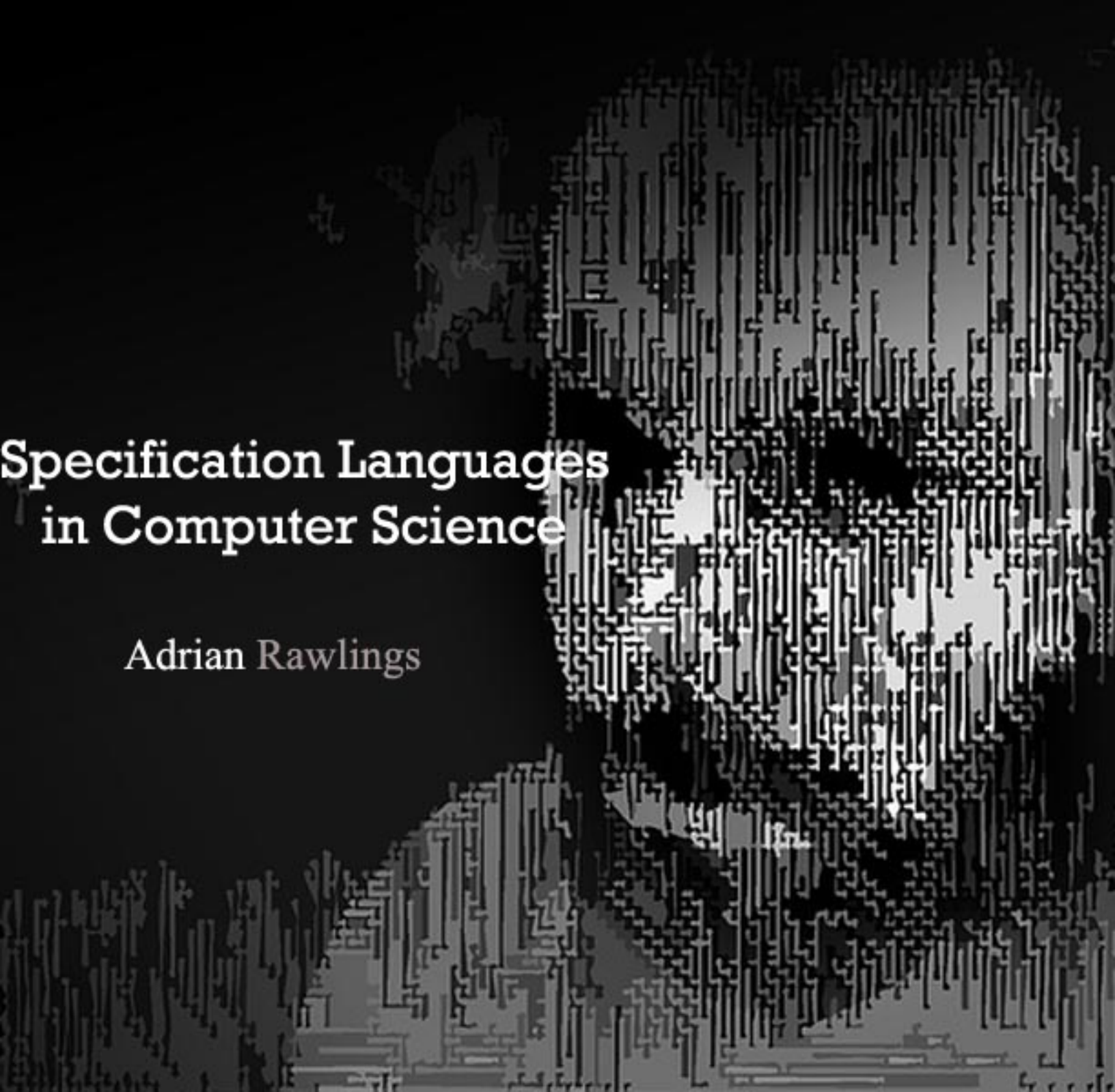
Specification Languages in Computer Science

Adrian Rawlings



First Edition, 2012

ISBN 978-81-323-3105-6

Published by:
Research World
4735/22 Prakashdeep Bldg,
Ansari Road, Darya Ganj,
Delhi - 110002
Email: info@wtbooks.com

Table of Contents

Introduction

A **specification language** is a formal language used in computer science. Unlike most programming languages, which are directly executable formal languages used to implement a system, specification languages are used during systems analysis, requirements analysis and systems design.

Specification languages are generally not directly executed. They describe the system at a much higher level than a programming language. Indeed, it is considered as an error if a requirement specification is cluttered with unnecessary implementation detail, because the specification is meant to describe the *what*, not the *how*.

A common fundamental assumption of many specification approaches is that programs are modelled as algebraic or model-theoretic structures that include a collection of sets of data values together with functions over those sets. This level of abstraction is commensurate with the view that the correctness of the input/output behaviour of a program takes precedence over all its other properties.

In the *property-oriented* approach to specification (taken e.g. by CASL), specifications of programs consist mainly of logical axioms, usually in a logical system in which equality has a prominent role, describing the properties that the functions are required to satisfy - often just by their interrelationship. This is in contrast to so-called model-oriented specification in frameworks like VDM and Z, which consist of a simple realization of the required behaviour.

Specifications must be subject to a process of *refinement* (the filling-in of implementation detail) before they can actually be implemented. The result of such a refinement process is an executable algorithm, which is either formulated in a programming language, or in an executable subset of the specification language at hand. For example, Hartmann pipelines, when properly applied, may be considered a dataflow specification which *is* directly executable. Another example is the Actor model which has no specific application content and must be *specialized* to be executable.

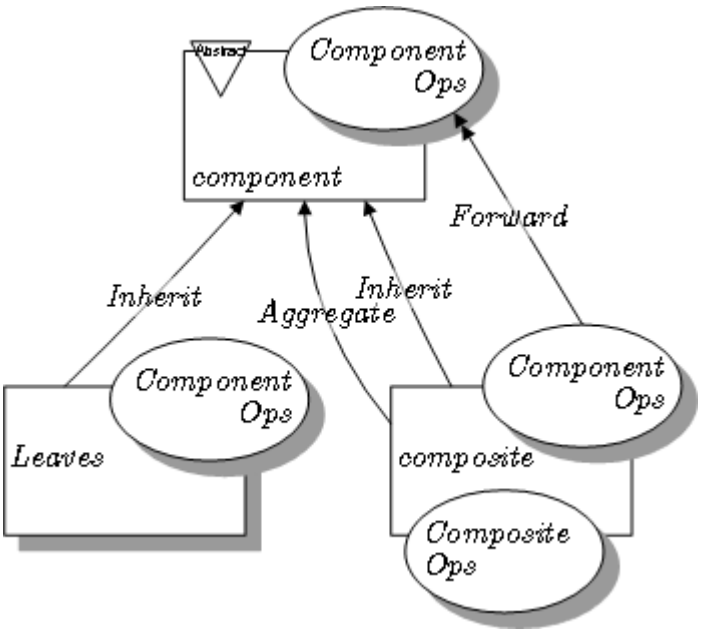
An important use of specification languages is enabling the creation of proofs of program correctness.

Languages

- CASL
- VDM
- Z notation
- LePUS3 (a visual, object-oriented design description language)
- Perfect

Chapter 1

LePUS3



Example: A composite pattern in LePUS3.

LePUS3 is an object-oriented, visual Design Description Language, namely a software modelling language and a formal specification language that is suitable primarily for modelling object-oriented (Java, C++, C#) programs and design motifs such as design patterns. It is defined as an axiomatized subset of First-order predicate logic. A specification in LePUS3 is also called a **Codechart**.

LePUS, the name of the first version of the language, is an abbreviation for *Language for Pattern Uniform Specification*.

Purpose

LePUS3 is tailored for the following purposes:

- *Scalability*: To model large-scale programs using small charts with only few symbols
- *Automated verifiability*: To allow programmers to check fully automatically conformance to design so as to keep the design in synch with the implementation
- *Program visualization*: To allow tools to reverse-engineer legible charts from plain source code modeling their design
- *Pattern implementation*: To allow tools to determine automatically whether your program implements a design pattern
- *Design abstraction*: To specify unimplemented programs without committing prematurely to implementation minutiae
- *Genericity*: To model a design pattern not as a specific implementation but as a design motif
- *Rigour*: To allow software designers to be sure exactly what design charts mean and reason rigorously about them

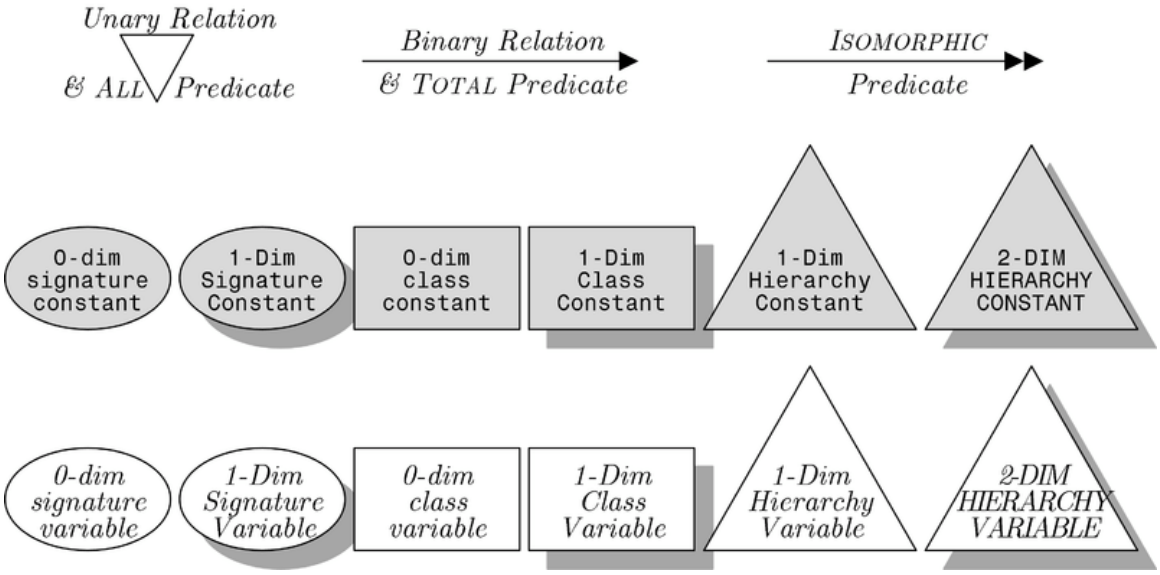
Context

LePUS3 belongs to the following families of languages:

- *Object-oriented software modeling languages* (e.g., UML): LePUS3 is a visual notation that is used to represent the building-blocks in the design of programs object-oriented programming languages
- *Formal specification languages*: Like other Logic Visual Languages, LePUS3 charts articulate sentences in mathematical logic. LePUS3 is axiomatized in and defined as a recursive (turing-decidable) subset of first-order predicate calculus. Its semantics are defined using finite structure (mathematical logic).
- *Architecture description languages*: LePUS3 is a non-functional specification language used to represent design decisions about programs in class-based object-oriented programming languages (such as Java and C++).
- *Tool supported specification languages*: Verification of LePUS3 charts (checking their consistency with a Java 1.4 program) can be established ('verified') by a click of a button, as demonstrated by the Two-Tier Programming Toolkit.
- *Software visualization notations* are notations which offer a graphical representation of the program, normally generated by reverse-engineering the source code of the program.

Vocabulary

LePUS3 was designed to accommodate for parsimony and for economy of expression. Its vocabulary consists of only 15 visual tokens.



LePUS3 Vocabulary

Tool support

The Two-Tier Programming Toolkit can be used to

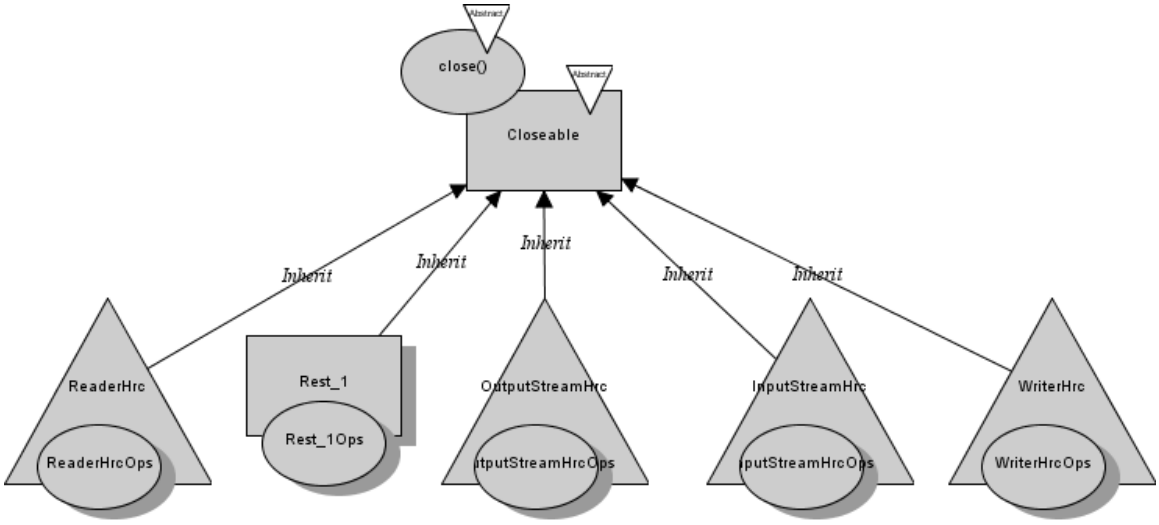
- create LePUS3 specifications (charts)
- verify automatically the consistency of LePUS3 charts with Java 1.4 programs; and
- reverse-engineer charts from Java source code.

Design patterns

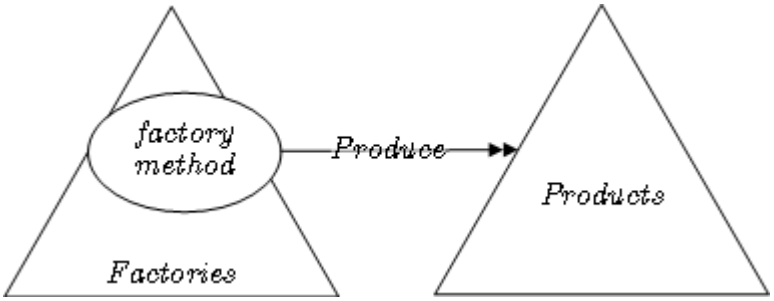
LePUS3 was specifically designed to model, among others, the 'Gang of Four' design patterns, including abstract factory, factory method, adapter, decorator, composite, proxy, iterator, state, strategy, template method, and visitor. The abbreviation LePUS for "Language for Pattern Uniform Specification" is used because the precursor of this language was primarily concerned with design patterns. The implementation of design patterns specified in LePUS3 can be automatically verified by the TTP Toolkit.

Examples

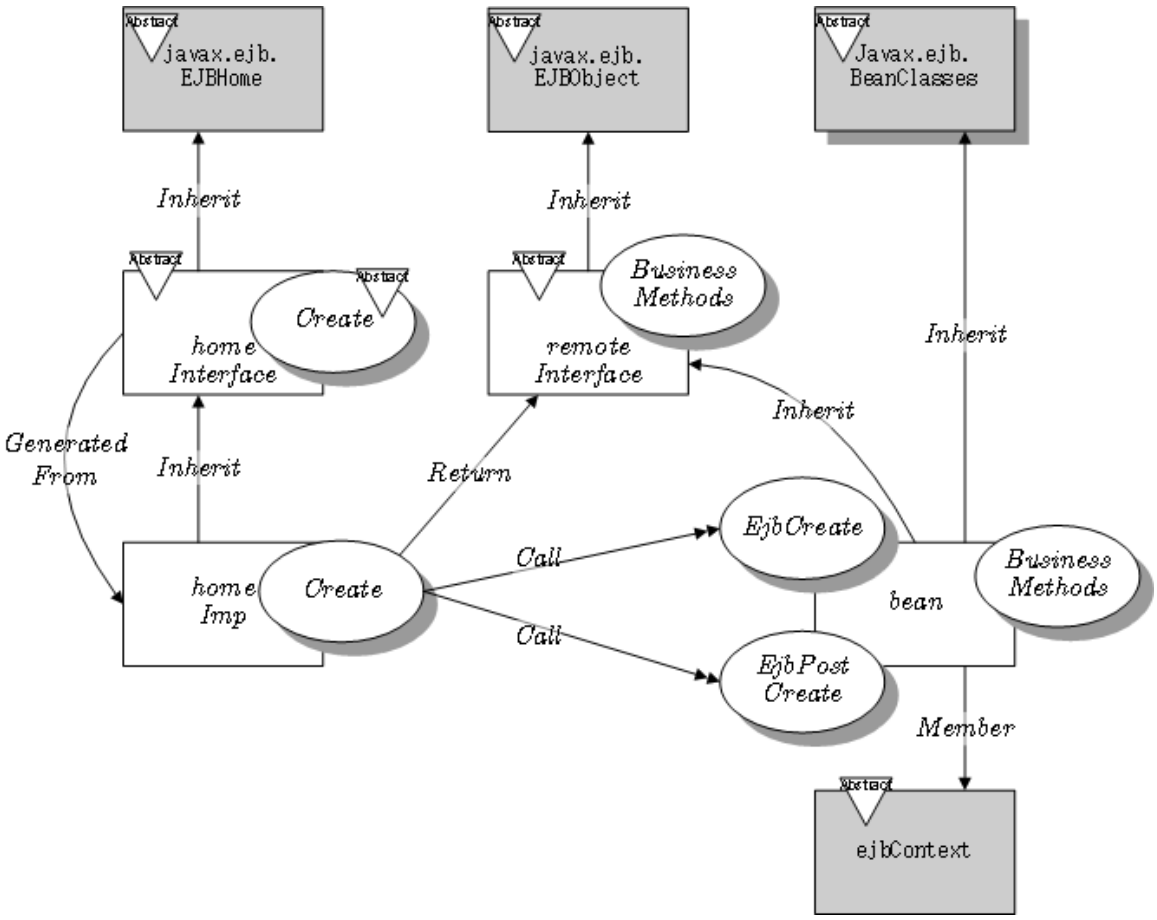
LePUS3 is particularly suitable for modelling large programs, design patterns, and object-oriented application frameworks. It is unsuitable for modelling non object-oriented programs, architectural styles, and undecidable and semi-decidable properties.



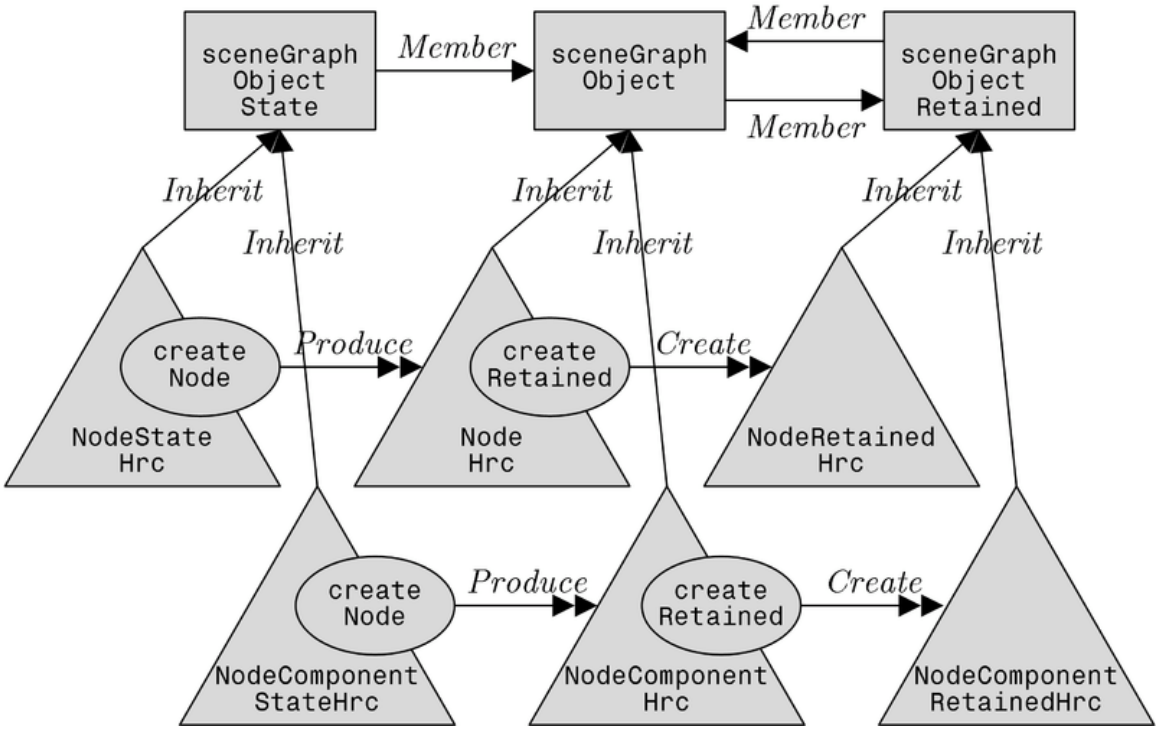
Closable hierarchy (java.io)



