**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:
- Seven Myths of Formal Methods Anthony Hall, Praxis Systems, IEEE Computer, September 1990
- 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 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
What is Formal Specification

- 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?

- 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
  - PL/I
  - UDL-3
  - VDM
- 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) logic
- Explicit function definitions using a relatively standard language
- Mathematical and computer-oriented syntax

Basic Types

<table>
<thead>
<tr>
<th>Type Symbol</th>
<th>Values</th>
<th>Example Values</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>nat</td>
<td>Natural numbers</td>
<td>0,1,...</td>
<td>+, -, *</td>
</tr>
<tr>
<td>nat_0</td>
<td>nat excl. 0</td>
<td>1,2,...</td>
<td>+, -, *</td>
</tr>
<tr>
<td>int</td>
<td>Integers</td>
<td>...,-1,0,1,...</td>
<td>+, -, *</td>
</tr>
<tr>
<td>real</td>
<td>Real Numbers</td>
<td>3.1415</td>
<td>+, -, *</td>
</tr>
<tr>
<td>char</td>
<td>Characters</td>
<td>'a', 'f', 's'</td>
<td>=, &lt;&gt;</td>
</tr>
<tr>
<td>bool</td>
<td>Booleans</td>
<td>true, false</td>
<td>and, or,</td>
</tr>
<tr>
<td>token*</td>
<td>Not applicable</td>
<td>Not applicable</td>
<td>=, &lt;&gt;</td>
</tr>
<tr>
<td>quote</td>
<td>Named values</td>
<td>&lt;Red&gt;, &lt;Bio&gt;</td>
<td>=, &lt;&gt;</td>
</tr>
</tbody>
</table>

Explicit Function Definitions

- VDM features a (functional/procedural) programming language
- Function definitions include a signature and the expression defining the function:
  
  \[ f: X_1 \times \ldots \times X_n \rightarrow R \]
  
  \[ f(x_1, \ldots, x_n) = \ldots \]

- Several arrows available
- Using computer notation:
  
  \[ f: X_1 \times \ldots \times X_n \rightarrow R \]
  
  \[ f(x_1, \ldots, x_n) = \ldots \]

- E.g.: define multiplication based on addition
  
  \[ \text{mult: nat \times nat \rightarrow nat} \]
  
  \[ \text{mult}(x, y) = \text{if } y = 1 \]
  
  \[ \text{then } x \]
  
  \[ \text{else mult}(x, y - 1) + y \]
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

\[
f(x_1, \ldots, x_n) : R
\]

\[
\text{pre: what has to be true before calling; post: what will be true after calling}
\]

- Computer notation:

\[
f(x_1: X_1, \ldots, x_n: X_n) \Rightarrow R
\]

\[
\text{Computer notation:}
\]

\[
f: X_1 \times \ldots \times X_n \rightarrow R
\]

\[
f(x_1, \ldots, x_n) \Rightarrow r(x_1, \ldots, x_n)
\]

\[
\text{pre } P(x_1, \ldots, x_n)
\]

\[
\text{post } Q(x_1, \ldots, x_n, r)
\]

- Example:

\[
mult(x: \text{nat}, y: \text{nat}) \Rightarrow \text{nat}
\]

\[
mult(x, y) = \begin{cases} 
  x & \text{if } y = 1 \\
  \text{mult}(x, y - 1) + y & \text{else}
\end{cases}
\]

\[
\text{pre true}
\]

\[
\text{post RESULT} = x * y
\]

- Implementations are required to be deterministic (e.g., \( x^2 \))

Proof Obligations

- \textbf{pre and post} conditions impose \textit{formulas} to be met by the function definition

\[
\text{pre-} f(x_1, \ldots, x_n) \Rightarrow \text{post-} f(x_1, \ldots, x_n, f(x_1, \ldots, x_n))
\]

