system
  - Checked against a temporal formula
  - Used to verify hardware, network protocols, complex software
  - Technology evolving
- Does not reason, however, about scheduling or resource assignment

Just Logic?

- Can't classical logic be used directly?
- After all: used to specify (implicitly) in, e.g., VDM
- E.g., proving theorems to return answers: *Green's dream*
- This is the basic idea of Logic Programming
- With some restrictions on the source language for efficiency reasons
- Several languages based on it, notably Prolog
- Grown up: Constraint Logic Programming
- Highly expressive and reasonably fast (adequate for many applications)