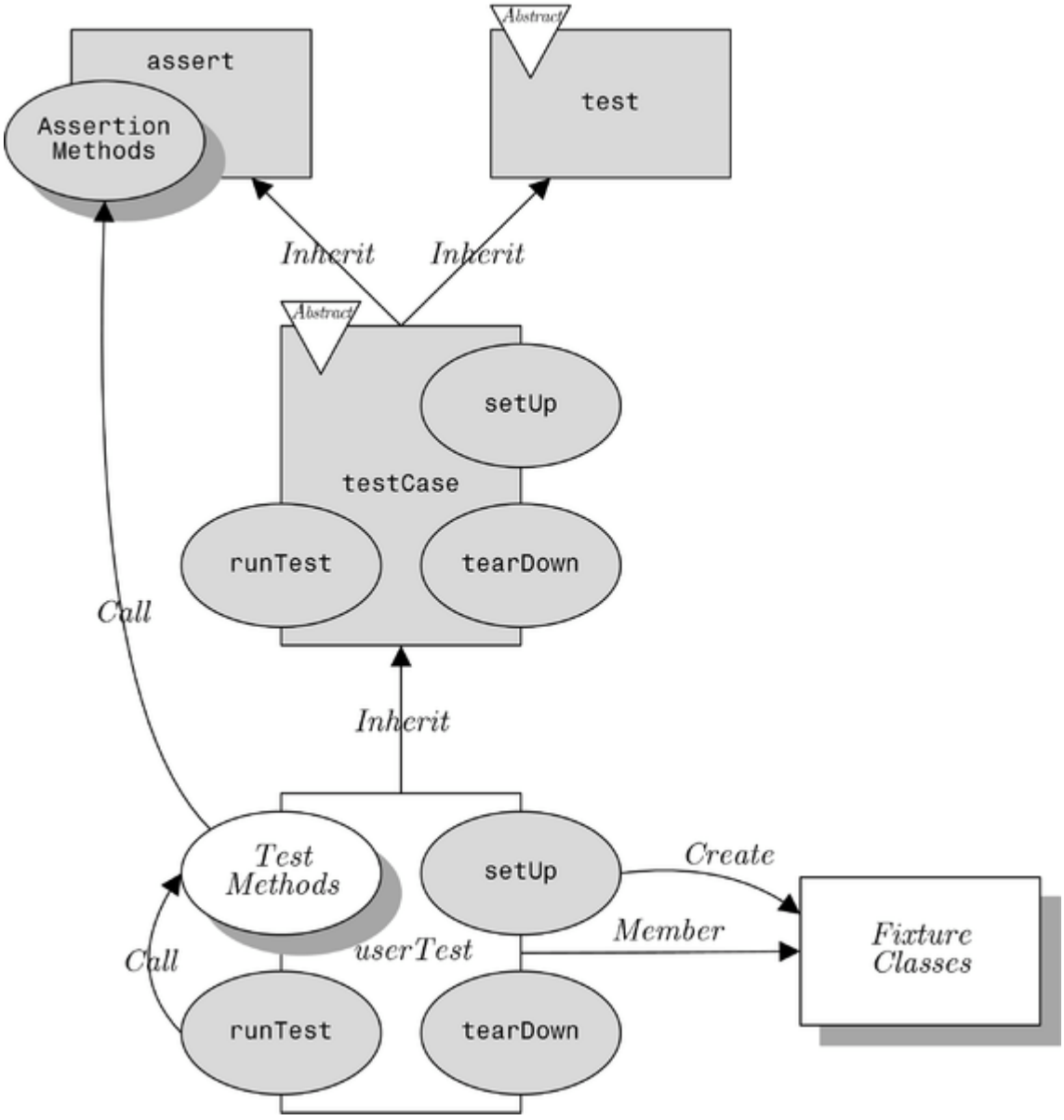
Factory method pattern



Enterprise JavaBeans


sceneGraph Object State
Member
sceneGraph Object
Member
Member
sceneGraph Object Retained
Inherit
Inherit
Inherit
Inherit
Inherit
Inherit
create Node
Produce
create Retained
Create
NodeState Hrc
Node Hrc
NodeRetained Hrc
create Node
Produce
create Retained
Create
NodeComponent StateHrc
NodeComponent Hrc
NodeComponent RetainedHrc

Java3D


assert
Assertion Methods
Abstract
test
Inherit
Inherit
Abstract
setUp
testCase
runTest
tearDown
Call
Inherit
Test Methods
setUp
Create
Member
Fixture Classes
Call
userTest
runTest
tearDown

JUnit

Chapter 2

Vienna Development Method

The **Vienna Development Method (VDM)** is one of the longest-established Formal Methods for the development of computer-based systems. Originating in work done at IBM's Vienna Laboratory in the 1970s, it has grown to include a group of techniques and tools based on a formal specification language - the VDM Specification Language (VDM-SL). It has an extended form, VDM++, which supports the modeling of object-oriented and concurrent systems. Support for VDM includes commercial and academic tools for analyzing models, including support for testing and proving properties of models and generating program code from validated VDM models. There is a history of industrial usage of VDM and its tools and a growing body of research in the formalism has led to notable contributions to the engineering of critical systems, compilers, concurrent systems and in logic for computer science.

Philosophy

Computing systems may be modeled in VDM-SL at a higher level of abstraction than is achievable using programming languages, allowing the analysis of designs and identification of key features, including defects, at an early stage of system development. Models that have been validated can be transformed into detailed system designs through a refinement process. The language has a formal semantics, enabling proof of the properties of models to a high level of assurance. It also has an executable subset, so that models may be analyzed by testing and can be executed through graphical user interfaces, so that models can be evaluated by experts who are not necessarily familiar with the modeling language itself.

History

The origins of VDM-SL lie in the IBM Laboratory in Vienna where the first version of the language was called the **V**ienna **D**efinition **L**anguage (VDL). The VDL was

essentially used for giving operational semantics descriptions in contrast to the VDM - Meta-IV which provided denotational semantics

«Towards the end of 1972 the Vienna group again turned their attention to the problem of systematically developing a compiler from a language definition. The overall approach adopted has been termed the "Vienna Development Method"... The meta-language actually adopted ("Meta-IV") is used to define major portions of PL/1 (as given in ECMA 74 - interestingly a "formal standards document written as an abstract interpreter") in BEKIČ 74.»

So Meta-IV, was "used to define major portions of" the PL/I programming language. Other programming languages described, or partially described, using Meta-IV and VDM-SL include the BASIC programming language, FORTRAN, the APL programming language, ALGOL 60, the Ada programming language and the Pascal programming language. Meta-IV evolved into several variants, generally described as the Danish, English and Irish Schools.

The "English School" derived from work by Cliff Jones on the aspects of VDM not specifically related to language definition and compiler design (Jones 1980, 1990). It stresses modelling persistent state through the use of data types constructed from a rich collection of base types. Functionality is typically described through operations which may have side-effects on the state and which are mostly specified implicitly using a precondition and postcondition. The "Danish School" (Bjørner *et al.* 1982) has tended to stress a constructive approach with explicit operational specification used to a greater extent. Work in the Danish school led to the first European validated Ada compiler.

An ISO Standard for the language was released in 1996 (ISO, 1996).

VDM Features

The VDM-SL and VDM++ syntax and semantics are described at length in the VDMTools language manuals and in the available texts. The ISO Standard contains a formal definition of the language's semantics.

A VDM-SL model is a system description given in terms of the functionality performed on data. It consists of a series of definitions of data types and functions or operations performed upon them.

Basic Types: numeric, character, token and quote types

VDM-SL includes basic types modelling numbers and characters as follows:

Basic Types		
`bool`	Boolean datatype	false, true
`nat`	natural numbers (including zero)	0, 1, 2, 3, ...

`nat1`	natural numbers (excluding zero)	1, 2, 3, 4, ...
`int`	integers	..., -3, -2, -1, 0, 1, 2, 3, ...
`rat`	rational numbers	a/b, where a and b are integers, b is not 0
`real`	real numbers	...
`char`	characters	A, B, C, ...
`token`	structureless tokens	...
`<A>`	the quote type containing the value `<A>`	...

Data types are defined to represent the main data of the modelled system. Each type definition introduces a new type name and gives a representation in terms of the basic types or in terms of types already introduced. For example, a type modelling user identifiers for a log-in management system might be defined as follows:

```
types
UserId = nat
```

For manipulating values belonging to data types, operators are defined on the values. Thus, natural number addition, subtraction etc. are provided, as are Boolean operators such as equality and inequality. The language does not fix a maximum or minimum representable number or a precision for real numbers. Such constraints are defined where they are required in each model by means of data type invariants—Boolean expressions denoting conditions that must be respected by all elements of the defined type. For example a requirement that user identifiers must be no greater than 9999 would be expressed as follows (where <= is the “less than or equal to” Boolean operator on natural numbers):

```
UserId = nat
inv uid == uid <= 9999
```

Since invariants can be arbitrarily complex logical expressions, and membership of a defined type is limited to only those values satisfying the invariant, type correctness in VDM-SL is not automatically decidable in all situations.

The other basic types include char for characters. In some cases, the representation of a type is not relevant to the model’s purpose and would only add complexity. In such cases, the members of the type may be represented as structureless tokens. Values of token types can only be compared for equality – no other operators are defined on them. Where specific named values are required, these are introduced as quote types. Each quote type consists of one named value of the same name as the type itself. Values of quote types (known as quote literals) may only be compared for equality.

For example, in modelling a traffic signal controller, it may be convenient to define values to represent the colours of the traffic signal as quote types:

```
<Red>, <Amber>, <FlashingAmber>, <Green>
```

Type Constructors: Union, Product and Composite Types

The basic types alone are of limited value. New, more structured data types are built using type constructors.

Basic Type Constructors	
`T1 | T2 | ... | Tn`	Union of types `T1,...,Tn`
`T1*T2*...*Tn`	Cartesian product of types `T1,...,Tn`
`T :: f1:T1 ... fn:Tn`	Composite (Record) type

The most basic type constructor forms the union of two predefined types. The type `(A|B)` contains all elements of the type A and all of the type `B`. In the traffic signal controller example, the type modelling the colour of a traffic signal could be defined as follows:

```
SignalColour =  <Red> | <Amber> | <FlashingAmber> | <Green>
```

Enumerated types in VDM-SL are defined as shown above as unions on quote types.

Cartesian product types may also be defined in VDM-SL. The type `(A1*…*An)` is the type composed of all pairs of values, the first element of which is from the type `A1` and the second from the type `A2` and so on. The composite or record type is a Cartesian product with labels for the fields. The type

```
T :: f1:A1
     f2:A2
     ...
     fn:An
```

is the Cartesian product with fields labelled `f1,…,fn`. An element of type `T` can be composed from its constituent parts by a constructor, written `mk_T`. Conversely, given an element of type `T`, the field names can be used to select the named component. For example, the type

```
Date :: day:nat1
        month:nat1
        year:nat
inv mk_Date(d,m,y) == day <=31 and month<=12
```

models a simple date type. The value `mk_Date(1,4,2001)` corresponds to 1 April 2001. Given a date `d`, the expression `d.month` is a natural number representing the month. Restrictions on days per month and leap years could be incorporated into the invariant if desired. Combining these:

```
mk_Date(1,4,2001).month = 4
```

Collections: Sets, Mappings and Sequences

Collection types model groups of values. Sets are finite unordered collections in which duplication between values is suppressed. Sequences are finite ordered collections (lists) in which duplication may occur and mappings represent finite correspondences between two sets of values.

The set type constructor (written `set of T` where `T` is a predefined type) constructs the type composed of all finite sets of values drawn from the type `T`. For example, the type definition

```
UGroup = set of UserId
```

defines a type `UGroup` composed of all finite sets of `UserId` values. Various operators are defined on sets for constructing their union, intersections, determining proper and non-strict subset relationships etc.

Main Operators on Sets (s, s1, s2 are sets)	
`{a, b, c}`	Set enumeration: the set of elements `a`, `b` and `c`
`{ x | x:T & P(x)}`	Set comprehension: the set of `x` from type `T` such that `P(x)`
`{i, ..., j}`	The set of integers in the range `i` to `j`
`e in set s`	`e` is an element of set `s`
`e not in set s`	`e` is not an element of set `s`
`s1 union s2`	Union of sets `s1` and `s2`
`s1 inter s2`	Intersection of sets `s1` and `s2`
`s1 \ s2`	Set difference of sets `s1` and `s2`
`dunion s`	Distributed union of set of sets `s`
`s1 psubset s2`	s1 is a (proper) subset of `s2`
`s1 subset s2`	s1 is a (weak) subset of `s2`
`card s`	The cardinality of set `s`

The finite sequence type constructor (written `seq of T` where `T` is a predefined type) constructs the type composed of all finite lists of values drawn from the type `T`. For example, the type definition

```
String = seq of char
```

Defines a type `String` composed of all finite strings of characters. Various operators are defined on sequences for constructing concatenation, selection of elements and subsequences etc. Many of these operators are partial in the sense that they are not defined for certain applications. For example, selecting the 5th element of a sequence that contains only three elements is undefined.

The order and repetition of items in a sequence is significant, so `[a, b]` is not equal to `[b, a]`, and `[a]` is not equal to `[a, a]`.

Main Operators on Sequences (s, s1,s2 are sequences)	
`[a, b, c]`	Sequence enumeration: the sequence of elements `a`, `b` and `c`
`[f(x) | x:T & P(x)]`	Sequence comprehension: sequence of expressions `f(x)` for each `x` of (numeric) type `T` such that `P(x)` holds (`x` values taken in numeric order)
`hd s`	The head (first element) of `s`
`tl s`	The tail (remaining sequence after head is removed) of `s`
`len s`	The length of `s`
`elems s`	The set of elements of `s`
`s(i)`	The i^{th} element of `s`
`inds s`	the set of indices for the sequence s
`s1^s2`	the sequence formed by concatenating sequences s1 and s2

A finite mapping is a correspondence between two sets, the domain and range, with the domain indexing elements of the range. It is therefore similar to a finite function. The mapping type constructor in VDM-SL (written `map T1 to T2`) where `T1` and `T2` are predefined types) constructs the type composed of all finite mappings from sets of `T1` values to sets of `T2` values. For example, the type definition

```
Birthdays = map String to Date
```

Defines a type `Birthdays` which maps character strings to `Date`. Again, operators are defined on mappings for indexing into the mapping, merging mappings, overwriting extracting sub-mappings.

Main Operators on Mappings	
`{a | -> r, b | -> s}`	Mapping enumeration: `a` maps to `r`, `b` maps to `s`
`{x | -> f(x) | x:T & P(x)}`	Mapping comprehension: `x` maps to `f(x)` for all `x` for type `T` such that `P(x)`
`dom m`	The domain of `m`
`rng m`	The range of `m`
`m(x)`	`m` applied to `x`
`m1 munion m2`	Union of mappings `m1` and `m2` (`m1`, `m2` must be consistent where they overlap)
`m1 ++ m2`	`m1` overwritten by `m2`

Structuring

The main difference between the VDM-SL and VDM++ notations are the way in which structuring is dealt with. In VDM-SL there is a conventional modular extension whereas VDM++ has a traditional object-oriented structuring mechanism with classes and inheritance.

Structuring in VDM-SL

In the ISO standard for VDM-SL there is an informative annex that contains different structuring principles. These all follow traditional information hiding principles with modules and they can be explained as:

- **Module naming**: Each module is syntactically started with the keyword `module` followed by the name of the module. At the end of a module the keyword `end` is written followed again by the name of the module.
- **Importing**: It is possible to import definitions that has been exported from other modules. This is done in an *imports section* that is started off with the keyword `imports` and followed by a sequence of imports from different modules. Each of these module imports are started with the keyword `from` followed by the name of the module and a module signature. The *module signature* can either simply be the keyword `all` indicating the import of all definitions exported from that module, or it can be a sequence of import signatures. The import signatures are specific for types, values, functions and operations and each of these are started with the corresponding keyword. In addition these import signatures name the constructs that there is a desire to get access to. In addition optional type information can be present and finally it is possible to *rename* each of the constructs upon import. For types one needs also to use the keyword `struct` if one wish to get access to the *internal structure* of a particular type.
- **Exporting**: The definitions from a module that one wish other modules to have access to are exported using the keyword `exports` followed by an exports module signature. The *exports module signature* can either simply consist of the keyword `all` or as a sequence of export signatures. Such *export signatures* are specific for types, values, functions and operations and each of these are started with the corresponding keyword. In case one wish to export the internal structure of a type the keyword `struct` must be used.
- **More exotic features**: In earlier versions of the VDM-SL tools there was also support for parameterized modules and instantiations of such modules. However, these features was taken out of VDMTools around 2000 because they was hardly ever used in industrial applications and there was a substantial number of tool challenges with these features.

Structuring in VDM++

In VDM++ structuring are done using classes and multiple inheritance. The key concepts are:

- **Class**: Each class is syntactically started with the keyword `class` followed by the name of the class. At the end of a class the keyword `end` is written followed again by the name of the class.
- **Inheritance**: In case a class inherits constructs from other classes the class name in the class heading can be followed by the keywords `is subclass of` followed by a comma-separated list of names of superclasses.
- **Access modifiers**: Information hiding in VDM++ is done in the same way as in most object oriented languages using access modifiers. In VDM++ definitions are per default private but in from of all definitions it is possible to use one of the access modifier keywords: `private`, `public` and `protected`.

Modelling Functionality

Functional Modelling

In VDM-SL, functions are defined over the data types defined in a model. Support for abstraction requires that it should be possible to characterize the result that a function should compute without having to say how it should be computed. The main mechanism for doing this is the *implicit function definition* in which, instead of a formula computing a result, a logical predicate over the input and result variables, termed a *postcondition*, gives the result's properties. For example, a function `SQRT` for calculating a square root of a natural number might be defined as follows:

```
SQRT(x:nat)r:real
post r*r = n
```

Here the postcondition does not define a method for calculating the result `r` but states what properties can be assumed to hold of it. Note that this defines a function that returns a valid square root; there is no requirement that it should be the positive or negative root. The specification above would be satisfied, for example, by a function that returned the negative root of 4 but the positive root of all other valid inputs. Note that functions in VDM-SL are required to be *deterministic* so that a function satisfying the example specification above must always return the same result for the same input.

A more constrained function specification is arrived at by strengthening the postcondition. For example the following definition constrains the function to return the positive root.

```
SQRT(x:nat)r:real
post r*r = n and r>=0
```

All function specifications may be restricted by *preconditions* which are logical predicates over the input variables only and which describe constraints that are assumed to be satisfied when the function is executed. For example, a square root calculating function that works only on positive real numbers might be specified as follows:

```
SQRTP(x:real)r:real
```

```
pre x >=0
post r*r = n and r>=0
```

The precondition and postcondition together form a *contract* that to be satisfied by any program claiming to implement the function. The precondition records the assumptions under which the function guarantees to return a result satisfying the postcondition. If a function is called on inputs that do not satisfy its precondition, the outcome is undefined (indeed, termination is not even guaranteed).

VDM-SL also supports the definition of executable functions in the manner of a functional programming language. In an *explicit* function definition, the result is defined by means of an expression over the inputs. For example, a function that produces a list of the squares of a list of numbers might be defined as follows:

```
SqList: seq of nat -> seq of nat
SqList(s) == if s = [] then [] else [(hd s)**2] ^ SqList(tl s)
```

This recursive definition consists of a function signature giving the types of the input and result and a function body. An implicit definition of the same function might take the following form:

```
SqListImp(s:seq of nat)r:seq of nat
post len r = len s and
     forall i in set inds s & r(i) = s(i)**2
```

The explicit definition is in a simple sense an implementation of the implicitly specified function. The correctness of an explicit function definition with respect to an implicit specification may be defined as follows.

Given an implicit specification:

```
f(p:T_p)r:T_r
pre pre-f(p)
post post-f(p, r)
```

and an explicit function:

```
f:T_p -> T_r
```

we say it satisfies the specification iff:

```
forall p in set T_p & pre-f(p) => f(p):T_r and post-f(p, f(p))
```

So, "`f` is a correct implementation" should be interpreted as "`f` satisfies the specification".

State-based Modelling

In VDM-SL, functions do not have side-effects such as changing the state of a persistent global variable. This is a useful ability in many programming languages, so a similar concept exists; instead of functions, *operations* are used to change **state variables** (AKA globals).

For example, if we have a state consisting of a single variable `someStateRegister : nat`, we could define this in VDM-SL as:

```
state Register of
  someStateRegister : nat
end
```

In VDM++ this would instead be defined as:

```
instance variables
  someStateRegister : nat
```

An operation to load a value into this variable might be specified as:

```
LOAD(i:nat)
ext wr someStateRegister:nat
post someStateRegister = i
```

The *externals* clause (`ext`) specifies which parts of the state can be accessed by the operation; `rd` indicating read-only access and `wr` being read/write access.

Sometimes it is important to refer to the value of a state before it was modified; for example, an operation to add a value to the variable may be specified as:

```
ADD(i:nat)
ext wr someStateRegister : nat
post someStateRegister = someStateRegister~ + i
```

Where the ~ symbol on the state variable in the postcondition indicates the value of the state variable before execution of the operation.

Examples

The *max* function

This is an example of an implicit function definition. The function returns the element from a set of positive integers:

```
max(s:set of nat)r:nat
pre card s > 0
post r in set s and
     forall r' in set s & r' <= r
```

The postcondition characterizes the result rather than defining an algorithm for obtaining it. The precondition is needed because no function could return an r in set s when the set is empty.

Natural number multiplication

```
multp(i,j:nat)r:nat
pre true
post r = i*j
```

Applying the proof obligation `forall p:T_p & pre-f(p) => f(p):T_r and post-f(p, f(p))` to an explicit definition of `multp`:

```
multp(i,j) ==
 if i=0
 then 0
 else if is-even(i)
      then 2*multp(i/2,j)
      else j+multp(i-1,j)
```

Then the proof obligation becomes:

```
forall i, j : nat & multp(i,j):nat and multp(i, j) = i*j
```

This can be shown correct by:

1. Proving that the recursion ends (this in turn requires proving that the numbers become smaller at each step)
2. Mathematical induction

Queue abstract data type

This is a classical example illustrating the use of implicit operation specification in a state-based model of a well-known data structure. The queue is modelled as a sequence composed of elements of a type `Qelt`. The representation is `Qelt` is immaterial and so is defined as a token type.

```
types
Qelt = token;
Queue = seq of Qelt;
state TheQueue of
  q : Queue
end
operations
ENQUEUE(e:Qelt)
ext wr q:Queue
post q = q~ ^ [e];
DEQUEUE()e:Qelt
ext wr q:Queue
pre q <> []
post q~ = [e]^q;
```

```
IS-EMPTY()r:bool
ext rd q:Queue
post r <=> (len q = 0)
```

Bank system example

As a very simple example of a VDM-SL model, consider a system for maintaining details of customer bank account. Customers are modelled by customer numbers (*CustNum*), accounts are modelled by account numbers (*AccNum*). The representations of customer numbers are held to be immaterial and so are modelled by a token type. Balances and overdrafts are modelled by numeric types.

```
AccNum = token;
CustNum = token;
Balance = int;
Overdraft = nat;
AccData :: owner : CustNum
           balance : Balance
state Bank of
  accountMap : map AccNum to AccData
  overdraftMap : map CustNum to Overdraft
inv mk_Bank(accountMap,overdraftMap) == for all a in set rng accountMap
& a.owner in set dom overdraftMap and
                                        a.balance >= -
overdraftMap(a.owner)
```

With operations: *NEWC* allocates a new customer number:

```
operations
NEWC(od : Overdraft)r : CustNum
ext wr overdraftMap : map CustNum to Overdraft
post r not in set dom ~overdraftMap and overdraftMap = ~overdraftMap ++
{ r |-> od};
```

NEWAC allocates a new account number and sets the balance to zero:

```
NEWAC(cu : CustNum)r : AccNum
ext wr accountMap : map AccNum to AccData
    rd overdraftMap map CustNum to Overdraft
pre cu in set dom overdraftMap
post r not in set dom accountMap~ and accountMap = accountMap~ ++ {r|->
mk_AccData(cu,0)}
```

ACINF returns all the balances of all the accounts of a customer, as a map of account number to balance:

```
ACINF(cu : CustNum)r : map AccNum to Balance
ext rd accountMap : map AccNum to AccData
post r = {an |-> accountMap(an).balance | an in set dom accountMap &
accountMap(an).owner = cu}
```

Tool Support

A number of different tools support VDM:

- VDMTools are the leading commercial tools for VDM and VDM++, owned, marketed, maintained and developed by CSK Systems, building on earlier versions developed by the Danish Company IFAD. The manuals) and a practical tutorial are available. All licenses are available, free of charge, for the full version of the tool. The full version includes automatic code generation for Java and C++, dynamic link library and CORBA support.
- Overture is a community-based open source initiative aimed at providing freely available tool support for VDM++ on top of the Eclipse platform. Its aim is to develop a framework for interoperable tools that will be useful for industrial application, research and education.
- SpecBox: from Adelard provides syntax checking, some simple semantic checking, and generation of a LaTeX file enabling specifications to be printed in mathematical notation. This tool is freely available but it is not being further maintained.
- LaTeX and LaTeX2e macros are available to support the presentation of VDM models in the ISO Standard Language's mathematical syntax. They have been developed and are maintained by the National Physical Laboratory in the UK. Documentation and the macros are available online.