- These formulas have to be \textit{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

\[
f: X_1 \times \ldots \times X_n \rightarrow R
\]

\[
f(x_1, \ldots, x_n) \Rightarrow r(x_1, \ldots, x_n)
\]

\[
\text{pre } P(x_1, \ldots, x_n)
\]

\[
\text{post } Q(x_1, \ldots, x_n, \text{RESULT})
\]

- Computer notation:

\[
\text{Example:}
\]

\[
mult: \text{nat} \times \text{nat} \rightarrow \text{nat}
\]

\[
mult(x, y) = \begin{cases} 
  x & \text{if } y = 1 \\
  \text{mult}(x, y - 1) + y & \text{else}
\end{cases}
\]

\[
\text{pre true}
\]

\[
\text{post RESULT} = x * y
\]

- \text{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

\[
\text{op}(x_1: X_1, \ldots, x_n: X_n) : R
\]

\[
\text{exl rd: I}
\]

\[
\text{wr io: IO}
\]

\[
\text{pre } P(x_1, \ldots, x_n, i, io)
\]

\[
\text{post } Q(x_1, \ldots, x_n, i, io, io, res)
\]

- External state: \( i \) and \( io \)
- Decorated \( io \) 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

- Both can be used at the same time
**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.

---

**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
    
    \[
    \text{Monitor} = \text{seq of int}
    \]

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

*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:
  \[ \text{Rising}(m) \rightarrow \text{bool} \]
  \[ \text{Rising}(m) = m(1) < m(5) \]
- For any monitor \( m \), the expression \( \text{Rising}(m) \) evaluates to true if the last reading in the sample in \( m \) is higher than the first, e.g.
  \[ \text{Rising([233, 45, 677, 650, 900]}, \text{true}) \]
  \[ \text{Rising([433, 45, 677, 650, 298]}, \text{false}) \]

Basic Logical Operators

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

<table>
<thead>
<tr>
<th>Traditional</th>
<th>VDM</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \neg A )</td>
<td>( \text{not } A )</td>
<td>Negation</td>
</tr>
<tr>
<td>( A \land B )</td>
<td>( A \land B )</td>
<td>Conjunction</td>
</tr>
<tr>
<td>( A \lor B )</td>
<td>( A \lor B )</td>
<td>Disjunction</td>
</tr>
<tr>
<td>( A \rightarrow B )</td>
<td>( A \rightarrow B )</td>
<td>Implication</td>
</tr>
<tr>
<td>( A \leftrightarrow B )</td>
<td>( A \leftrightarrow B )</td>
<td>Biimplication</td>
</tr>
</tbody>
</table>

Interpretation of expressions usually done using truth tables

- De Morgan law: \( \neg (A \lor B) \equiv \neg A \land \neg B \)
- Continually over limit: all readings in the sample exceed 400 C
  \[ \text{COverLimit}(m) \Rightarrow \text{bool} \]
  \[ \text{COverLimit}(m) = \]

- Implication: predicates which must be true under certain conditions
  \[ A \rightarrow B \equiv \neg A \lor B \]
  \[ \text{Safe}: \text{If readings do not exceed 400 C by the middle of the sample, the reactor is safe. If readings exceed 400 C by the middle of the sample, the reactor is still safe provided that the reading at the end of the sample is less than 400 C.} \]
  \[ \text{Safe}(m) = \]

- Bimpliation allows us to express equivalence
  \[ A \leftrightarrow B \equiv (A \rightarrow B) \land (B \rightarrow A) \]
  \[ \text{Alarm} \text{ is true if and only if the reactor is not safe} \]
  \[ \text{Alarm}(m) = \]

- 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:

Universal quantifier:

Bindings restrict the set of value a variable ranges over

Type bindings:

Set bindings:

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)

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

or, in VDM notation,

Would this predicate be true for the following value of m?