Industrial experience

VDM has been applied widely in a variety of application domains. The most well-known of these applications are:

- Ada and CHILL compilers: The first European validated Ada compiler was developed by DDC-International using VDM. Likewise the semantics of CHILL and Modula-2 were described in their standards using VDM.
- ConForm: An experiment at British Aerospace comparing the conventional development of a trusted gateway with a development using VDM.
- Dust-Expert: A project carried out by Adelard in the UK for a safety related application determining that the safety is appropriate in the layout of industrial plants.
- The development of VDMTools: Most components of the VDMTools tool suite are themselves developed using VDM. This development has been made at IFAD in Denmark and CSK in Japan.
- TradeOne: Certain key components of the TradeOne back-office system developed by CSK systems for the Japanese stock exchange were developed using VDM. Comparative measurements exist for developer productivity and defect density of the VDM-developed components versus the conventionally developed code.
- FeliCa Networks have reported the development of an operating system for an integrated circuit for cellular telephone applications.

Refinement

Use of VDM starts with a very abstract model and develops this into an implementation. Each step involves **Data Reification**, then **Operation Decomposition**.

Data reification develops the abstract data types into more concrete data structures, while operation decomposition develops the (abstract) implicit specifications of operations and functions into algorithms that can be directly implemented in a computer language of choice.

Specification		Implementation
Abstract data type	—— Data reification →	Data structure
Operations	—— Operation decomposition →	Algorithms

Data reification

Data reification (stepwise refinement) involves finding a more concrete representation of the abstract data types used in a specification. There may be several steps before an implementation is reached. Each reification step for an abstract data representation `ABS_REP` involves proposing a new representation `NEW_REP`. In order to show that the new representation is accurate, a *retrieve function* is defined that relates `NEW_REP` to `ABS_REP`, i.e. `retr : NEW_REP -> ABS_REP`. The correctness of a data reification depends on proving *adequacy*, i.e.

```
forall a:ABS_REP & exists r:NEW_REP & a = retr(r)
```

Since the data representation has changed, it is necessary to update the operations and functions so that they operate on `NEW_REP`. The new operations and functions should be shown to preserve any data type invariants on the new representation. In order to prove that the new operations and functions model those found in the original specification, it is necessary to discharge two proof obligations:

- Domain rule:

```
forall r: NEW_REP & pre-OPA(retr(r)) => pre-OPR(r)
```

- Modelling rule:

```
forall ~r,r:NEW_REP & pre-OPA(retr(~r)) and post-OPR(~r,r) => post-
OPA(retr(~r,), retr(r))
```

Example data reification

In a business security system, workers are given ID cards; these are fed into card readers on entry to and exit from the factory. Operations required:

- `INIT()` initialises the system, assumes that the factory is empty
- `ENTER(p : Person)` records that a worker is entering the factory; the workers' details are read from the ID card)
- `EXIT(p : Person)` records that a worker is exiting the factory
- `IS-PRESENT(p : Person) r : bool` checks to see if a specified worker is in the factory or not

Formally, this would be:

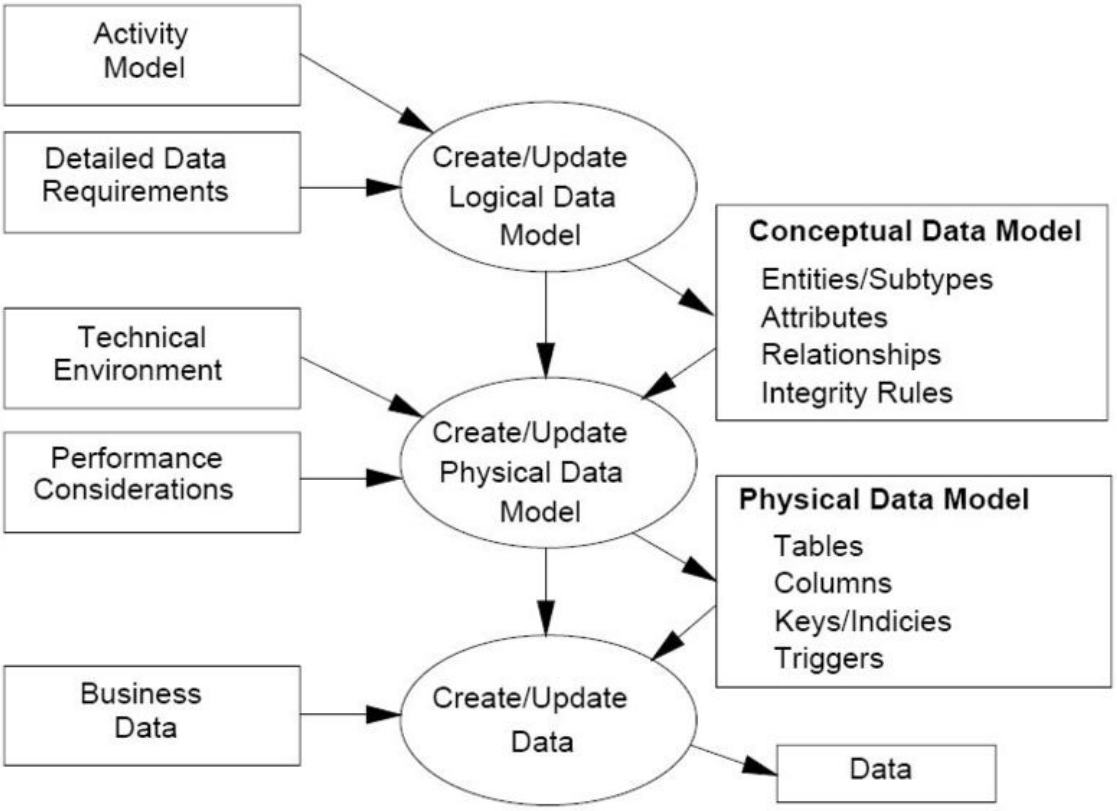
```
types
Person = token;
Workers = set of Person;
state AWCCS of
  pres: Workers
end
operations
INIT()
ext wr pres: Workers
post pres = {};
ENTER(p : Person)
ext wr pres : Workers
pre p not in set pres
post pres = pres~ union {p};
EXIT(p : Person)
ext wr pres : Workers
pre p in set pres
post pres = pres~\{p};
ISPRESENT(p : Person) r : bool
ext rd pres : Workers
post r <=> p in set pres~
```

As most programming languages have a concept comparable to a set (often in the form of an array), the first step from the specification is to represent the data in terms of a sequence. These sequences must not allow repetition, as we do not want the same worker to appear twice, so we must add an invariant to the new data type. In this case, ordering is not important, so `[a,b]` is the same as `[b,a]`.

The Vienna Development Method is valuable for model-based systems. It is not appropriate if the system is time-based. For such cases, the calculus of communicating systems (CCS) is more useful.

Chapter 3

Data Modeling



The data modeling process. The figure illustrates the way data models are developed and used today. A conceptual data model is developed based on the data requirements for the application that is being developed, perhaps in the context of an activity model. The data model will normally consist of entity types, attributes, relationships, integrity rules, and the definitions of those objects. This is then used as the start point for interface or database design.

Data modeling in software engineering is the process of creating a data model by applying formal data model descriptions using data modeling techniques.

Overview

Data modeling is a method used to define and analyze data requirements needed to support the business processes of an organization. The data requirements are recorded as a conceptual data model with associated data definitions. Actual implementation of the conceptual model is called a logical data model. To implement one conceptual data model may require multiple logical data models. Data modeling defines not just data elements, but their structures and relationships between them Data modeling techniques and methodologies are used to model data in a standard, consistent, predictable manner in order to manage it as a resource. The use of data modeling standards is strongly recommended for all projects requiring a standard means of defining and analyzing data within an organization, e.g., using data modeling:

- to manage data as a resource;
- for the integration of information systems;
- for designing databases/data warehouses (aka data repositories)

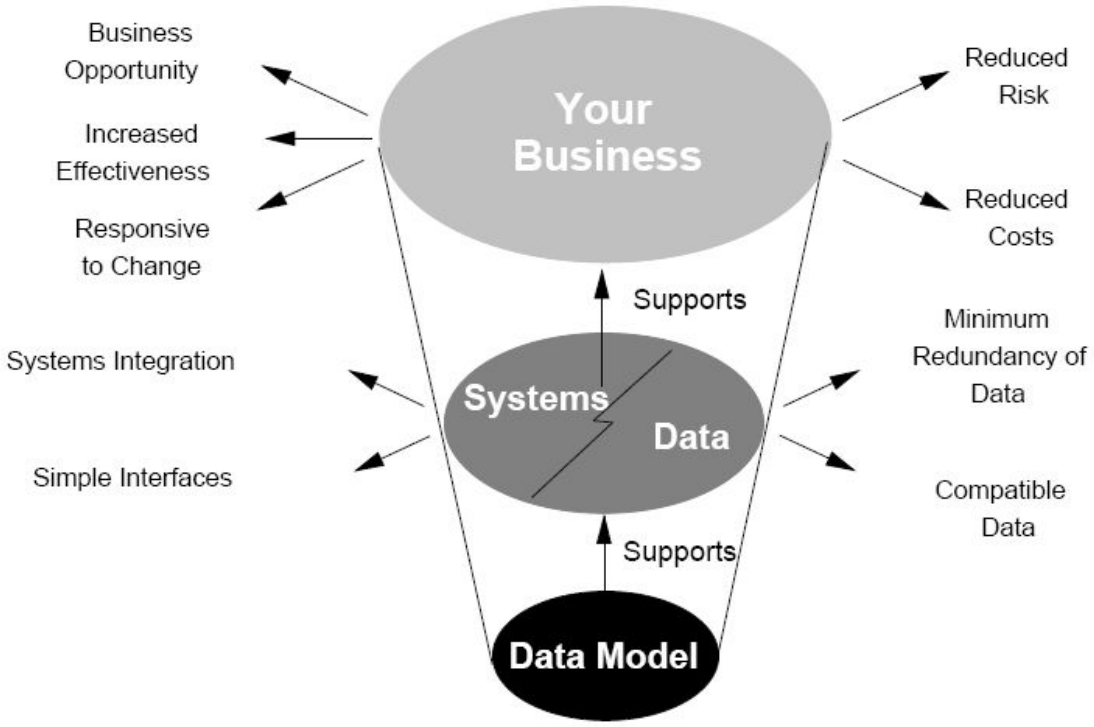
Data modeling may be performed during various types of projects and in multiple phases of projects. Data models are progressive; there is no such thing as the final data model for a business or application. Instead a data model should be considered a living document that will change in response to a changing business. The data models should ideally be stored in a repository so that they can be retrieved, expanded, and edited over time. Whitten (2004) determined two types of data modeling:

- Strategic data modeling: This is part of the creation of an information systems strategy, which defines an overall vision and architecture for information systems is defined. Information engineering is a methodology that embraces this approach.
- Data modeling during systems analysis: In systems analysis logical data models are created as part of the development of new databases.

Data modeling is also a technique for detailing business requirements for a database. It is sometimes called *database modeling* because a data model is eventually implemented in a database.

Data modeling topics

Data models



How data models deliver benefit.

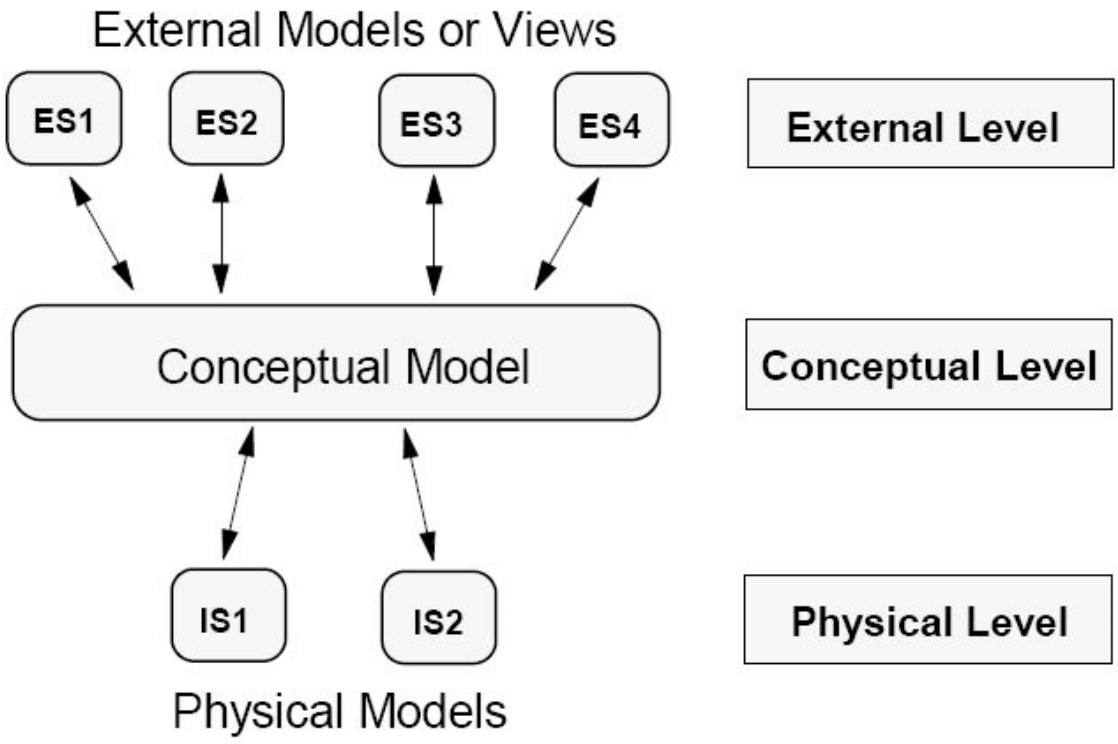
Data models support data and computer systems by providing the definition and format of data. If this is done consistently across systems then compatibility of data can be achieved. If the same data structures are used to store and access data then different applications can share data. The results of this are indicated above. However, systems and interfaces often cost more than they should, to build, operate, and maintain. They may also constrain the business rather than support it. A major cause is that the quality of the data models implemented in systems and interfaces is poor.

- Business rules, specific to how things are done in a particular place, are often fixed in the structure of a data model. This means that small changes in the way business is conducted lead to large changes in computer systems and interfaces.
- Entity types are often not identified, or incorrectly identified. This can lead to replication of data, data structure, and functionality, together with the attendant costs of that duplication in development and maintenance.
- Data models for different systems are arbitrarily different. The result of this is that complex interfaces are required between systems that share data. These interfaces can account for between 25-70% of the cost of current systems.
- Data cannot be shared electronically with customers and suppliers, because the structure and meaning of data has not been standardised. For example,

engineering design data and drawings for process plant are still sometimes exchanged on paper.

The reason for these problems is a lack of standards that will ensure that data models will both meet business needs and be consistent.

Conceptual, logical and physical schemes



The ANSI/SPARC three level architecture. This shows that a data model can be an external model (or view), a conceptual model, or a physical model. This is not the only way to look at data models, but it is a useful way, particularly when comparing models.

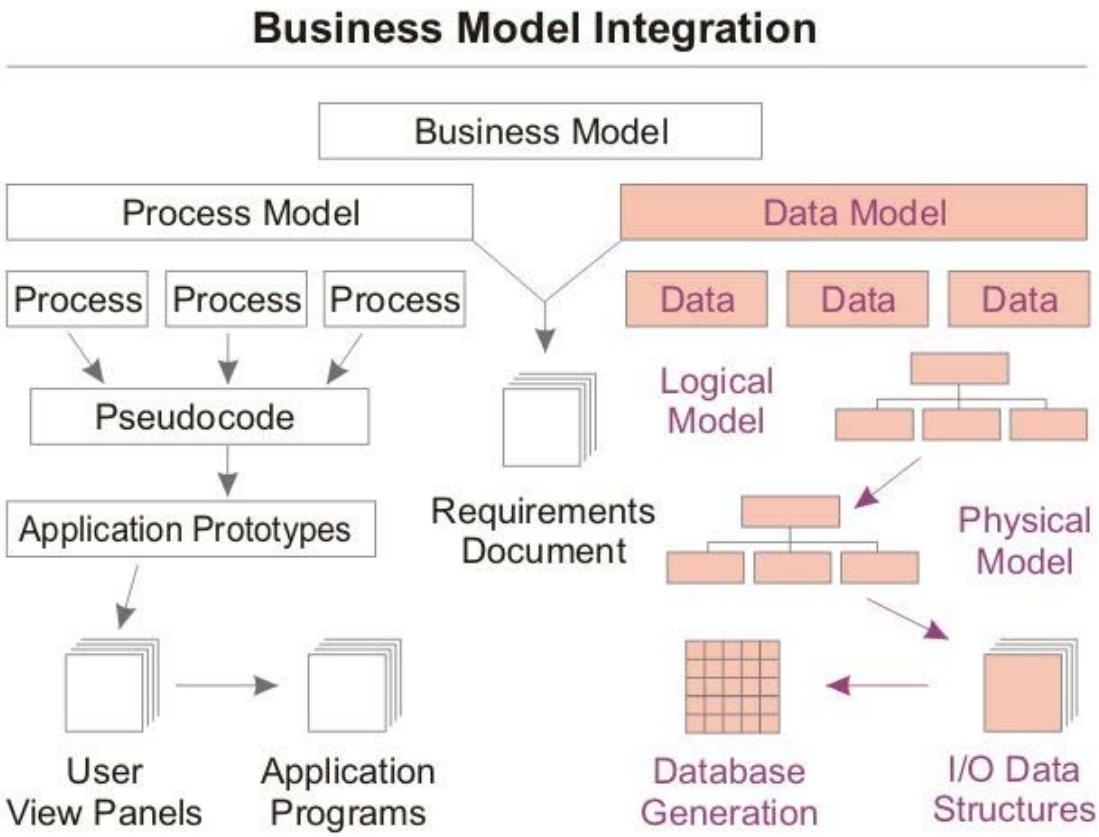
A data model *instance* may be one of three kinds according to ANSI in 1975:

- Conceptual schema: describes the semantics of a domain, being the scope of the model. For example, it may be a model of the interest area of an organization or industry. This consists of entity classes, representing kinds of things of significance in the domain, and relationships assertions about associations between pairs of entity classes. A conceptual schema specifies the kinds of facts or propositions that can be expressed using the model. In that sense, it defines the allowed expressions in an artificial 'language' with a scope that is limited by the scope of the model.
- Logical schema: describes the structure of some domain of information. This consists of descriptions of tables and columns, object oriented classes, and XML tags, among other things.

- Physical schema: describes the physical means by which data are stored. This is concerned with partitions, CPUs, tablespaces, and the like.

The significance of this approach, according to ANSI, is that it allows the three perspectives to be relatively independent of each other. Storage technology can change without affecting either the logical or the conceptual model. The table/column structure can change without (necessarily) affecting the conceptual model. In each case, of course, the structures must remain consistent with the other model. The table/column structure may be different from a direct translation of the entity classes and attributes, but it must ultimately carry out the objectives of the conceptual entity class structure. Early phases of many software development projects emphasize the design of a conceptual data model. Such a design can be detailed into a logical data model. In later stages, this model may be translated into physical data model. However, it is also possible to implement a conceptual model directly.

Data modeling process



Data modeling in the context of Business Process Integration.

In the context of Business Process Integration, see figure, data modeling will result in database generation. It complements business process modeling, which results in application programs to support the business processes.

The actual database design is the process of producing a detailed data model of a database. This logical data model contains all the needed logical and physical design choices and physical storage parameters needed to generate a design in a Data Definition Language, which can then be used to create a database. A fully attributed data model contains detailed attributes for each entity. The term database design can be used to describe many different parts of the design of an overall database system. Principally, and most correctly, it can be thought of as the logical design of the base data structures used to store the data. In the relational model these are the tables and views. In an Object database the entities and relationships map directly to object classes and named relationships. However, the term database design could also be used to apply to the overall process of designing, not just the base data structures, but also the forms and queries used as part of the overall database application within the Database Management System or DBMS.

In the process, system interfaces account for 25% to 70% of the development and support costs of current systems. The primary reason for this cost is that these systems do not share a common data model. If data models are developed on a system by system basis, then not only is the same analysis repeated in overlapping areas, but further analysis must be performed to create the interfaces between them. Most systems contain the same basic components, redeveloped for a specific purpose. For instance the following can use the same basic classification model as a component:

- Materials Catalogue,
- Product and Brand Specifications,
- Equipment specifications.

The same components are redeveloped because we have no way of telling they are the same thing.

Modeling methodologies

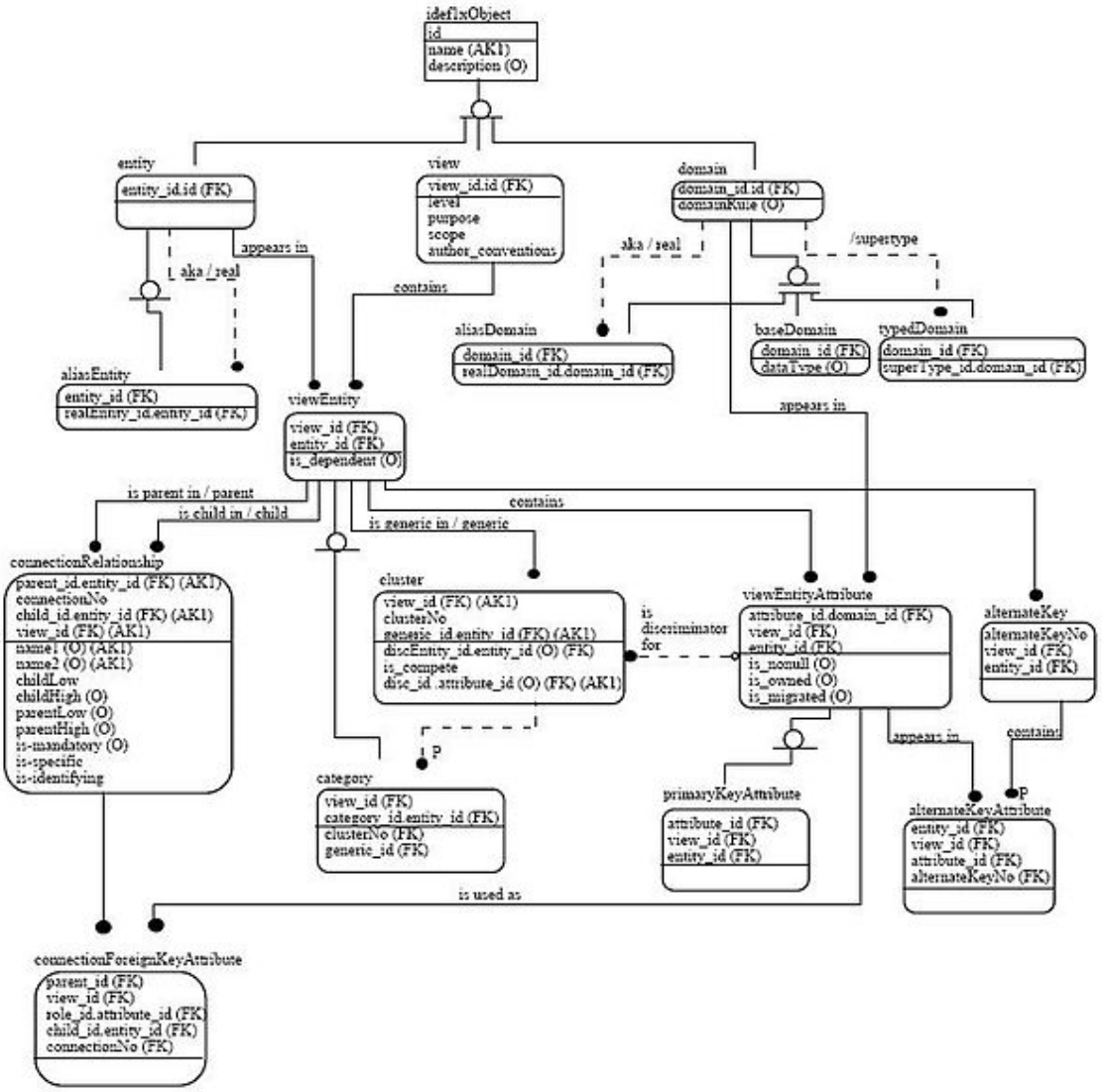
Data models represent information areas of interest. While there are many ways to create data models, according to Len Silverston (1997) only two modeling methodologies stand out, top-down and bottom-up:

- Bottom-up models are often the result of a reengineering effort. They usually start with existing data structures forms, fields on application screens, or reports. These models are usually physical, application-specific, and incomplete from an enterprise perspective. They may not promote data sharing, especially if they are built without reference to other parts of the organization.
- Top-down logical data models, on the other hand, are created in an abstract way by getting information from people who know the subject area. A system may not

implement all the entities in a logical model, but the model serves as a reference point or template.

Sometimes models are created in a mixture of the two methods: by considering the data needs and structure of an application and by consistently referencing a subject-area model. Unfortunately, in many environments the distinction between a logical data model and a physical data model is blurred. In addition, some CASE tools don't make a distinction between logical and physical data models.

Entity relationship diagrams



Example of a IDEF1X Entity relationship diagrams used to model IDEF1X itself. The name of the view is mm. The domain hierarchy and constraints are also given. The constraints are expressed as sentences in the formal theory of the meta model.

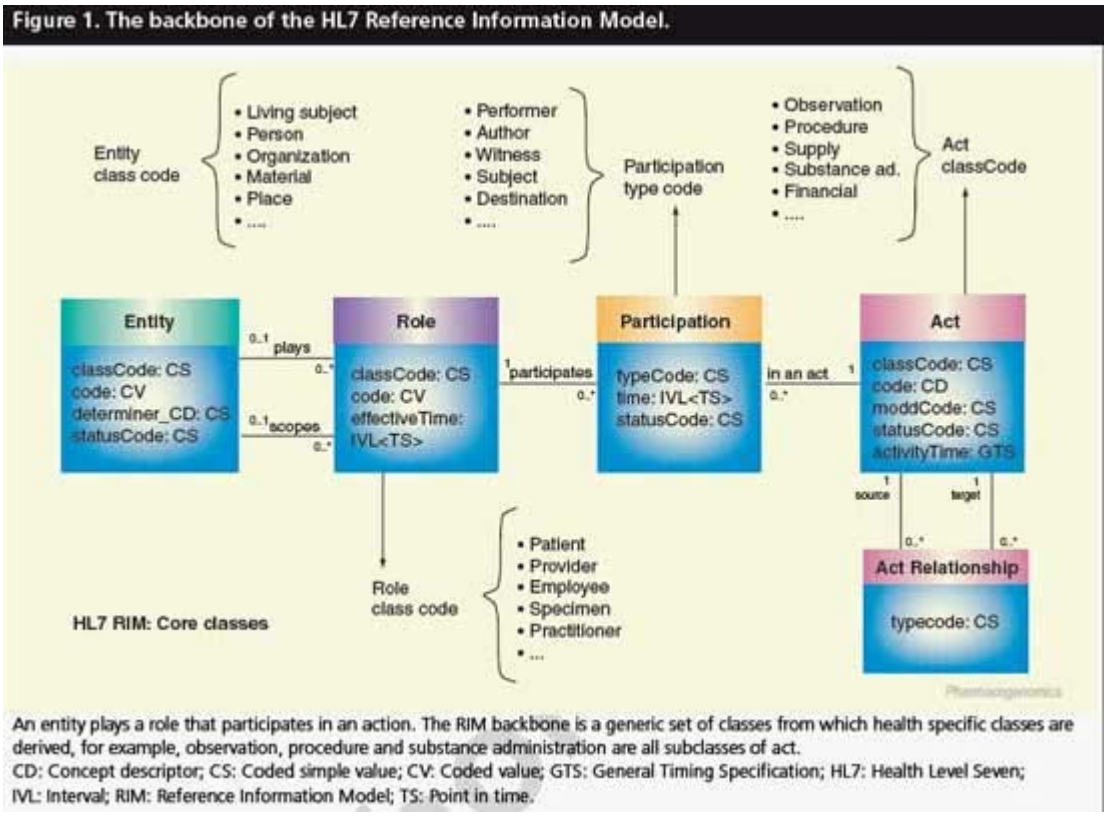
There are several notations for data modeling. The actual model is frequently called "Entity relationship model", because it depicts data in terms of the entities and relationships described in the data. An entity-relationship model (ERM) is an abstract conceptual representation of structured data. Entity-relationship modeling is a relational schema database modeling method, used in software engineering to produce a type of conceptual data model (or semantic data model) of a system, often a relational database, and its requirements in a top-down fashion.

These models are being used in the first stage of information system design during the requirements analysis to describe information needs or the type of information that is to be stored in a database. The data modeling technique can be used to describe any ontology (i.e. an overview and classifications of used terms and their relationships) for a certain universe of discourse i.e. area of interest.

Several techniques have been developed for the design of data models. While these methodologies guide data modelers in their work, two different people using the same methodology will often come up with very different results. Most notable are:

- Bachman diagrams
- Barker's Notation
- Chen's Notation
- Data Vault Modeling
- Extended Backus–Naur form
- IDEF1X
- Object-relational mapping
- Object Role Modeling
- Relational Model

Generic data modeling



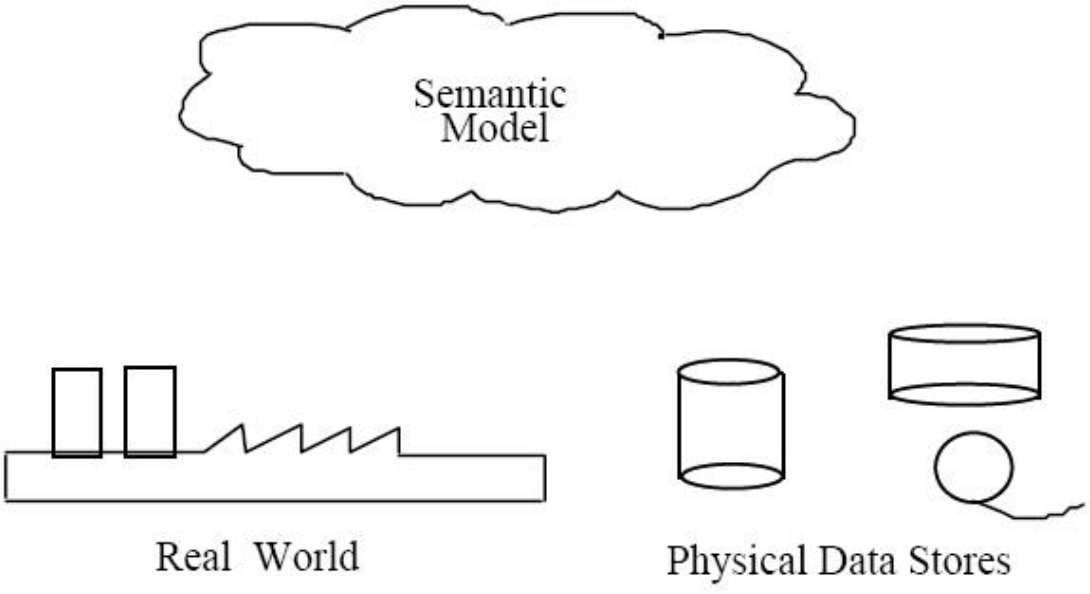
Example of a Generic data model.

Generic data models are generalizations of conventional data models. They define standardized general relation types, together with the kinds of things that may be related by such a relation type. The definition of generic data model is similar to the definition of a natural language. For example, a generic data model may define relation types such as a 'classification relation', being a binary relation between an individual thing and a kind of thing (a class) and a 'part-whole relation', being a binary relation between two things, one with the role of part, the other with the role of whole, regardless the kind of things that are related.

Given an extensible list of classes, this allows the classification of any individual thing and to specify part-whole relations for any individual object. By standardization of an extensible list of relation types, a generic data model enables the expression of an unlimited number of kinds of facts and will approach the capabilities of natural languages. Conventional data models, on the other hand, have a fixed and limited domain scope, because the instantiation (usage) of such a model only allows expressions of kinds of facts that are predefined in the model.

Semantic data modeling

The logical data structure of a DBMS, whether hierarchical, network, or relational, cannot totally satisfy the requirements for a conceptual definition of data because it is limited in scope and biased toward the implementation strategy employed by the DBMS.



Semantic data models.

Therefore, the need to define data from a conceptual view has led to the development of semantic data modeling techniques. That is, techniques to define the meaning of data within the context of its interrelationships with other data. As illustrated in the figure the real world, in terms of resources, ideas, events, etc., are symbolically defined within physical data stores. A semantic data model is an abstraction which defines how the stored symbols relate to the real world. Thus, the model must be a true representation of the real world.

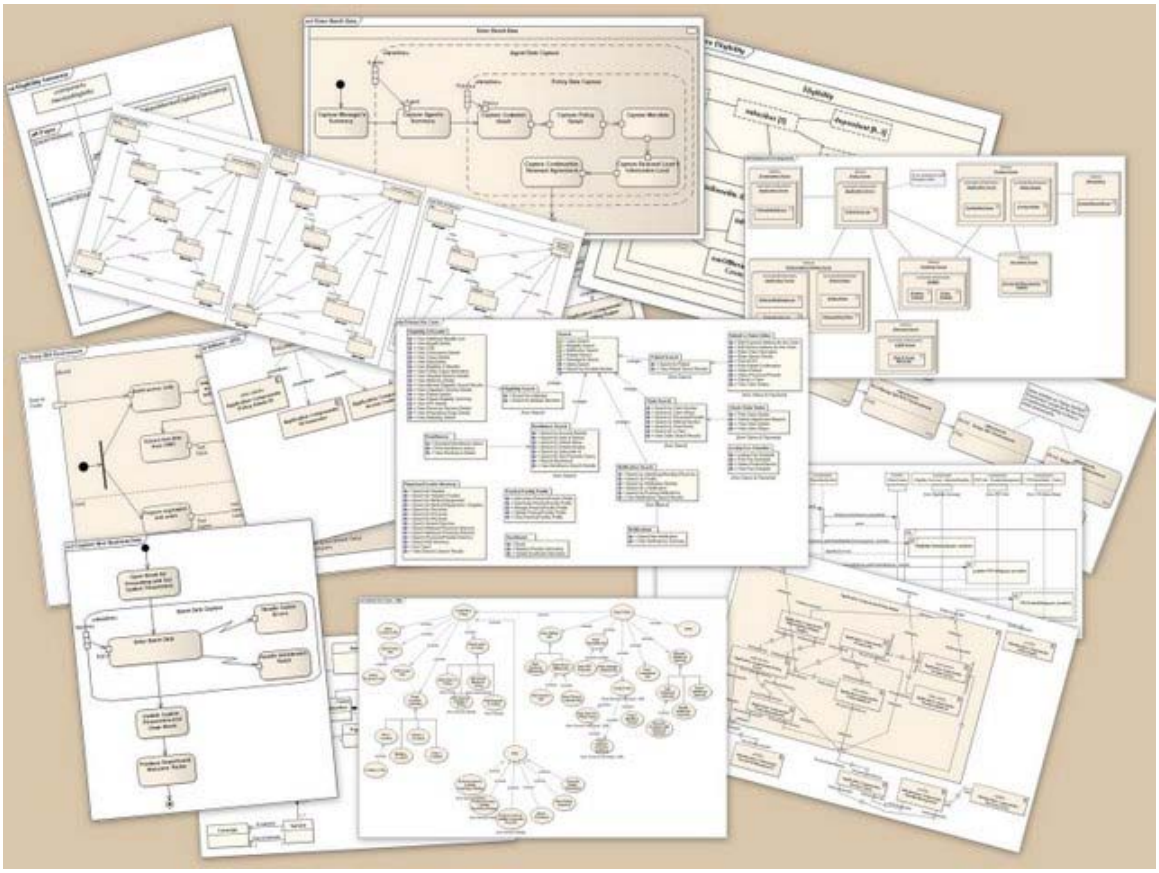
A semantic data model can be used to serve many purposes, such as:.

- planning of data resources
- building of shareable databases
- evaluation of vendor software
- integration of existing databases

The overall goal of semantic data models is to capture more meaning of data by integrating relational concepts with more powerful abstraction concepts known from the Artificial Intelligence field. The idea is to provide high level modeling primitives as integral part of a data model in order to facilitate the representation of real world situations.

Chapter 4

Unified Modeling Language



A collage of UML diagrams.

Unified Modeling Language (**UML**) is a standardized general-purpose modeling language in the field of object-oriented software engineering. The standard is managed, and was created by, the Object Management Group.

UML includes a set of graphic notation techniques to create visual models of object-oriented software-intensive systems.

Overview

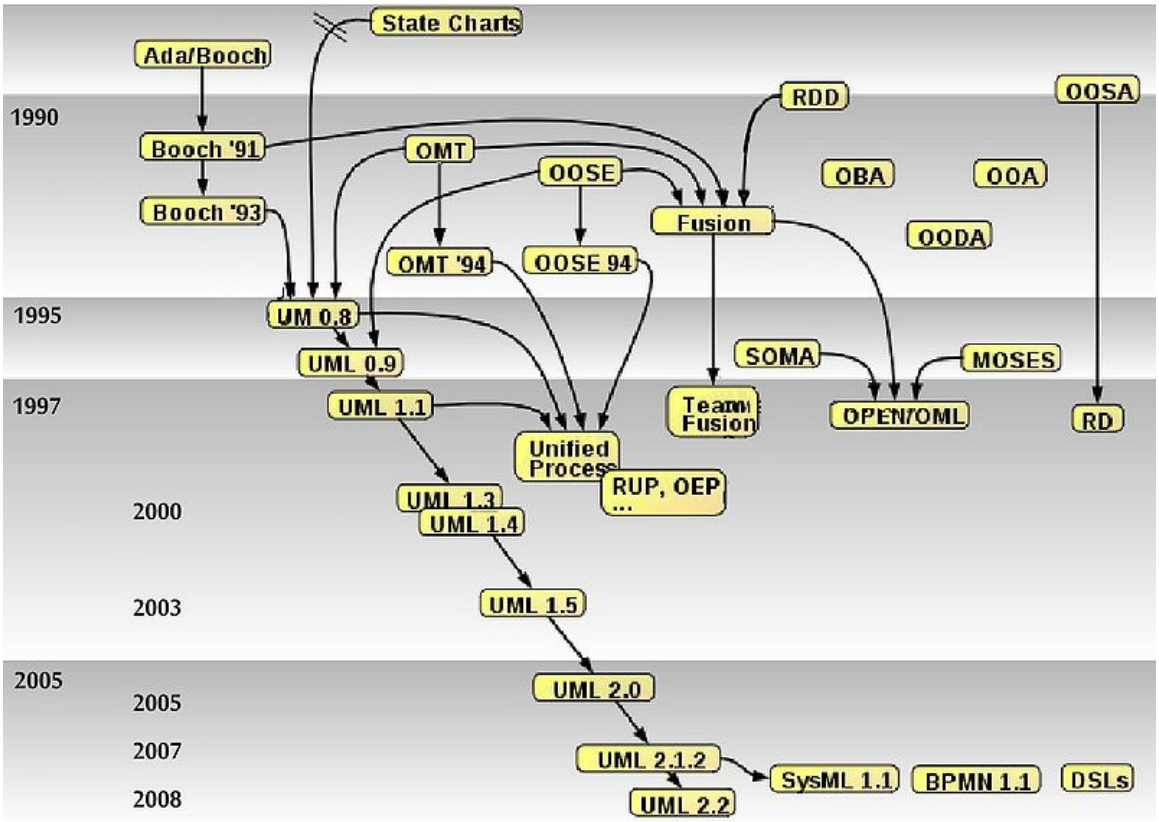
The Unified Modeling Language (UML) is used to specify, visualize, modify, construct and document the artifacts of an object-oriented software-intensive system under development. UML offers a standard way to visualize a system's architectural blueprints, including elements such as:

- activities
- actors
- business processes
- database schemas
- (logical) components
- programming language statements
- reusable software components.

UML combines techniques from data modeling (entity relationship diagrams), business modeling (work flows), object modeling, and component modeling. It can be used with all processes, throughout the software development life cycle, and across different implementation technologies. UML has synthesized the notations of the Booch method, the Object-modeling technique (OMT) and Object-oriented software engineering (OOSE) by fusing them into a single, common and widely usable modeling language. UML aims to be a standard modeling language which can model concurrent and distributed systems. UML is a de facto industry standard, and is evolving under the auspices of the Object Management Group (OMG).

UML models may be automatically transformed to other representations (e.g. Java) by means of QVT-like transformation languages. UML is extensible, with two mechanisms for customization: profiles and stereotypes.

History



History of object-oriented methods and notation.

Before UML 1.x

After Rational Software Corporation hired James Rumbaugh from General Electric in 1994, the company became the source for the two most popular object-oriented modeling approaches of the day: Rumbaugh's Object-modeling technique (OMT), which was better for object-oriented analysis (OOA), and Grady Booch's Booch method, which was better for object-oriented design (OOD). They were soon assisted in their efforts by Ivar Jacobson, the creator of the object-oriented software engineering (OOSE) method. Jacobson joined Rational in 1995, after his company, Objectory AB, was acquired by Rational. The three methodologists were collectively referred to as the *Three Amigos*.

In 1996 Rational concluded that the abundance of modeling languages was slowing the adoption of object technology, so repositioning the work on an unified method, they tasked the Three Amigos with the development of a non-proprietary Unified Modeling Language. Representatives of competing object technology companies were consulted during OOPSLA '96; they chose *boxes* for representing classes rather than the *cloud* symbols that were used in Booch's notation.

Under the technical leadership of the Three Amigos, an international consortium called the UML Partners was organized in 1996 to complete the *Unified Modeling Language*

(UML) specification, and propose it as a response to the OMG RFP. The UML Partners' UML 1.0 specification draft was proposed to the OMG in January 1997. During the same month the UML Partners formed a Semantics Task Force, chaired by Cris Kobryn and administered by Ed Eykholt, to finalize the semantics of the specification and integrate it with other standardization efforts. The result of this work, UML 1.1, was submitted to the OMG in August 1997 and adopted by the OMG in November 1997.

UML 1.x

As a modeling notation, the influence of the OMT notation dominates (e. g., using rectangles for classes and objects). Though the Booch "cloud" notation was dropped, the Booch capability to specify lower-level design detail was embraced. The use case notation from Objectory and the component notation from Booch were integrated with the rest of the notation, but the semantic integration was relatively weak in UML 1.1, and was not really fixed until the UML 2.0 major revision.

Concepts from many other OO methods were also loosely integrated with UML with the intent that UML would support all OO methods. Many others also contributed, with their approaches flavouring the many models of the day, including: Tony Wasserman and Peter Pircher with the "Object-Oriented Structured Design (OOSD)" notation (not a method), Ray Buhr's "Systems Design with Ada", Archie Bowen's use case and timing analysis, Paul Ward's data analysis and David Harel's "Statecharts"; as the group tried to ensure broad coverage in the real-time systems domain. As a result, UML is useful in a variety of engineering problems, from single process, single user applications to concurrent, distributed systems, making UML rich but also large.

The Unified Modeling Language is an international standard:

ISO/IEC 19501:2005 Information technology – Open Distributed Processing – Unified Modeling Language (UML) Version 1.4.2

UML 2.x

UML has matured significantly since UML 1.1. Several minor revisions (UML 1.3, 1.4, and 1.5) fixed shortcomings and bugs with the first version of UML, followed by the UML 2.0 major revision that was adopted by the OMG in 2005.

Although UML 2.1 was never released as a formal specification, versions 2.1.1 and 2.1.2 appeared in 2007, followed by UML 2.2 in February 2009. UML 2.3 was formally released in May 2010.