Quantiﬁers in VDM

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

\[ \forall x \in X \cdot E(x) \]

\[ \forall x \in X \cdot E(x) \]

\[ \forall x \in X \cdot E(x) \]

\[ \forall x \in X \cdot E(x) \]

\[ \forall x \in X \cdot E(x) \]

\[ \forall x \in X \cdot E(x) \]

\[ \forall x \in X \cdot E(x) \]

Any good book on classical logic should include a detailed discussion on them.

VDM completes them with rules for types and equality

LPF: Logic of Partial Functions

\[ f : X_1 \times \ldots \times X_n \rightarrow \text{It total} \]

If for any \( c_1, \ldots, c_n \in X_1 \times \ldots \times X_n \), 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?

No value ever returned if \( x < y \),

\[ \text{subp}(x, y) = \begin{cases} 0 & \text{if } x = y \\ \text{subp}(x, y + 1) + 1 & \text{else} \end{cases} \]

\[ \text{pre } y = x \\ \text{post RESULT } = x - y \]

Proof obligation:

\[ \forall x \in \mathbb{N} \cdot y \leq x \rightarrow \text{subp}(x, y) \in \mathbb{N} \land \text{subp}(x, y) = x - y \]
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
  \[ \forall x : \mathbb{N} \cdot x = 0 \lor x = 1 \]

Can't evaluate disjunction when \( x = 0 \)

- Even if order-sensitive operators (\( \text{cand} \), \( \text{cor} \)) are used
- However, it is a key property of numbers

Basic LPF Operators

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

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A ∨ B</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A ∧ B</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

Negation: negating the undefined is undefined

<table>
<thead>
<tr>
<th>A</th>
<th>~A</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

Points to Take into Account

- 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

Last Operators and Some Properties

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

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:
  - \( T_1 = \text{set of } T_2 \)
  - \( T_1 = \text{class of all possible finite sets with elements drawn from } T_2 \)
- Examples:
  - Coins = set of nat
  - Alphabet = set of char
- Values:
  - \{'a', 'g', 'K'\}
  - \{{-2, -3, 4}, {}, {3, 0}\}

**Defining Sets**

- Enumeration: \( () \), \( \{4.3, 5.6\} \)
- Integer subrange: \( \{3, \ldots, 11\} \)
- Comprehension: \( \{\text{expression} \mid \text{binding} \& \text{predicate}\} \)
- Set of values of expression under each assignment of variables in binding which satisfy predicate.
- Examples:
  - Coins = set of nat
  - Alphabet = nat
- Values:
  - \( \{x \mid x \in \text{nat} \& x < 5\} \)
  - \( \{y \mid y \in \text{nat} \& y < 0\} \)
  - \( \{x+y, x, y \in \text{nat} \& x \leq 3 \& y < 4\} \)
  - \( \{x^y, x, y \in \text{nat} \& (x > 1 \text{ or } y > 1) \& x \cdot y < k\} \)
- 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: \( T_V = \text{set of } X \)
  - \(_\cup \_ : T_X \times T_X \to T_X \) Set union
  - \(_\cap \_ : T_X \times T_X \to T_X \) Set intersection
  - \(_\setminus \_ : T_X \times T_X \to T_X \) Set difference
  - \(_\subseteq \_ : T_X \times T_X \to \text{bool} \) Subset testing
- 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!
- One-to-one or many-to-one, never many-to-*
- This is an adequate basis for many other types:
  - Arrays: \( \text{inds } s \to T \)
  - Bank accounts: \( \text{BankNumber } \to \text{Owner} \)
  - (Hash) Tables, ...
- Mappings have to be finite to be well defined