There are four parts to the UML 2.x specification:

1. The Superstructure that defines the notation and semantics for diagrams and their model elements

2. The Infrastructure that defines the core metamodel on which the Superstructure is based
3. The Object Constraint Language (OCL) for defining rules for model elements
4. The UML Diagram Interchange that defines how UML 2 diagram layouts are exchanged

The current versions of these standards follow: UML Superstructure version 2.3, UML Infrastructure version 2.3, OCL version 2.2, and UML Diagram Interchange version 1.0.

Although many UML tools support some of the new features of UML 2.x, the OMG provides no test suite to objectively test compliance with its specifications.

Topics

Software development methods

UML is not a development method by itself; however, it was designed to be compatible with the leading object-oriented software development methods of its time (for example OMT, Booch method, Objectory). Since UML has evolved, some of these methods have been recast to take advantage of the new notations (for example OMT), and new methods have been created based on UML, such as IBM Rational Unified Process (RUP). Others include Abstraction Method and Dynamic Systems Development Method.

Modeling

It is important to distinguish between the UML model and the set of diagrams of a system. A diagram is a partial graphic representation of a system's model. The model also contains documentation that drive the model elements and diagrams (such as written use cases).

UML diagrams represent two different views of a system model:

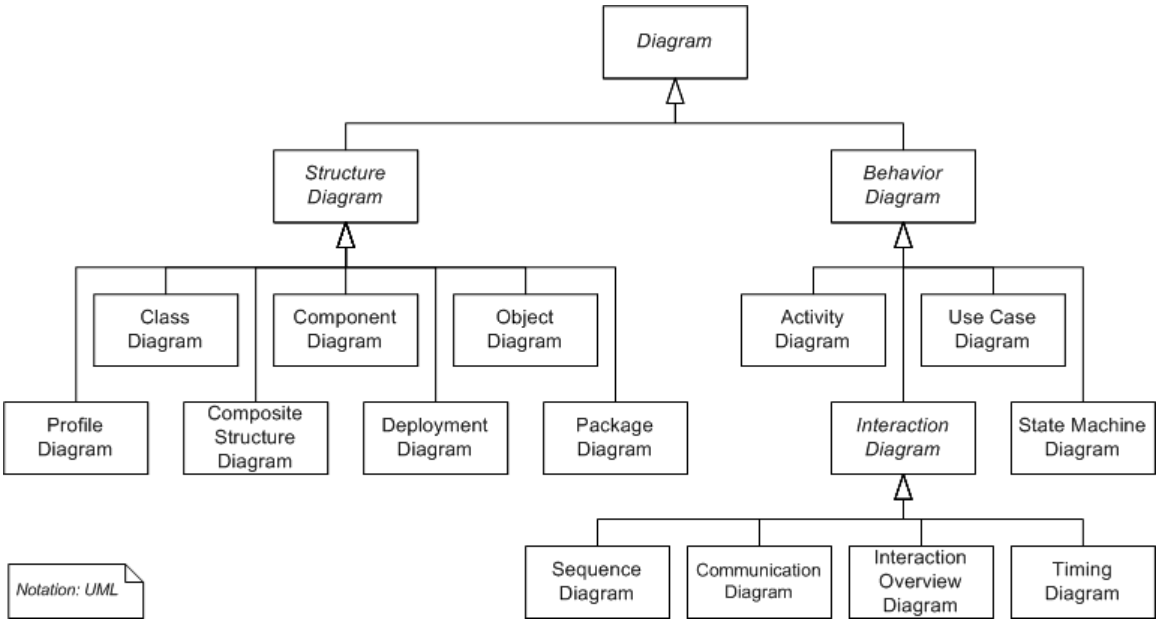
- Static (or *structural*) view: emphasizes the static structure of the system using objects, attributes, operations and relationships. The structural view includes class diagrams and composite structure diagrams.
- Dynamic (or *behavioral*) view: emphasizes the dynamic behavior of the system by showing collaborations among objects and changes to the internal states of objects. This view includes sequence diagrams, activity diagrams and state machine diagrams.

UML models can be exchanged among UML tools by using the XMI interchange format.

Diagrams overview

UML 2.2 has 14 types of diagrams divided into two categories. Seven diagram types represent *structural* information, and the other seven represent general types of *behavior*,

including four that represent different aspects of *interactions*. These diagrams can be categorized hierarchically as shown in the following class diagram:



UML does not restrict UML element types to a certain diagram type. In general, every UML element may appear on almost all types of diagrams; this flexibility has been partially restricted in UML 2.0. UML profiles may define additional diagram types or extend existing diagrams with additional notations.

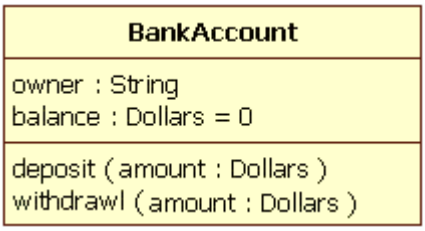
In keeping with the tradition of engineering drawings, a comment or note explaining usage, constraint, or intent is allowed in a UML diagram.

Structure diagrams

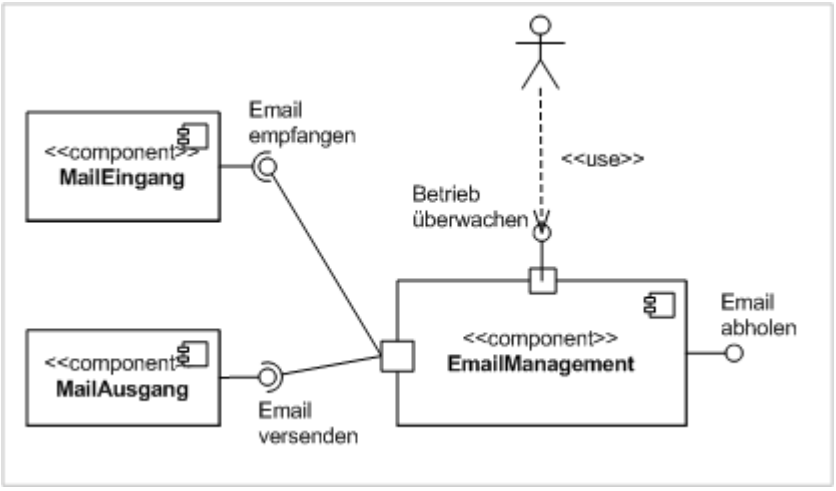
Structure diagrams emphasize the things that must be present in the system being modeled. Since structure diagrams represent the structure, they are used extensively in documenting the software architecture of software systems.

- Class diagram: describes the structure of a system by showing the system's classes, their attributes, and the relationships among the classes.
- Component diagram: describes how a software system is split up into components and shows the dependencies among these components.
- Composite structure diagram: describes the internal structure of a class and the collaborations that this structure makes possible.
- Deployment diagram: describes the hardware used in system implementations and the execution environments and artifacts deployed on the hardware.
- Object diagram: shows a complete or partial view of the structure of a modeled system at a specific time.
- Package diagram: describes how a system is split up into logical groupings by showing the dependencies among these groupings.

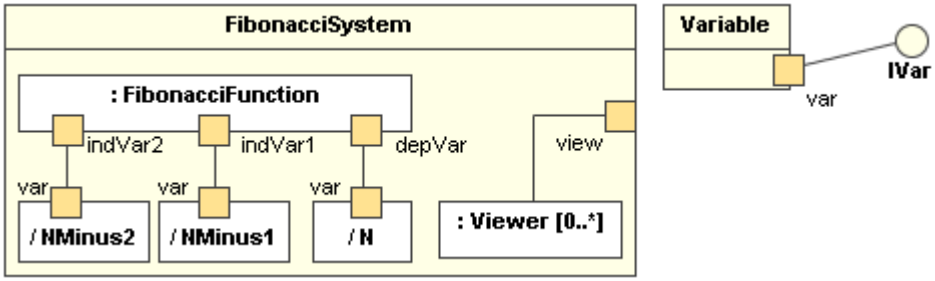
- Profile diagram: operates at the metamodel level to show stereotypes as classes with the <<stereotype>> stereotype, and profiles as packages with the <<profile>> stereotype. The extension relation (solid line with closed, filled arrowhead) indicates what metamodel element a given stereotype is extending.



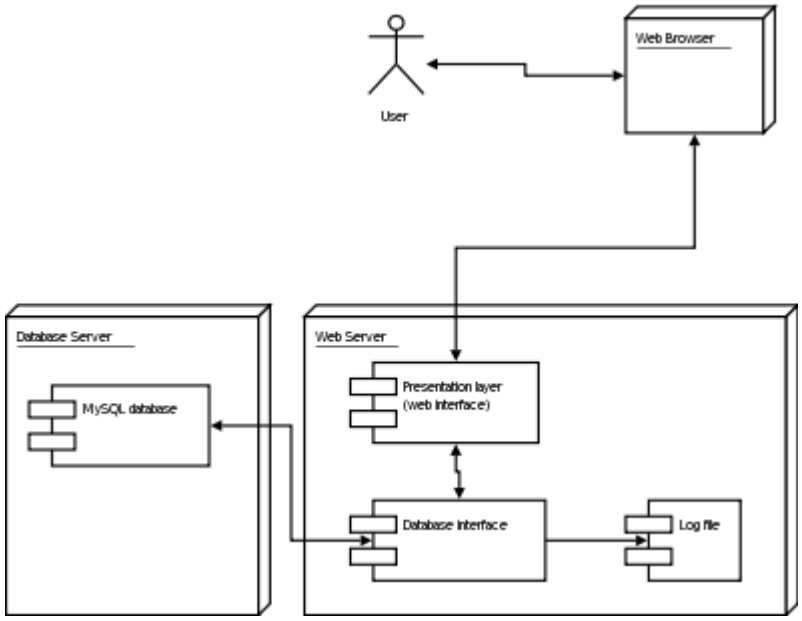
Class diagram



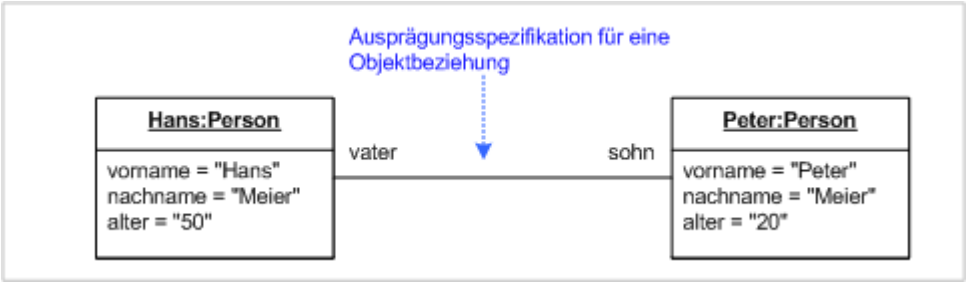
Component diagram



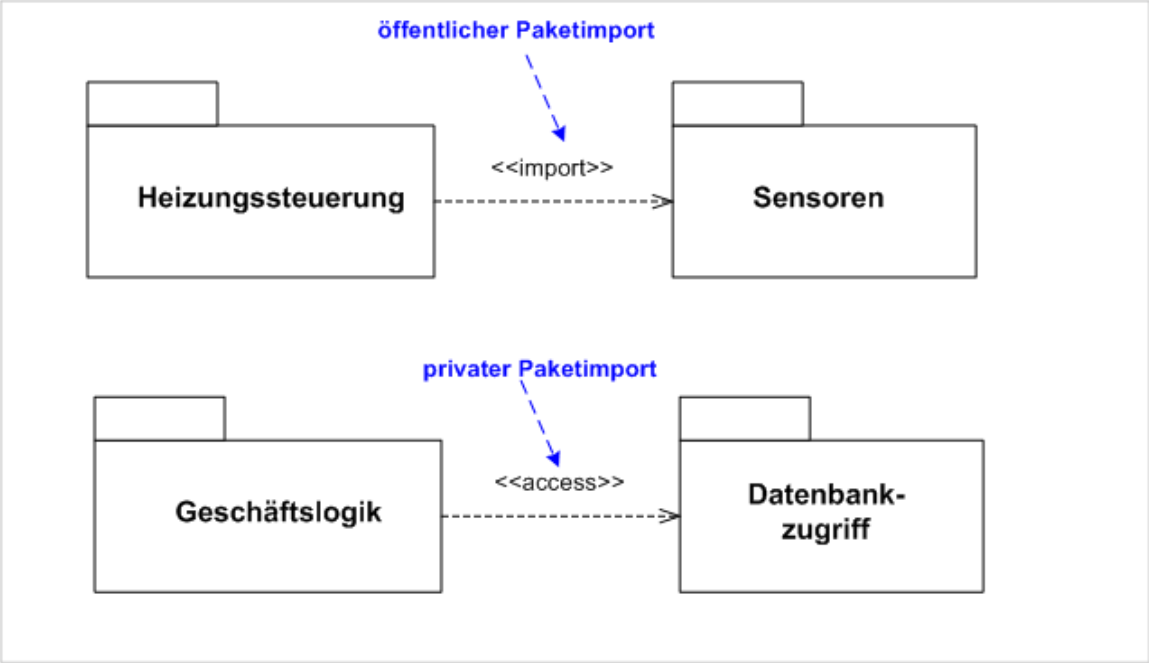
Composite structure diagrams



Deployment diagram



Object diagram

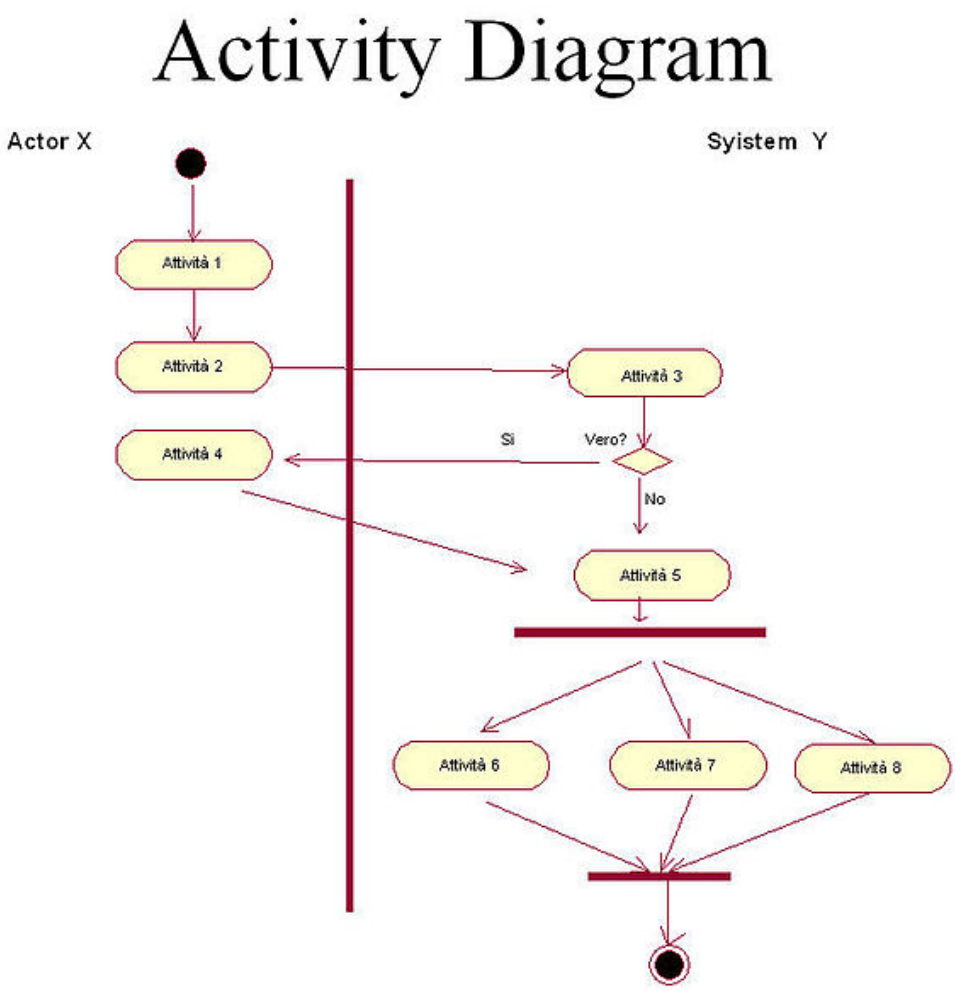


Package diagram

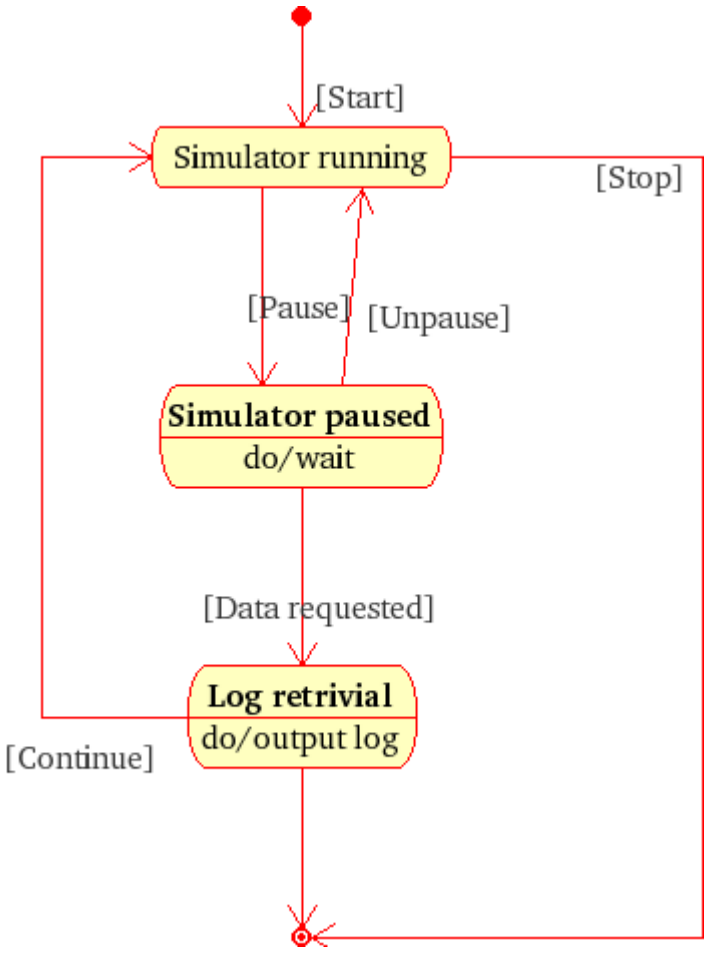
Behaviour diagrams

Behavior diagrams emphasize what must happen in the system being modeled. Since behavior diagrams illustrate the behavior of a system, they are used extensively to describe the functionality of software systems.

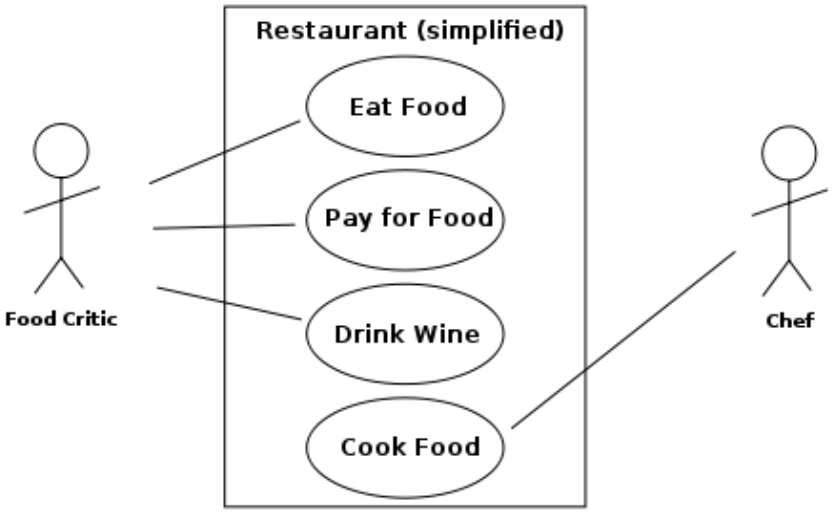
- Activity diagram: describes the business and operational step-by-step workflows of components in a system. An activity diagram shows the overall flow of control.
- UML state machine diagram: describes the states and state transitions of the system.
- Use case diagram: describes the functionality provided by a system in terms of actors, their goals represented as use cases, and any dependencies among those use cases.