**Mapping Constructors**

- Type constructor:
  - \( T_1 = \text{map } T_2 \to T_3 \)
- E.g.: map nat to real
- Mapping enumeration: finite set of maplets
  - \( \{0 \mid \to 1, 1 \mid \to 1, 2 \mid \to 2, 3 \mid \to 6\} \)
  - \( \{0 \mid \to 0!, 1 \mid \to 1!, 2 \mid \to 2!, 3 \mid \to 3!\} \)
- Mapping comprehension:
  - \( \{\text{expression} \mid \to \text{expression} \mid \text{binding} \& \text{predicate}\} \)
- Examples:
  - \( \{x \mid x \cdot x \leq 3 \mid x, y \in \text{nat} \& x \cdot y < k\} \)
  - \( \{x \mid y \in \{0, \ldots, 9\} \& y \in \text{nat} \& 10 \cdot y \mod 10 = x \} \)

**Operators on Mappings**

- \( T_{X,Y} = \text{map } X \to Y \)
- \( \text{dom} : T_{X,Y} \to \text{set of } X \)
- \( \text{rng} : T_{X,Y} \to \text{set of } Y \)
- \( \text{range} : T_{X,Y} \times T_X \to Y \)
- \( \text{lookup} : T_{X,Y} \times T_X \to Y \)
- \( \text{union} : T_{X,Y} \times T_{X,Y} \to T_{X,Y} \)
- \( \text{mapping union; partial} \)
- \( \text{Combining mappings} \)
- 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:**
  \[
  T_1 = \text{seq of } T_2
  \]
- **Examples:**
  - Naturals = seq of nat
  - Matrix = seq of (seq of real)
- **Values (write the corresponding type!):**
  - \([1, 2, -6, 7]\)
  - \([\{4.5, 7.6\}, \{-5, -0.9\}]\)

### Operators on Sequences

- **Assume:**
  \[
  T_X = \text{seq of } X
  \]
- **First element; partial**
- **Tail; partial**
- **Length of sequence**
- **Set of elements in sequence**
- **Concatenation**
- **N-th element; partial**
- **Subsequence; partial**

### Sequence Example

- **Alternatively merging two sequences**
  \[
  \text{Merge}(s1, s2) = \begin{cases} 
  s2 & \text{if } s1 = [] \\
  [\text{hd } s1, \text{hd } s2] \hat{\text{Merge}}(\text{tl } s1, \text{tl } s2) & \text{else}
  \end{cases}
  \]
- **Write down the corresponding postcondition**
- **Note that the algorithm**
  \[
  \text{Merge}(s1, s2) = \begin{cases} 
  s2 & \text{if } s1 = [] \\
  [\text{hd } s1] \hat{\text{Merge}}(\text{tl } s2, \text{tl } s1) & \text{else}
  \end{cases}
  \]
- **should correspond to the same specification**

### Records

- **Combine items of different types in a single unit**
- **Type constructor:**
  \[
  \text{RecType} :: \text{FieldName1: Type1} \rightleftharpoons \text{FieldName2: Type2} \rightleftharpoons \ldots
  \]
- **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:**
  \[
  \text{mk-RecType} : \text{Type1} \times \text{Type2} \times \cdots \rightarrow \text{RecType}
  \]
- **E.g.,**
  - \(\text{mk_CarDef}(345, "XFDB8767DD")\)
- **Also, for each field a consulting function is created:**
  \[
  \text{FieldName} : \text{RecType} \rightarrow \text{Type}
  \]
- **E.g.,**
  - \(\text{Plate(mk_CarDef(345, "XFDB8767DD")] = 345}\)
  - \(\text{Engine(mk_CarDef(345, "XFDB8767DD") = "XFDB8767DD"}\)
  - **Updating: \(\mu\) function changes a single field**
  - **Assume**
    - Car = mk_CarDef(345, "XFDB8767DD")
    - \(\text{mu(Car, Plate |--> 256) = mk_CarDef(256, "XFDB8767DD")}\)

### Product, Union, Optional Components

- **Cartesian product: tuple construction**
  \[
  T = T_1 \times T_2 \times \ldots
  \]
- **Values are tuples, assumed right associative, with selectors \(\text{fst}\) and \(\text{snd}\)**
- **Union of types:**
  \[
  T = T_1 | T_2 | \ldots
  \]
  - **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:**
  \[
  T = [T_1]
  \]
  - **Also as part of products, records**
  - **If missing, value is \text{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
- Invariant belongs to the data type, not to the function