UML Activity Diagram



State Machine diagram

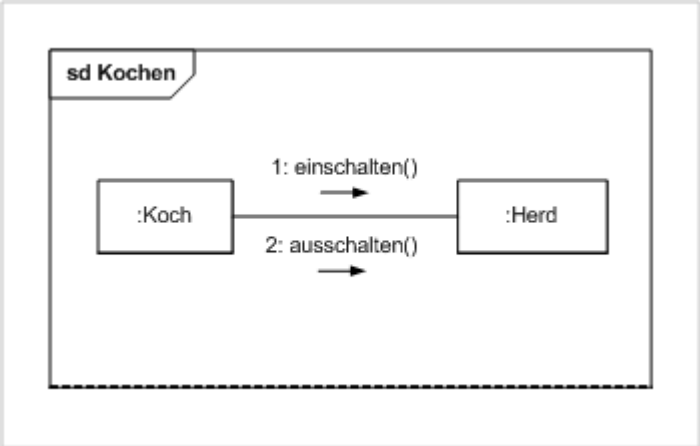


Use case diagram

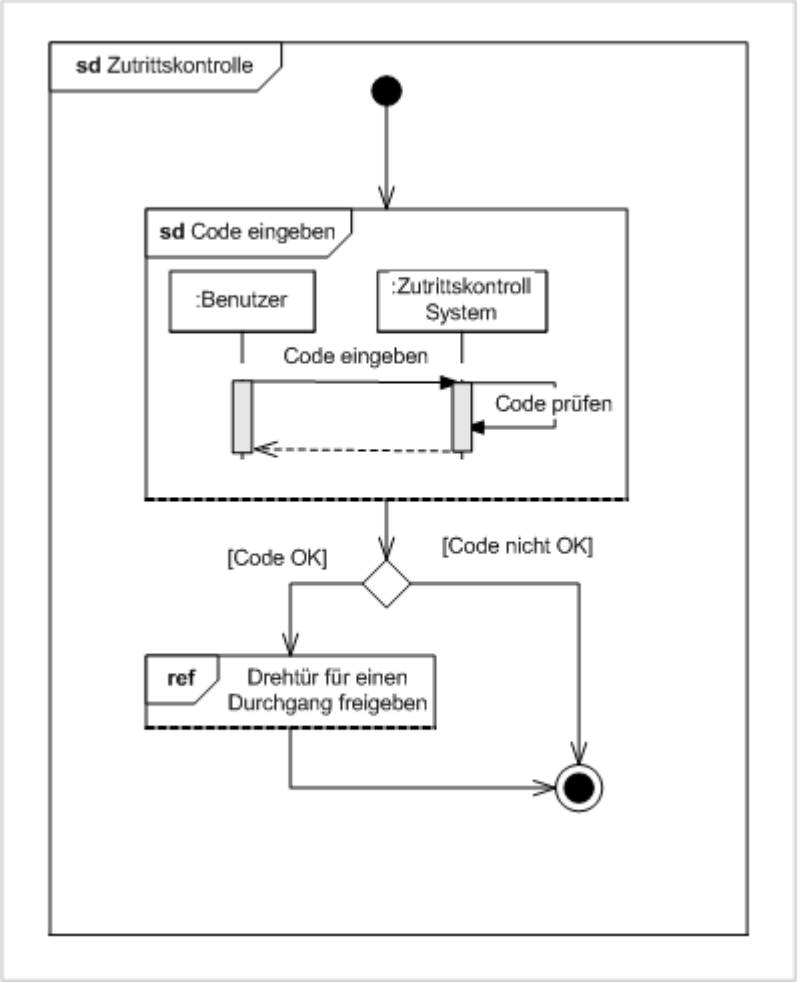
Interaction diagrams

Interaction diagrams, a subset of behaviour diagrams, emphasize the flow of control and data among the things in the system being modeled:

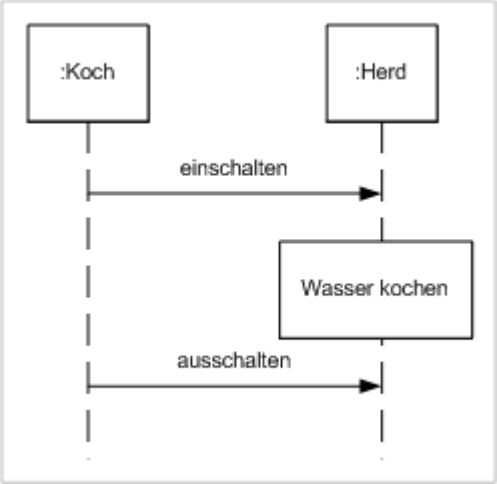
- Communication diagram: shows the interactions between objects or parts in terms of sequenced messages. They represent a combination of information taken from Class, Sequence, and Use Case Diagrams describing both the static structure and dynamic behavior of a system.
- Interaction overview diagram: provides an overview in which the nodes represent communication diagrams.
- Sequence diagram: shows how objects communicate with each other in terms of a sequence of messages. Also indicates the lifespans of objects relative to those messages.
- Timing diagrams: a specific type of interaction diagram where the focus is on timing constraints.



Communication diagram



Interaction overview diagram



Sequence diagram

The Protocol State Machine is a sub-variant of the State Machine. It may be used to model network communication protocols.

Meta modeling

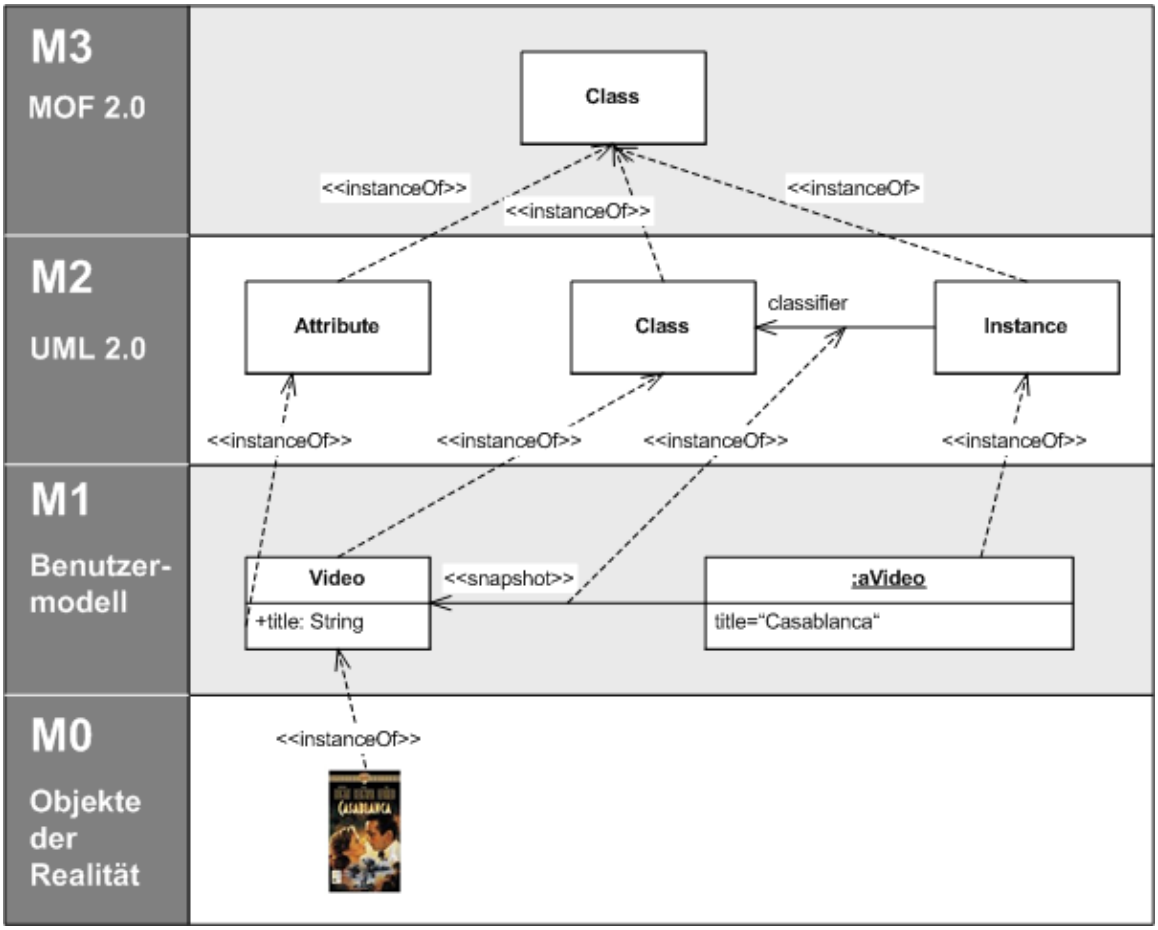


Illustration of the Meta-Object Facility.

The Object Management Group (OMG) has developed a metamodeling architecture to define the Unified Modeling Language (UML), called the Meta-Object Facility (MOF). The Meta-Object Facility is a standard for model-driven engineering, designed as a four-layered architecture, as shown in the image at right. It provides a meta-meta model at the top layer, called the M0 layer. This M0-model is the language used by Meta-Object Facility to build metamodels, called M1-models. The most prominent example of a Layer 1 Meta-Object Facility model is the UML metamodel, the model that describes the UML itself. These M1-models describe elements of the M2-layer, and thus M2-models. These would be, for example, models written in UML. The last layer is the M3-layer or data layer. It is used to describe runtime instance of the system.

Beyond the M0-model, the Meta-Object Facility describes the means to create and manipulate models and metamodels by defining CORBA interfaces that describe those operations. Because of the similarities between the Meta-Object Facility M0-model and

UML structure models, Meta-Object Facility metamodels are usually modeled as UML class diagrams. A supporting standard of the Meta-Object Facility is XMI, which defines an XML-based exchange format for models on the M0-, M1-, or M2-Layer.

Criticisms

Although UML is a widely recognized and used modeling standard, it is frequently criticized for the following:

Standards bloat
: Bertrand Meyer, in a satirical essay framed as a student's request for a grade change, apparently criticized UML as of 1997 for being unrelated to object-oriented software development; a disclaimer was added later pointing out that his company nevertheless supports UML. Ivar Jacobson, a co-architect of UML, said that objections to UML 2.0's size were valid enough to consider the application of intelligent agents to the problem. It contains many diagrams and constructs that are redundant or infrequently used.

Problems in learning and adopting
: The problems cited in this section make learning and adopting UML problematic, especially when required of engineers lacking the prerequisite skills. In practice, people often draw diagrams with the symbols provided by their CASE tool, but without the meanings those symbols are intended to provide.

Linguistic incoherence
: The extremely poor writing of the UML standards themselves—assumed to be the consequence of having been written by a non-native English speaker—seriously reduces their normative value. In this respect the standards have been widely cited, and indeed pilloried, as prime examples of unintelligible geekspeak.

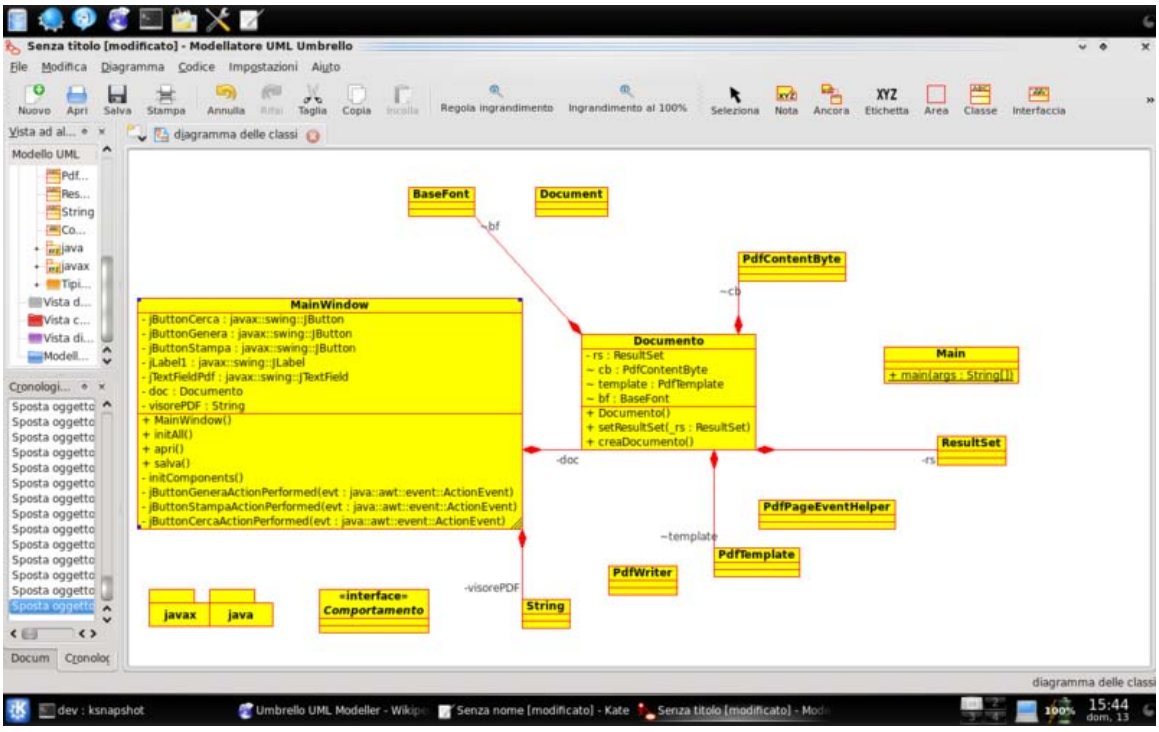
Capabilities of UML and implementation language mismatch
: As with any notational system, UML is able to represent some systems more concisely or efficiently than others. Thus a developer gravitates toward solutions that reside at the intersection of the capabilities of UML and the implementation language. This problem is particularly pronounced if the implementation language does not adhere to orthodox object-oriented doctrine, as the intersection set between UML and implementation language may be that much smaller.

Dysfunctional interchange format
: While the XMI (XML Metadata Interchange) standard is designed to facilitate the interchange of UML models, it has been largely ineffective in the practical interchange of UML 2.x models. This interoperability ineffectiveness is attributable to two reasons. Firstly, XMI 2.x is large and complex in its own right, since it purports to address a technical problem more ambitious than exchanging UML 2.x models. In particular, it attempts to provide a mechanism for facilitating the exchange of any arbitrary modeling language defined by the OMG's Meta-Object Facility (MOF). Secondly, the UML 2.x Diagram Interchange specification lacks sufficient detail to facilitate reliable interchange of UML 2.x notations between modeling tools. Since UML is a visual modeling language, this shortcoming is substantial for modelers who don't want to redraw their diagrams.

Modeling experts have written sharp criticisms of UML, including Bertrand Meyer's "UML: The Positive Spin", and Brian Henderson-Sellers and Cesar Gonzalez-Perez in "Uses and Abuses of the Stereotype Mechanism in UML 1.x and 2.0".

UML modelling tools

The most well-known UML modelling tool is IBM Rational Rose. Other tools include Rational Rhapsody, MagicDraw UML, StarUML, ArgoUML, Umbrello, BOUML, PowerDesigner, and Dia. Some of popular development environments also offer UML modelling tools, e.g.: Eclipse, NetBeans, and Visual Studio.



Screenshot of Umbrello UML Modeller.

Chapter 5

Hardware Description Language

In electronics, a **hardware description language** or **HDL** is any language from a class of computer languages, specification languages, or modeling languages for formal description and design of electronic circuits, and most-commonly, digital logic. It can describe the circuit's operation, its design and organization, and tests to verify its operation by means of simulation.

HDLs are standard text-based expressions of the spatial and temporal structure and behaviour of electronic systems. Like concurrent programming languages, HDL syntax and semantics includes explicit notations for expressing concurrency. However, in contrast to most software programming languages, HDLs also include an explicit notion of time, which is a primary attribute of hardware. Languages whose only characteristic is to express circuit connectivity between a hierarchy of blocks are properly classified as netlist languages used on electric computer-aided design (CAD).

HDLs are used to write executable specifications of some piece of hardware. A simulation program, designed to implement the underlying semantics of the language statements, coupled with simulating the progress of time, provides the hardware designer with the ability to model a piece of hardware before it is created physically. It is this executability that gives HDLs the illusion of being programming languages, when they are more-precisely classed as specification languages or modeling languages. Simulators capable of supporting discrete-event (digital) and continuous-time (analog) modeling exist, and HDLs targeted for each are available.

It is certainly possible to represent hardware semantics using traditional programming languages such as C++, although to function such programs must be augmented with extensive and unwieldy class libraries. Primarily, however, software programming languages do not include any capability for explicitly expressing time, and this is why they do not function as a hardware description language. Before the recent introduction of SystemVerilog, C++ integration with a logic simulator was one of the few ways to use

OOP in hardware verification. SystemVerilog is the first major HDL to offer object orientation and garbage collection.

Using the proper subset of virtually any (hardware description or software programming) language, a software program called a synthesizer (or synthesis tool) can infer hardware logic operations from the language statements and produce an equivalent netlist of generic hardware primitives to implement the specified behaviour. Synthesizers generally ignore the expression of any timing constructs in the text. Digital logic synthesizers, for example, generally use clock edges as the way to time the circuit, ignoring any timing constructs. The ability to have a synthesizable subset of the language does not itself make a hardware description language.

History

The first hardware description languages were ISP (Instruction Set Processor), developed at Carnegie Mellon University, and KARL, developed at University of Kaiserslautern, both around 1977. ISP was, however, more like a software programming language used to describe relations between the inputs and the outputs of the design. Therefore, it could be used to simulate the design, but not to synthesize it. KARL included design calculus language features supporting VLSI chip floorplanning and structured hardware design, which was also the basis of KARL's interactive graphic sister language ABL, implemented in the early 1980s as the ABLED graphic VLSI design editor, by the telecommunication research center CSELT at Torino, Italy. In the mid 80's, a VLSI design framework was implemented around KARL and ABL by an international consortium funded by the commission of the European Union (chapter in). In 1983 Data-I/O introduced ABEL. It was targeted for describing programmable logical devices and was basically used to design finite state machines.

The first modern HDL, Verilog, was introduced by Gateway Design Automation in 1985. Cadence Design Systems later acquired the rights to Verilog-XL, the HDL-simulator that would become the de-facto standard (of Verilog simulators) for the next decade. In 1987, a request from the U.S. Department of Defense led to the development of VHDL (VHSIC Hardware Description Language, where VHSIC is Very High Speed Integrated Circuit). VHDL was based on the Ada programming language. Initially, Verilog and VHDL were used to document and simulate circuit-designs already captured and described in another form (such as a schematic file.) HDL-simulation enabled engineers to work at a higher level of abstraction than simulation at the schematic-level, and thus increased design capacity from hundreds of transistors to thousands.

The introduction of logic-synthesis for HDLs pushed HDLs from the background into the foreground of digital-design. Synthesis tools compiled HDL-source files (written in a constrained format called RTL) into a manufacturable gate/transistor-level netlist description. Writing synthesizeable RTL files required practice and discipline on the part of the designer; compared to a traditional schematic-layout, synthesized-RTL netlists were almost always larger in area and slower in performance. Circuit design by a skilled engineer, using labor-intensive schematic-capture/hand-layout, would almost always

outperform its logically-synthesized equivalent, but synthesis's productivity advantage soon displaced digital schematic-capture to exactly those areas that were problematic for RTL-synthesis: extremely high-speed, low-power, or asynchronous circuitry. In short, logic synthesis propelled HDL technology into a central role for digital design.

Within a few years, both VHDL and Verilog emerged as the dominant HDLs in the electronics industry, while older and less-capable HDLs gradually disappeared from use. But VHDL and Verilog share many of the same limitations: neither HDL is suitable for analog/mixed-signal circuit simulation. Neither possesses language constructs to describe recursively-generated logic structures. Specialized HDLs (such as Confluence) were introduced with the explicit goal of fixing a specific Verilog/VHDL limitation, though none were ever intended to replace VHDL/Verilog.

Over the years, a lot of effort has gone into improving HDLs. The latest iteration of Verilog, formally known as IEEE 1800-2005 SystemVerilog, introduces many new features (classes, random variables, and properties/assertions) to address the growing need for better testbench randomization, design hierarchy, and reuse. A future revision of VHDL is also in development, and is expected to match SystemVerilog's improvements.

Design using HDL

Efficiency gains realized using HDL means a majority of modern digital circuit design revolves around it. Most designs begin as a set of requirements or a high-level architectural diagram. Control and decision structures are often prototyped in flowchart applications, or entered in a state-diagram editor. The process of writing the HDL description is highly dependent on the nature of the circuit and the designer's preference for coding style . The HDL is merely the 'capture language'—often beginning with a high-level algorithmic description such as MATLAB or a C++ mathematical model. Designers often use scripting languages (such as Perl) to automatically generate repetitive circuit structures in the HDL language. Special text editors offer features for automatic indentation, syntax-dependent coloration, and macro-based expansion of entity/architecture/signal declaration.

The HDL code then undergoes a code review, or auditing. In preparation for synthesis, the HDL description is subject to an array of automated checkers. The checkers report deviations from standardized code guidelines, identify potential ambiguous code constructs before they can cause misinterpretation, and check for common logical coding errors, such as dangling ports or shorted outputs. This process aids in resolving errors before the code is synthesized.

In industry parlance, HDL design generally ends at the synthesis stage. Once the synthesis tool has mapped the HDL description into a gate netlist, this netlist is passed off to the back-end stage. Depending on the physical technology (FPGA, ASIC gate array, ASIC standard cell), HDLs may or may not play a significant role in the back-end flow. In general, as the design flow progresses toward a physically realizable form, the design database becomes progressively more laden with technology-specific information, which

cannot be stored in a generic HDL description. Finally, an integrated circuit is manufactured or programmed for use.

Simulating and debugging HDL code

Essential to HDL design is the ability to simulate HDL programs. Simulation allows an HDL description of a design (called a model) to pass design verification, an important milestone that validates the design's intended function (specification) against the code implementation in the HDL description. It also permits architectural exploration. The engineer can experiment with design choices by writing multiple variations of a base design, then comparing their behavior in simulation. Thus, simulation is critical for successful HDL design.

To simulate an HDL model, an engineer writes a top-level simulation environment (called a testbench). At minimum, a testbench contains an instantiation of the model (called the device under test or DUT), pin/signal declarations for the model's I/O, and a clock waveform. The testbench code is event driven: the engineer writes HDL statements to implement the (testbench-generated) reset-signal, to model interface transactions (such as a host–bus read/write), and to monitor the DUT's output. An HDL simulator — the program that executes the testbench — maintains the simulator clock, which is the master reference for all events in the testbench simulation. Events occur only at the instants dictated by the testbench HDL (such as a reset-toggle coded into the testbench), or in reaction (by the model) to stimulus and triggering events. Modern HDL simulators have a full-featured graphical user interfaces, complete with a suite of debug tools. These allow the user to stop and restart the simulation at any time, insert simulator breakpoints (independent of the HDL code), and monitor or modify any element in the HDL model hierarchy. Modern simulators can also link the HDL environment to user-compiled libraries, through a defined PLI/VHPI interface. Linking is system-dependent (Win32/Linux/SPARC), as the HDL simulator and user libraries are compiled and linked outside the HDL environment.

Design verification is often the most time-consuming portion of the design process, due to the disconnect between a device's functional specification, the designer's interpretation of the specification, and the imprecision of the HDL language. The majority of the initial test/debug cycle is conducted in the HDL *simulator* environment, as the early stage of the design is subject to frequent and major circuit changes. An HDL description can also be prototyped and tested in hardware — programmable logic devices are often used for this purpose. Hardware prototyping is comparatively more expensive than HDL simulation, but offers a real-world view of the design. Prototyping is the best way to check interfacing against other hardware devices and hardware prototypes. Even those running on slow FPGAs offer much faster simulation times than pure HDL simulation.

Design Verification with HDLs

Historically, design verification was a laborious, repetitive loop of writing and running simulation test cases against the design under test. As chip designs have grown larger and

more complex, the task of design verification has grown to the point where it now dominates the schedule of a design team. Looking for ways to improve design productivity, the EDA industry developed the Property Specification Language.

In formal verification terms, a property is a factual statement about the expected or assumed behavior of another object. Ideally, for a given HDL description, a property or properties can be proven true or false using formal mathematical methods. In practical terms, many properties cannot be proven because they occupy an unbounded solution space. However, if provided a set of operating assumptions or constraints, a property checker can prove (or disprove) more properties, over the narrowed solution space.

The assertions do not model circuit activity, but capture and document the "designer's intent" in the HDL code. In a simulation environment, the simulator evaluates all specified assertions, reporting the location and severity of any violations. In a synthesis environment, the synthesis tool usually operates with the policy of halting synthesis upon any violation. Assertion-based verification is still in its infancy, but is expected to become an integral part of the HDL design toolset.