---

An Invariant Example

- Polar coordinate system: \((r, \theta)\)
- We want rotate points (construction comes for free)
  
  \[
  \text{PolPoint} = \text{Polar} :: \text{Radius: real} \quad \text{Angle: real}
  \]
  
  \[
  \text{Rotate: PolPoint} * \text{real} \rightarrow \text{PolPoint}
  \]
  
  \[
  \text{Rotate} (P, R) = \ldots
  \]
  
  \[
  \text{pre true}
  \]
  
  \[
  \text{post RESULT} = \mu (P, \text{Angle} |\rightarrow \text{Angle}(P) + R)
  \]
  
  \[
  \text{inv P} = (\text{Radius}(P) > 0 \land 0 \leq \text{Angle}(P) < 2\pi) \lor \text{Radius}(P) = 0 \land \text{Angle}(P) = 0
  \]

- Postcondition and function definitions have to be changed to respect invariant \(\text{inv-Polar}\)

---

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:
  \[
  \text{IStack} = \text{seq of int}
  \]
- Operations naturally use the corresponding sequence operations:
  
  \[
  \text{Empty: ()} \rightarrow \text{IStack}
  \]
  
  \[
  \text{Empty} () = []
  \]
  
  \[
  \text{pre true}
  \]
  
  \[
  \text{post RESULT} = []
  \]
  
  \[
  \text{Pop: IStack} \rightarrow \text{IStack}
  \]
  
  \[
  \text{Pop} (S) = \text{tl } S
  \]
  
  \[
  \text{pre S} \neq []
  \]
  
  \[
  \text{post RESULT} = \text{tl } S
  \]
  
  \[
  \text{Top: IStack} \rightarrow \text{int}
  \]
  
  \[
  \text{Top} (S) = \text{hd } S
  \]
  
  \[
  \text{pre S} \neq []
  \]
  
  \[
  \text{post RESULT} = \text{hd } S
  \]
  
  \[
  \text{Push: IStack} * \text{int} \rightarrow \text{IStack}
  \]
  
  \[
  \text{Push} (S, E) = [E] ^* S
  \]
  
  \[
  \text{pre true}
  \]
  
  \[
  \text{post E} = \text{hd } \text{RESULT} \land S = \text{tl } \text{RESULT}
  \]

---

Stack: Proof Obligations

- Different obligations if only implicit, explicit, or both definitions are used
- We will have a look at some proof obligations
- Pop, Top: Need to ensure precondition
  \[
  \forall S : \text{IStack} \land S \neq []
  \]
  
  \[
  \text{Impossible to ensure in isolation: every call to Pop, Top has to guarantee it}
  \]
  
  \[
  \text{Push: need to ensure that algorithm really implements postcondition if precondition is assumed}
  \]
  
  \[
  \forall S \in \text{IStack}, E \in \text{Z} : \text{pre-Push}(S, E) \rightarrow \text{post-Push}(S, E, [E] ^* S)
  \]
  
  \[
  \text{Trivial in this case}
  \]
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
- Usually a computation method can be read after several reification 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 Definition

```c
#define TYPE_IStck type_I

class type_I:
    public SEQ<Integer> {
        public:
            type_I() :
                SEQ<Integer>() {}
            type_I(const SEQ<Integer> &c):
                SEQ<Integer>(c) {}
            type_I(const Generic &c):
                SEQ<Integer>{c} {}
    }
```

Stack: Code for Operations

```c
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());
}
```

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)
- Recall that records generate automatically functions to construct consult
- Other possibility:
  - IStck = int × [ IStck ]
- And use functions fst, snd to access components