HDL and programming languages

A HDL is analogous to a software programming language, but with major differences. Many programming languages are inherently procedural (single-threaded), with limited syntactical and semantic support to handle concurrency. HDLs, on the other hand, resemble concurrent programming languages in their ability to model multiple parallel processes (such as flipflops, adders, etc.) that automatically execute independently of one another. Any change to the process's input automatically triggers an update in the simulator's process stack. Both programming languages and HDLs are processed by a compiler (usually called a synthesizer in the HDL case), but with different goals. For HDLs, 'compiler' refers to synthesis, a process of transforming the HDL code listing into a physically realizable gate netlist. The netlist output can take any of many forms: a "simulation" netlist with gate-delay information, a "handoff" netlist for post-synthesis place and route, or a generic industry-standard EDIF format (for subsequent conversion to a JEDEC-format file).

On the other hand, a software compiler converts the source-code listing into a microprocessor-specific object-code, for execution on the target microprocessor. As HDLs and programming languages borrow concepts and features from each other, the boundary between them is becoming less distinct. However, pure HDLs are unsuitable for general purpose software application development, just as general-purpose programming languages are undesirable for modeling hardware. Yet as electronic systems grow increasingly complex, and reconfigurable systems become increasingly mainstream, there is growing desire in the industry for a single language that can perform some tasks of both hardware design and software programming. SystemC is an example of such—embedded system hardware can be modeled as non-detailed architectural blocks (blackboxes with modeled signal inputs and output drivers). The target application is written in C/C++, and natively compiled for the host-development system (as opposed to

targeting the embedded CPU, which requires host-simulation of the embedded CPU). The high level of abstraction of SystemC models is well suited to early architecture exploration, as architectural modifications can be easily evaluated with little concern for signal-level implementation issues. However, the threading model used in SystemC and its reliance on shared memory mean that it does not handle parallel execution or lower level models well.

In an attempt to reduce the complexity of designing in HDLs, which have been compared to the equivalent of assembly languages, there are moves to raise the abstraction level of the design. Companies such as Cadence, Synopsys and Agility Design Solutions are promoting SystemC as a way to combine high level languages with concurrency models to allow faster design cycles for FPGAs than is possible using traditional HDLs. Approaches based on standard C or C++ (with libraries or other extensions allowing parallel programming) are found in the Catapult C tools from Mentor Graphics, the Impulse C tools from Impulse Accelerated Technologies, and the free and open-source ROCCC 2.0 tools from Jacquard Computing Inc. Annapolis Micro Systems, Inc.'s CoreFire Design Suite and National Instruments LabVIEW FPGA provide a graphical dataflow approach to high-level design entry. Languages such as SystemVerilog, SystemVHDL, and Handel-C seek to accomplish the same goal, but are aimed at making existing hardware engineers more productive versus making FPGAs more accessible to existing software engineers. There is more information on C to HDL and Flow to HDL in their respective articles.

Languages

Analogue circuit design

Abbreviation	Name	Use
AHDL	Analog Hardware Descriptive Language (HDL)	an open analog hardware description language
SpectreHDL	SpectreHDL	a proprietary analogue hardware description language
Verilog-AMS	Verilog for Analog and Mixed-Signal	an open standard extending Verilog for analog and mixed analog/digital simulation
HDL-A™	HDL-A	a proprietary analogue hardware description language

Digital circuit design

The two most widely-used and well-supported HDL varieties used in industry are Verilog and VHDL.

Abbreviation	Name	Notice

ABEL	Advanced Boolean Expression Language	
AHDL	Altera HDL	a proprietary language from Altera)
AHPL	A Hardware Programing language	
Bluespec		high-level HDL originally based on Haskell, now with a SystemVerilog syntax
C-to-Verilog		Converter from C to Verilog
Confluence		a functional HDL; has been discontinued)
CoWareC		a C-based HDL by CoWare. Now discontinued in favor of SystemC
CUPL		a proprietary language from Logical Devices, Inc.)
ELLA		no longer in common use)
Handel-C		a C-like design language
HJJ	Hardware Join Java	based on Join Java)
HML		based on SML
Hydra		based on Haskell
Impulse C	another C-like HDL	
ParC	Parallel C++	C++ extended with HDL style threading and communication for task-parallel programming
JHDL		based on Java)
Lava		based on Haskell
Lola		a simple language used for teaching
M		A HDL from Mentor Graphics
MyHDL		based on Python
PALASM		for Programmable Array Logic (PAL) devices
ROCCC 2.0	Riverside Optimizing Compiler for Configurable Computing	Free and open-source C to HDL tool
RHDL		based on the Ruby programming language)

Ruby (hardware description language)		
SystemC		a standardized class of C++ libraries for high-level behavioral and transaction modeling of digital hardware at a high level of abstraction, i.e. system-level
SystemVerilog		a superset of Verilog, with enhancements to address system-level design and verification
SystemTCL		SDL based on Tcl.
Verilog		most widely-used and well-supported HDL
VHDL	VHSIC HDL	most widely-used and well-supported HDL

Chapter 6

Algebraic Petri Nets & Fundamental Modeling Concepts

Algebraic Petri Nets

Algebraic Petri Net (APN) is an evolution of the well known **Petri net** in which elements of **User Defined Data Types** (called **Algebraic Abstract Data Types (AADT)**) replace black tokens. This formalism can be compared to Coloured Petri Nets CPN in many aspects. However, in the APN case, the semantics of the data types is given by an axiomatization enabling proofs and computations on it.

Algebraic Petri Nets were invented by Jacques Vautherin in 85 in is PhD thesis and later improved by Wolfang Reisig.

The formalism has two aspects :

- The control part which is handled by a Petri Net.
- The data part which is handled by one or many AADTs.

AADT can be themselves split in two part :

- The **signature** (Sort and Ops in the example below) which gives the valid constants and operations of the term algebra.
- The **axiomatization** (Axioms in the example below) which gives the semantic of the operations described in the signature part.

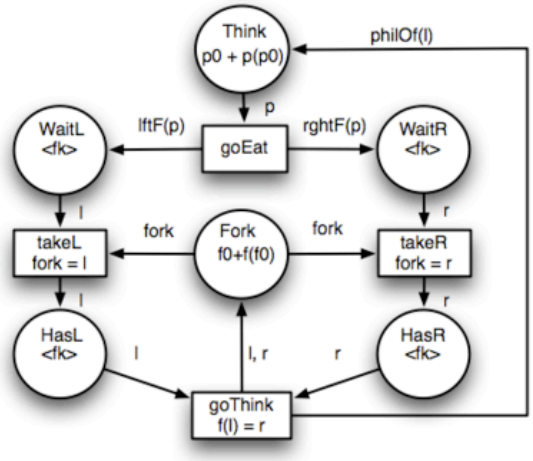
The following picture describes an Algebraic Petri Net model of the Dining Philosophers problem. There are two **AADT** in this model, one for the forks algebra, one for the philosophers algebra. Please note that the philosophers AADT uses the fork AADT.

Since all philosophers can take their left fork without taking their right fork, executing this model can result in a **deadlock**.

The control part is composed of :

- **Places** contain **Multiset** (bags) of tokens. Those tokens are elements of a term algebra built upon the signature of the **AADT**(in the example, terms that represent either a philosopher or a fork). Each place contains one and only one multiset of terms, the place is typed by its multiset.
- **Arcs** can be labeled with multisets of either closed or free terms. Again terms are built from the AADT signature.
- **Transitions** are events that can be fired whenever there are enough resources (namely enough tokens in the input places to satisfy all the input arcs) and the guard (firing conditions) of the transition holds. Then the produced tokens are put in the target places of the output arcs. Usually Term Rewriting is used for the operational semantics in order to check if conditions hold and to compute output terms.

In the example below only transition **goEat** is firable at the beginning. One **goEat** has been fired, **takeL** and **takeR** are also enabled and thus can also be fired.




```
ADT Philosopher
Sort philo
Use fork
Ops
p0: → philo
p: philo → philo
lftF: philo → fork
rghtF: philo → fork
philoOf : fork → philo
Axioms
p(p(po))=p0
lftF(p0)=f0
lftF(p(ph))=f(lftF(ph))
rghtF(p0)=f(f0)
rghtF(p(ph))=f(rghtF(ph))
philOf(f0)=p0
philOf(f(fk))=p(philOf(fk))

With ph : philo, fk : fork
```

```
ADT Fork
Sort fork
Ops
f0: → fork
f: fork → fork
Axioms
f(f(f0))=f0

With fk : fork
```

Algebraic Petri Nets are the basic formalism of more advanced ones such as CO-OPN.

Fundamental Modeling Concepts

Fundamental Modeling Concepts (FMC) provide a framework to describe software-intensive systems. It strongly emphasizes the communication about software-intensive systems by using a semi-formal graphical notation that can easily be understood.

Introduction

FMC distinguishes three perspectives to look at a software system:

- Structure of the system
- Processes in the system
- Value domains of the system

FMC defines a dedicated diagram type for each perspective. FMC diagrams use a simple and lean notation. The purpose of FMC diagrams is to facilitate the communication about a software system, not only between technical experts but also between technical experts and business or domain experts. The comprehensibility of FMC diagrams has made them famous among its supporters.

The common approach when working with FMC is to start with a high-level diagram of the compositional structure of a system. This “big picture” diagram serves as a reference in the communication with all involved stakeholders of the project. Later on, the high-level diagram is iteratively refined to model technical details of the system. Complementary diagrams for processes observed in the system or value domains found in the system are introduced as needed.

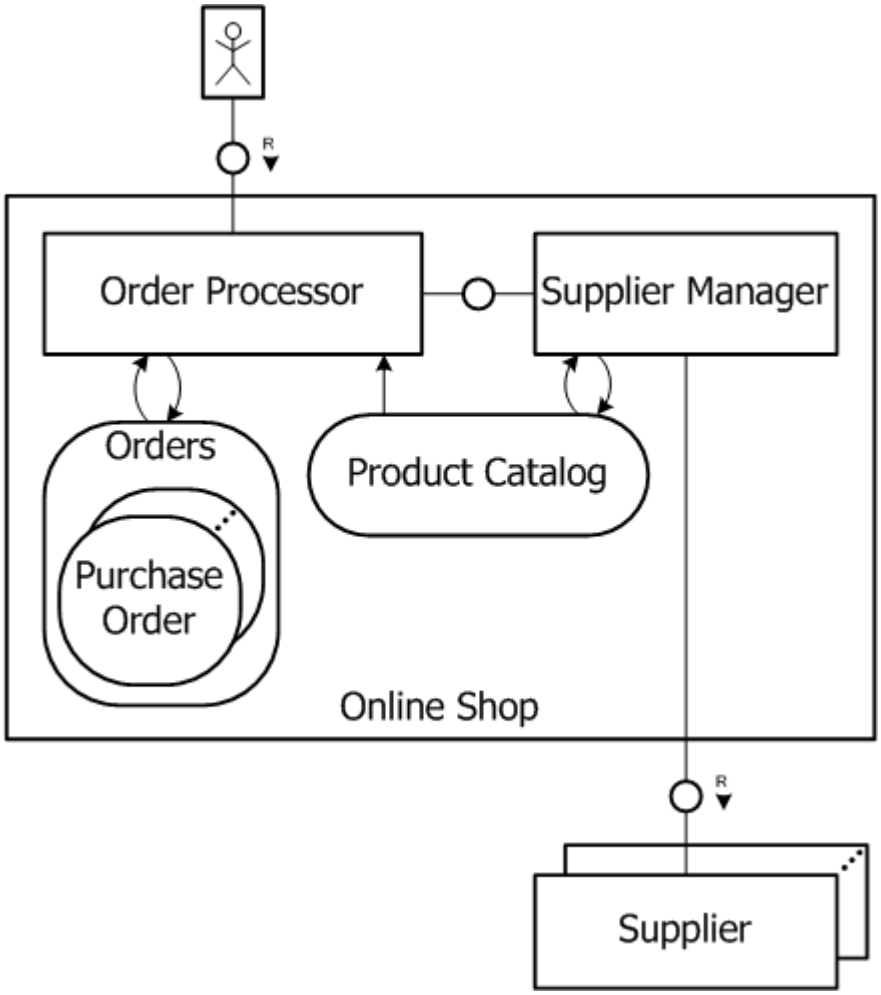
Diagram Types

FMC uses three diagram types to model different aspects of a system:

- **Compositional Structure Diagram** depicts the static structure of a system. This diagram type is also known as FMC Block Diagram
- **Dynamic Structure Diagram** depicts processes that can be observed in a system. This diagram type is also known as FMC Petri-net
- **Value Range Structure Diagram** depicts structures of values found in the system. This diagram type is also known as FMC E/R Diagram

All FMC diagrams are bipartite graphs. Each Bipartite graph consists of two disjoint sets of vertices with the condition that no vertex is connected to another vertex of the same set. In FMC diagrams, members of one set are represented by angular shapes, and members of the other set are represented by curved shapes. Each element in an FMC diagram can be refined by another diagram of the same type, provided that the combined graph is also bipartite. This mechanism allows modeling all relevant layers of abstraction with the same notation.

Compositional Structure Diagram



Example of FMC Compositional Structure Diagram

Compositional structure diagrams depict the static structure of a system, and the relationships between system components. System components can be active or passive. **Agents** are active system components. They perform activities in the system. **Storages** and **channels** are passive components which store or transmit information.

The image to the right is an example of a compositional structure diagram. It contains the agents *Order Processor*, *Supplier Manager*, *Supplier*, *Online Shop* and an unnamed **human agent**. Agents are represented by rectangles. The dots and the shadow of the agent *Supplier* indicate that this agent has multiple instances, i.e. the *Supplier Manager* communicates with one or many suppliers. The so called human agent represents a user interacting with the system.

The diagram contains the storages *Orders*, *Purchase Order* and *Product Catalog*. Storages are represented by curved shapes. Agents can read from storages, write to storages or modify the content of storages. The directions of the arrows indicate which

operation is performed by an agent. In the diagram, the *Supplier Manager* can modify the content of the *Product Catalog*, whereas the *Order Processor* can only read the content of the *Product Catalog*.

Agents communicate via channels. The direction of information flow is either indicated by arrows (not shown in the picture), by a request-response-symbol (e.g. between *Supplier Manager* and *Supplier*) or omitted (e.g. between *Order Processor* and *Supplier Manager*).

Dynamic Structure Diagram

Dynamic structures are derived from petri nets.

"They are used to express system behavior over time, depicting the actions performed by the agents. So they clarify how a system is working and how communication takes place between different agents."

Value Range Structure Diagram

Value range structure diagrams (also known as FMC Entity Relationship Diagrams) can be compared with the Entity-relationship model.

"[They] are used to depict value range structures or topics as mathematical structures. Value range structures describe observable values at locations within the system whereas topic diagrams allow a much wider usage in order to cover all correlations between interesting points."

Chapter 7

Business Process Execution Language

Business Process Execution Language (BPEL), short for *Web Services Business Process Execution Language* (WS-BPEL) is an OASIS standard executable language for specifying actions within business processes with web services. Processes in Business Process Execution Language export and import information by using web service interfaces exclusively.

Overview

Web service interactions can be described in two ways: executable business processes and abstract business processes. Executable business processes model actual behavior of a participant in a business interaction. Abstract business processes are partially specified processes that are not intended to be executed. An Abstract Process may hide some of the required concrete operational details. Abstract Processes serve a descriptive role, with more than one possible use case, including observable behavior and/or process template. WS-BPEL is meant to be used to model the behavior of both Executable and Abstract Processes.

WS-BPEL provides a language for the specification of Executable and Abstract business processes. By doing so, it extends the Web Services interaction model and enables it to support business transactions. WS-BPEL defines an interoperable integration model that should facilitate the expansion of automated process integration both within and between businesses.

The origins of BPEL can be traced to WSFL and XLANG. It is serialized in XML and aims to enable *programming in the large*. The concepts of *programming in the large* and *programming in the small* distinguish between two aspects of writing the type of long-running asynchronous processes that one typically sees in business processes.

Programming in the large generally refers to the high-level state transition interactions of a process—BPEL refers to this concept as an Abstract Process. A BPEL Abstract Process represents a set of publicly observable behaviors in a standardized fashion. An Abstract Process includes information such as when to wait for messages, when to send messages, when to compensate for failed transactions, etc. *Programming in the small*, in contrast, deals with short-lived programmatic behavior, often executed as a single transaction and involving access to local logic and resources such as files, databases, etc. BPEL's development came out of the notion that programming in the large and programming in the small required different types of languages.

History

IBM and Microsoft had each defined their own, fairly similar, "programming in the large" languages: WSFL and XLANG, respectively. With the advent and popularity of BPML, and the growing success of BPMI.org and the open BPMS movement led by JBoss and Intalio Inc., IBM and Microsoft decided to combine these languages into a new language, BPEL4WS. In April 2003, BEA Systems, IBM, Microsoft, SAP and Siebel Systems submitted BPEL4WS 1.1 to OASIS for standardization via the Web Services BPEL Technical Committee. Although BPEL4WS appeared as both a 1.0 and 1.1 version, the OASIS WS-BPEL technical committee voted on 14 September 2004 to name their spec "WS-BPEL 2.0". (This change in name aligned BPEL with other Web Service standard naming conventions which start with "WS-" and took account of the significant enhancements made between BPEL4WS 1.1 and WS-BPEL 2.0.) If not discussing a specific version, the moniker **BPEL** is commonly used.

In June 2007, Active Endpoints, Adobe Systems, BEA, IBM, Oracle and SAP published the BPEL4People and WS-HumanTask specifications, which describe how human interaction in BPEL processes can be implemented.

Business Process Execution Language topics

BPEL Design Goals

There were ten original design goals associated with BPEL:

1. Define business processes that interact with external entities through web service operations defined using WSDL 1.1, and that manifest themselves as Web services defined using WSDL 1.1. The interactions are “abstract” in the sense that the dependence is on portType definitions, not on port definitions.
2. Define business processes using an XML-based language. Do not define a graphical representation of processes or provide any particular design methodology for processes.
3. Define a set of Web service orchestration concepts that are meant to be used by both the external (abstract) and internal (executable) views of a business process. Such a business process defines the behavior of a single autonomous entity, typically operating in interaction with other similar peer entities. It is recognized

that each usage pattern (i.e. abstract view and executable view) will require a few specialized extensions, but these extensions are to be kept to a minimum and tested against requirements such as import/export and conformance checking that link the two usage patterns.
4. Provide both hierarchical and graph-like control regimes, and allow their use to be blended as seamlessly as possible. This should reduce the fragmentation of the process modeling space.
5. Provide data manipulation functions for the simple manipulation of data needed to define process data and control flow.
6. Support an identification mechanism for process instances that allows the definition of instance identifiers at the application message level. Instance identifiers should be defined by partners and may change.
7. Support the implicit creation and termination of process instances as the basic lifecycle mechanism. Advanced lifecycle operations such as "suspend" and "resume" may be added in future releases for enhanced lifecycle management.
8. Define a long-running transaction model that is based on proven techniques like compensation actions and scoping to support failure recovery for parts of long-running business processes.
9. Use Web Services as the model for process decomposition and assembly.
10. Build on Web services standards (approved and proposed) as much as possible in a composable and modular manner.

The BPEL language

BPEL is an orchestration language, not a choreography language. The primary difference between orchestration and choreography is executability and control. An orchestration specifies an executable process that involves message exchanges with other systems, such that the message exchange sequences are controlled by the orchestration designer. A choreography specifies a protocol for peer-to-peer interactions, defining, e.g., the legal sequences of messages exchanged with the purpose of guaranteeing interoperability. Such a protocol is not directly executable, as it allows many different realizations (processes that comply with it). A choreography can be realized by writing an orchestration (e.g. in the form of a BPEL process) for each peer involved in it. The orchestration and the choreography distinctions are based on analogies: orchestration refers to the central control (by the conductor) of the behavior of a distributed system (the orchestra consisting of many players), while choreography refers to a distributed system (the dancing team) which operates according to rules (the choreography) but without centralized control.

BPEL's focus on modern business processes, plus the histories of WSFL and XLANG, led BPEL to adopt web services as its external communication mechanism. Thus BPEL's messaging facilities depend on the use of the Web Services Description Language (WSDL) 1.1 to describe outgoing and incoming messages.

In addition to providing facilities to enable sending and receiving messages, the BPEL programming language also supports:

- A property-based message correlation mechanism
- XML and WSDL typed variables
- An extensible language plug-in model to allow writing expressions and queries in multiple languages: BPEL supports XPath 1.0 by default
- Structured-programming constructs including if-then-elseif-else, while, sequence (to enable executing commands in order) and flow (to enable executing commands in parallel)
- A scoping system to allow the encapsulation of logic with local variables, fault-handlers, compensation-handlers and event-handlers
- Serialized scopes to control concurrent access to variables

Relationship of BPEL to BPMN

There is no standard graphical notation for WS-BPEL, as the OASIS technical committee decided this was out of scope. Some vendors have invented their own notations. These notations take advantage of the fact that most constructs in BPEL are block-structured (e.g. sequence, while, pick, scope, etc.) This feature enables a direct visual representation of BPEL process descriptions in the form of *structograms*, in a style reminiscent of a Nassi–Shneiderman diagram.

Others have proposed to use a substantially different business process modeling language, namely Business Process Modeling Notation (BPMN), as a graphical front-end to capture BPEL process descriptions. As an illustration of the feasibility of this approach, the BPMN specification includes an informal and partial mapping from BPMN to BPEL 1.1. A more detailed mapping of BPMN to BPEL has been implemented in a number of tools, including an open-source tool known as BPMN2BPEL. However, the development of these tools has exposed fundamental differences between BPMN and BPEL, which make it very difficult, and in some cases impossible, to generate human-readable BPEL code from BPMN models. Even more difficult is the problem of BPMN-to-BPEL round-trip engineering: generating BPEL code from BPMN diagrams and maintaining the original BPMN model and the generated BPEL code synchronized, in the sense that any modification to one is propagated to the other.

Adding 'programming in the small' support to BPEL

BPEL's control structures such as 'if-then-elseif-else' and 'while' as well as its variable manipulation facilities depend on the use of 'programming in the small' languages to provide logic. All BPEL implementations must support XPath 1.0 as a default language. But the design of BPEL envisages extensibility so that systems builders can use other languages as well. BPELJ is an effort related to JSR 207 that may enable Java to function as a 'programming in the small' language within BPEL.

WS-BPEL 2.0

What's new in WS-BPEL 2.0?