Stack Two: Operations

```c
Empty: () \rightarrow IStck
Empty () == nil
pre true
post RESULT = nil;
```

```c
Pop: IStck \rightarrow IStck
Pop (S) == S.Next
pre S \neq nil
post \exists Head ∈ Z . S = mk_IStckNode(RESULT.Tail);
```

```c
Push: IStck * int \rightarrow IStck
Push (S, E) == mk_IStckNode(RESULT, S)
pred True
post RESULT = mk_IStckNode(E, S);
```
Stack Two: Type Implementation

- Type a little more involved
- Custom record definition

```cpp
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_IStckNode p3);
        TYPE_IStckNode (const Generic &c): Record(c) {}
};
```

Stack Two: Sample Code

```cpp
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 == []
- I; J 2 inds S I > J ! S(I) >= S(J)
- Invariant: restricts which elements of the type are admissible
- Why S == [] ? 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

```cpp
Empty () == []
pre true
post RESULT == []
```

Proof Obligations

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

```
forall S in SortedSeq, E in Z • inv-SortedSeq(S) ➔
true = (S == []) ∨
true = (E <= hd(S)) ∨
true = (E > hd(S))
```

- Unneeded if if-then-else had been used
- Note the true = … to work around possible undefinedness

Proof Obligations

- Proof obligation for the recursive call

```
forall S in SortedSeq, E in Z • inv-SortedSeq(S) ➔
true = (S == []) ➔
true = (E <= hd(S)) ➔
true = (E > hd(S)) ➔
pre-Insert(S, E)
```

- 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
- Verification 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

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
  - Satisfiability of explicit definitions: The result of every function (assuming the preconditions hold) is in the right domain
  - Satisfiability of implicit definitions: For every input satisfying the precondition there is an object satisfying the postcondition

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

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
  - Satisfiability of explicit definitions: The result of every function (assuming the preconditions hold) is in the right domain
  - Satisfiability of implicit definitions: 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 exist 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\} \{Post\}
  \{Sentence\}
  ```
- 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

<table>
<thead>
<tr>
<th>sorts: Stack, z, b</th>
</tr>
</thead>
<tbody>
<tr>
<td>new: Stack * z = Stack</td>
</tr>
<tr>
<td>pop: Stack * z = Stack U {error}</td>
</tr>
<tr>
<td>top: Stack * z = z U {error}</td>
</tr>
<tr>
<td>empty: Stack * b</td>
</tr>
</tbody>
</table>

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

Implementations must obey the equations

Temporal Logic

- Aims at specifying and validating concurrent and distributed systems
- Pnueli, 77: time added to propositional logic
- Semantics: State of a program ≡ assignment of values to variables
- Behaviors: List of states a program traverses in time
- Specify / prove existence of some behaviors
- Temporal operators:
  - in the next moment in time
  - at every future moment
  - some future moment

Specifying with Temporal Logics

- Interesting properties can be written very concisely:
  - \( \square \) (send → □ received): it is always the case that if a message is sent it will be received in the future
  - \( \square \) (send → □(received ∨ 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
  - \( \square \) (send ∧ □ → □ received): it is always the case that if a message is received it cannot be sent again
  - We should be able to deduce that \( \square \) (send ∧ □ received) is inconsistent (message continually resent, never received)

Temporal Logics

- Many different temporal logics exist:
  - Different operators
  - Different idea of time (continuous, discrete, branching, . . .)
  - Even propositional, linear, discrete temporal logic has high complexity:
    \( \vdash (\varphi \rightarrow \square \varphi) \rightarrow (\varphi \rightarrow \square \varphi) \)
    (induction axiom) can be read as
    \[ [\forall i (\varphi(i) \rightarrow \varphi(i + 1))] \rightarrow [\varphi(0) \rightarrow \forall j \varphi(j)] \]
  - I.e., the FOL induction axiom
  - Decision procedure is PSPACE-complete
  - Predicate temporal logic: things get even worse

The Difficulty

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