- New activity types: repeatUntil, validate, forEach (parallel and sequential), rethrow, extensionActivity, compensateScope
- Renamed activities: switch/case renamed to if/else, terminate renamed to exit
- Termination Handler added to scope activities to provide explicit behavior for termination
- Variable initialization
- XSLT for variable transformations (New XPath extension function bpws:doXslTransform)
- XPath access to variable data (XPath variable syntax $variable[.part]/location)
- XML schema variables in Web service activities (for WS-I doc/lit style service interactions)
- Locally declared messageExchange (internal correlation of receive and reply activities)
- Clarification of Abstract Processes (syntax and semantics)
- Enable expression language overrides at each activity

Chapter 8

Goal-Oriented Requirements Language & Meta-Object Facility

Goal-Oriented Requirements Language

Goal-oriented Requirements Language (GRL), an i*-based modeling language used in systems development, is designed to support goal-oriented modeling and reasoning about requirements especially the non-functional requirements

GRL topics

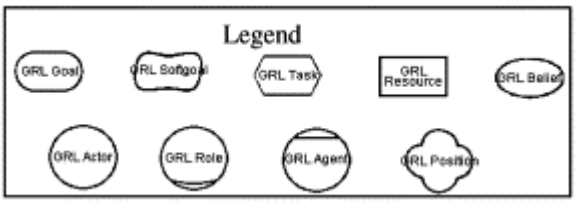
Concepts

Goal-oriented Requirements Language (GRL) allows to express conflict between goals and helps to make decisions that resolve conflicts. There are three main categories of concepts in GRL:

- intentional elements,
- intentional relationships and
- actors.

They are called for intentional because they are used in models that primarily concerned with answering "why" question of requirements (for ex. why certain choices for behavior or structure were made, what alternatives exist and what is the reason for choosing of certain alternative.)

Intentional elements

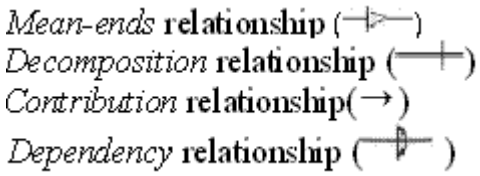


GRL Notation

Intentional elements are: goal, soft goal, task, belief and resource.

- Goal is condition or situation that can be achieved or not. Goal is used to define the functional requirements of the system. In GRL notation goal is represented by a rounded rectangle with the goal name inside.
- Task is used to represent different ways of how to accomplish goal. In GRL notation task is represented by hexagon with the task name inside.
- Softgoal is used to define non-functional requirements. It's usually a quality attribute of one of the intentional elements. In GRL notation softgoal is represented by irregular curvilinear shape with the softgoal name inside.
- Resource is a physical or informational object that is available for use in the task. Resource is represented in GRL as a rectangle.
- Belief is used to represent assumptions and relevant conditions. This construct is represented as ellipse in GRL notation.
- Actor is an active object that carries out actions to achieve the goal. In GRL notation actor is represented as a circle with the actor name inside.
- Agent is a concrete actor, such as a human individual or machine.

Relationships



GRL relationships

Intentional relationships are: means-ends, decomposition, contribution, correlation and dependency.

- Means-ends relationship shows how the goal can be achieved. For example it can be used to connect task to a goal.
- Decomposition relationship is used to show the sub-components of a task.
- Contribution relationship describes how one element influence another one.
- Correlation relationship describes side effects of existence of one element to others.
- Dependency relationship describe interdependences between agents.

GRL Tool Support

At present, GRL is supported by a general-purpose organization modelling tool - OME (Organization Modeling Environment). OME provides support to various modelling frameworks by loading the framework and its functional modules dynamically.

Meta-Object Facility

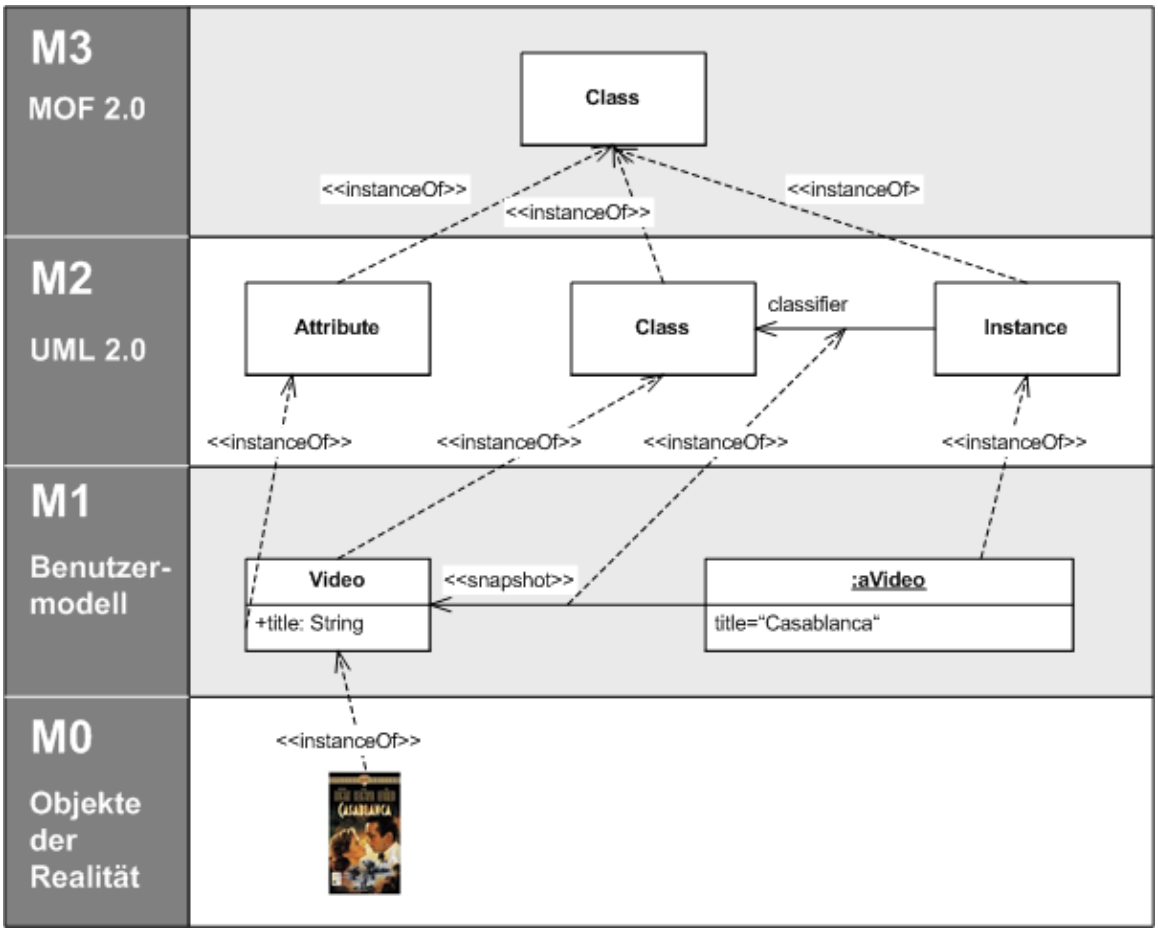


Illustration of the Meta-Object Facility.

The **Meta-Object Facility** (**MOF**) is an Object Management Group (OMG) standard for model-driven engineering. The official reference page may be found at OMG's website.

Overview

MOF originated in the Unified Modeling Language (UML); the OMG was in need of a metamodeling architecture to define the UML. MOF is designed as a four-layered architecture. It provides a meta-meta model at the top layer, called the M3 layer. This M3-model is the language used by MOF to build metamodels, called M2-models. The most prominent example of a Layer 2 MOF model is the UML metamodel, the model that describes the UML itself. These M2-models describe elements of the M1-layer, and thus M1-models. These would be, for example, models written in UML. The last layer is the M0-layer or data layer. It is used to describe real-world objects.

Beyond the M3-model, MOF describes the means to create and manipulate models and metamodels by defining CORBA interfaces that describe those operations. Because of the

similarities between the MOF M3-model and UML structure models, MOF metamodels are usually modeled as UML class diagrams. A supporting standard of MOF is XMI, which defines an XML-based exchange format for models on the M3-, M2-, or M1-Layer.

Metamodeling architecture

MOF is a *closed* metamodeling architecture; it defines an M3-model, which conforms to itself. MOF allows a *strict* meta-modeling architecture; every model element on every layer is strictly in correspondence with a model element of the layer above. MOF only provides a means to define the structure, or abstract syntax of a language or of data. For defining metamodels, MOF plays exactly the role that EBNF plays for defining programming language grammars. MOF is a Domain Specific Language (DSL) used to define metamodels, just as EBNF is a DSL for defining grammars. Similarly to EBNF, MOF could be defined in MOF.

In short MOF uses the notion of **MOF::Classes** (not to be confused with **UML::Classes**), as known from object orientation, to define concepts (model elements) on a metalayer. MOF may be used to define object-oriented metamodels (as UML for example) as well as non object-oriented metamodels (as a Petri net or a Web Service metamodel).

As of May 2006, the OMG has defined two variants of MOF:

- EMOF for Essential MOF
- CMOF for Complete MOF

In June 2006, a *request for proposal* was issued by OMG for a third variant, SMOF (Semantic MOF).

The variant **ECore** that has been defined in the **Eclipse Modeling Framework** is more or less aligned on OMG's EMOF.

Another related standard is OCL, which describes a formal language that can be used to define model constraints in terms of predicate logic.

A very important new standard is QVT which introduces means to query, view and transform MOF-based models.

International standard

MOF is an international standard:

ISO/IEC 19502:2005 Information technology -- Meta Object Facility (MOF)

MOF can be viewed as a standard to write metamodels, for example in order to model the abstract syntax of Domain Specific Languages. Kermeta is an extension to MOF allowing executable actions to be attached to EMOF meta-models, hence making it possible to also model a DSL operational semantics and readily obtain an interpreter for it.

JMI defines a Java API for manipulating MOF models.

OMG's MOF is not to be confused with the Managed Object Format (MOF) defined by the Distributed Management Task Force (DMTF) in section 6 of the Common Information Model (CIM) Infrastructure Specification, version 2.5.0.

Chapter 9

Modeling Language

A **modeling language** is any artificial language that can be used to express information or knowledge or systems in a structure that is defined by a consistent set of rules. The rules are used for interpretation of the meaning of components in the structure.

Overview

A modeling language can be graphical or textual.

- *Graphical* modeling languages use a diagram technique with named symbols that represent concepts and lines that connect the symbols and represent relationships and various other graphical notation to represent constraints.
- *Textual* modeling languages typically use standardized keywords accompanied by parameters to make computer-interpretable expressions.

An example of a graphical modeling language and a corresponding textual modeling language is EXPRESS.

Not all modeling languages are executable, and for those that are, the use of them doesn't necessarily mean that programmers are no longer required. On the contrary, executable modeling languages are intended to amplify the productivity of skilled programmers, so that they can address more challenging problems, such as parallel computing and distributed systems.

A large number of modeling languages appear in the literature.

Type of modeling languages

Graphical types

Example of graphical modeling languages in the field of computer science, project management and systems engineering:

- Behavior Trees are a formal, graphical modeling language used primarily in systems and software engineering. Commonly used to unambiguously represent the hundreds or even thousands of natural language requirements that are typically used to express the stakeholder needs for a large-scale software-integrated system.
- Business Process Modeling Notation (BPMN, and the XML form BPML) is an example of a Process Modeling language.
- EXPRESS and EXPRESS-G (ISO 10303-11) is an international standard general-purpose data modeling language.
- Extended Enterprise Modeling Language (EEML) is commonly used for business process modeling across a number of layers.
- Flowchart is a schematic representation of an algorithm or a stepwise process,
- Fundamental Modeling Concepts (FMC) modeling language for software-intensive systems.
- IDEF is a family of modeling languages, which include IDEF0 for functional modeling, IDEF1X for information modeling, IDEF3 for business process modeling, IDEF4 for Object-Oriented Design and IDEF5 for modeling ontologies.
- Jackson Structured Programming (JSP) is a method for structured programming based on correspondences between data stream structure and program structure
- LePUS3 is an object-oriented visual Design Description Language and a formal specification language that is suitable primarily for modeling large object-oriented (Java, C++, C#) programs and design patterns.
- Object Role Modeling (ORM) in the field of software engineering is a method for conceptual modeling, and can be used as a tool for information and rules analysis.
- Petri nets use variations on exactly one diagramming technique and topology, namely the bipartite graph. The simplicity of its basic user interface easily enabled extensive tool support over the years, particularly in the areas of model checking, graphically-oriented simulation, and software verification.
- Southbeach Notation is a visual modeling language used to describe situations in terms of agents that are considered useful or harmful from the modeler's perspective. The notation shows how the agents interact with each other and whether this interaction improves or worsens the situation.
- Specification and Description Language (SDL) is a specification language targeted at the unambiguous specification and description of the behaviour of reactive and distributed systems.
- SysML is a Domain-Specific Modeling language for systems engineering that is defined as a UML profile (customization).

- Unified Modeling Language (UML) is a general-purpose modeling language that is an industry standard for specifying software-intensive systems. UML 2.0, the current version, supports thirteen different diagram techniques, and has widespread tool support.
- Service-Oriented Modeling Framework (SOMF) is a holistic language for designing enterprise and application level architecture models in the space of enterprise architecture, virtualization, service-oriented architecture (SOA), cloud computing, and more.
- Architecture description language (ADL) is a language used to describe and represent the system architecture of a system.

Examples of graphical modeling languages in other fields of science.

- EAST-ADL is a Domain-Specific Modeling language dedicated to automotive system design.
- Energy Systems Language (ESL), a language that aims to model ecological energetics & global economics.

More specific types

In the field of computer science recently more specific types of modeling languages have emerged.

Algebraic

Algebraic Modeling Languages (AML) are high-level programming languages for describing and solving high complexity problems for large scale mathematical computation (i.e. large scale optimization type problems). One particular advantage of AMLs like AIMMS, AMPL, GAMS, LPL, MPL, OPL and OptimJ is the similarity of its syntax to the mathematical notation of optimization problems. This allows for a very concise and readable definition of problems in the domain of optimization, which is supported by certain language elements like sets, indices, algebraic expressions, powerful sparse index and data handling variables, constraints with arbitrary names. The algebraic formulation of a model does not contain any hints how to process it.

Discipline-Specific

A discipline-specific modeling (DspM) language is focused on deliverables affiliated with a specific software development life cycle stage. Therefore, such language offers a distinct vocabulary, syntax, and notation for each stage, such as discovery, analysis, design, architecture, contraction, etc. For example, for the analysis phase of a project, the modeler employes specific analysis notation to deliver an analysis proposition diagram. During the design phase, however, logical design notation is used to depict relationship between software entities. In addition, the discipline-specific modeling language best practices does not preclude practitioners from combining the various notations in a single diagram.

Domain-specific

Domain-specific modeling (DSM) is a software engineering methodology for designing and developing systems, most often IT systems such as computer software. It involves systematic use of a graphical domain-specific language (DSL) to represent the various facets of a system. DSM languages tend to support higher-level abstractions than General-purpose modeling languages, so they require less effort and fewer low-level details to specify a given system.

Framework-specific

A framework-specific modeling language (FSML) is a kind of domain-specific modeling language which is designed for an object-oriented application framework. FSMLs define framework-provided abstractions as FSML concepts and decompose the abstractions into features. The features represent implementation steps or choices.

A FSML concept can be configured by selecting features and providing values for features. Such a concept configuration represents how the concept should be implemented in the code. In other words, concept configuration describes how the framework should be completed in order to create the implementation of the concept.

Object-oriented

Object modeling language are modeling languages based on a standardized set of symbols and ways of arranging them to model (part of) an object oriented software design or system design.

Some organizations use them extensively in combination with a software development methodology to progress from initial specification to an implementation plan and to communicate that plan to an entire team of developers and stakeholders. Because a modeling language is visual and at a higher-level of abstraction than code, using models encourages the generation of a shared vision that may prevent problems of differing interpretation later in development. Often software modeling tools are used to construct these models, which may then be capable of automatic translation to code.

Virtual reality

Virtual Reality Modeling Language (VRML), before 1995 known as the Virtual Reality Markup Language is a standard file format for representing 3-dimensional (3D) interactive vector graphics, designed particularly with the World Wide Web in mind.

Others

- Architecture Description Language
- Face Modeling Language

- Generative Modelling Language
- Java Modeling Language
- Promela
- Rebeca Modeling Language
- Service Modeling Language
- Web Services Modeling Language
- X3D

Applications

Various kinds of modeling languages are applied in different disciplines, including computer science, information management, business process modeling, software engineering, and systems engineering. Modeling languages can be used to specify:

- system requirements,
- structures and
- behaviors.

Modeling languages are intended to be used to precisely specify systems so that stakeholders (e.g., customers, operators, analysts, designers) can better understand the system being modeled.

The more mature modeling languages are precise, consistent and executable. Informal diagramming techniques applied with drawing tools are expected to produce useful pictorial representations of system requirements, structures and behaviors, but not much else. Executable modeling languages applied with proper tool support, however, are expected to automate system verification and validation, simulation and code generation from the same representations.

Chapter 10

Promela

PROMELA (**Process** or **Protocol Meta Language**) is a verification modeling language. The language allows for the dynamic creation of concurrent processes to model, for example, distributed systems. In PROMELA models, communication via message channels can be defined to be synchronous (i.e., rendezvous), or asynchronous (i.e., buffered). PROMELA models can be analyzed with the SPIN model checker, to verify that the modeled system produces the desired behavior.

Introduction

PROMELA is a process modeling language whose intended use is to verify the logic of parallel systems. Given a program in PROMELA, Spin can verify the model for correctness by performing random or iterative simulations of the modeled system's execution, or it can generate a C program that performs a fast exhaustive verification of the system state space. During simulations and verifications SPIN checks for the absence of deadlocks, unspecified receptions, and unexecutable code. The verifier can also be used to prove the correctness of system invariants and it can find non-progress execution cycles. Finally, it supports the verification of linear time temporal constraints; either with Promela never-claims or by directly formulating the constraints in temporal logic. Each model can be verified with Spin under different types of assumptions about the environment. Once the correctness of a model has been established with Spin, that fact can be used in the construction and verification of all subsequent models.

PROMELA programs consist of *processes, message channels*, and *variables*. Processes are global objects that represent the concurrent entities of the distributed system. Message channels and variables can be declared either globally or locally within a process. Processes specify behavior, channels and global variables define the environment in which the processes run.

PROMELA Language Reference

Data Types

The basic data types used in PROMELA are presented in the table below. The sizes in bits are given for a PC i386/Linux machine.

Name	Size (bits)	Usage	Range
bit	1	unsigned	0..1
bool	1	unsigned	0..1
byte	8	unsigned	0..255
mtype	8	unsigned	0..255
short	16	signed	$-2^{15}..2^{15} - 1$
int	32	signed	$-2^{31}..2^{31} - 1$

The names bit and bool are synonyms for a single bit of information. A byte is an unsigned quantity that can store a value between 0 and 255. shorts and ints are signed quantities that differ only in the range of values they can hold.

Variables can also be declared as arrays. For example, the declaration:

```
int x ;
```

declares an array of 10 integers that can be accessed in array subscript expressions like:

```
x[0] = x + x;
```

But the arrays can not be ennumerated on creation, so they must be initialised as follows:

```
int x;
x[0] = 1;
x = 2;
x = 3;
```

The index to an array can be any expression that determines a unique integer value. The effect of an index outside the range is undefined. Multi-dimensional arrays can be defined indirectly with the help of the typedef construct (see below).

Processes

The state of a variable or of a message channel can only be changed or inspected by processes. The behavior of a process is defined by a *proctype* declaration. For example, the following declares a process type *A* with one variable state:

```
proctype A()
{
```

```
    byte state;
    state = 3;
}
```

The *proctype* definition only declares process behavior, it does not execute it. Initially, in the PROMELA model, just one process will be executed: a process of type *init*, that must be declared explicitly in every PROMELA specification.

New processes can be spawned using the *run* statement, which takes an argument consisting of the name of a *proctype*, from which a process is then instantiated. The *run* operator can be used in the body of the *proctype* definitions, not only in the initial process. This allows for dynamic creation of processes in PROMELA.

An executing process disappears when it terminates—that is, when it reaches the end of the body in the *proctype* definition, and all child processes that it started have terminated.

A proctype may also be *active* (below).

The Atomic Construct

By prefixing a sequence of statements enclosed in curly braces with the keyword *atomic* the user can indicate that the sequence is to be executed as one indivisible unit, non-interleaved with any other processes.

```
atomic
{
    statements;
}
```

It is a runtime error if any statement, other than the first statement, blocks in an atomic sequence. Atomic sequences can be an important tool in reducing the complexity of verification models. Note that atomic sequences restricts the amount of interleaving that is allowed in a distributed system. Intractable models can be made tractable by labeling all manipulations of local variables with atomic sequences.

Message Passing

Message channels are used to model the transfer of data from one process to another. They are declared either locally or globally, for instance as follows:

```
chan qname =  of {short}
```

This declares a buffered channel that can store up to 16 messages of type *short* (*capacity* is 16 here).

The statement:

```
qname ! expr;
```

sends the value of the expression *expr* to the channel with name *qname*, that is, it appends the value to the tail of the channel.

The statement:

```
qname ? msg;
```

receives the message, retrieves it from the head of the channel, and stores it in the variable msg. The channels pass messages in first-in-first-out order.

A rendezvous port can be declared as a message channel with the store length zero. For example, the following:

```
chan port = [0] of {byte}
```

defines a rendezvous port that can pass messages of type *byte*. Message interactions via such rendezvous ports are by definition synchronous, i.e. sender or receiver (the one that arrives *first* at the channel) will block for the contender that arrives *second* (receiver or sender).

When a buffered channel has been filled to its capacity (sending is "capacity" number of outputs ahead of receiving inputs), the default behavior of the channel is to become synchronous, and the sender will block on the next sending. Observe that there is no common message buffer shared between channels. Increasing complexity, as compared to using a channel as unidirectional and point to point, it *is* possible to share channels between multiple receivers or multiple senders, and to merge independent data-streams into a single shared channel. From this follows that a single channel may also be used for bidirectional communication.

Control Flow Constructs

There are three control flow constructs in PROMELA. They are the *case selection*, the *repetition* and the *unconditional jump*.

Case Selection

The simplest construct is the selection structure. Using the relative values of two variables *a* and *b*, for example we can write:

```
if
:: (a != b) -> option1
:: (a == b) -> option2
fi
```

The selection structure contains two execution sequences, each preceded by a double colon. One sequence from the list will be executed. A sequence can be selected only if its first statement is executable. The first statement of a control sequence is called a guard.

In the example above, the guards are mutually exclusive, but they need not be. If more than one guard is executable, one of the corresponding sequences is selected non-deterministically. If all guards are unexecutable the process will block until one of them can be selected. (Opposite, the occam *programming* language would *stop* or not be able to proceed on no executable guards.)

```
if
:: (A == true) -> option1;
:: (B == true) -> option2; /* May arrive here also if A==true */
:: else ->; fallthrough_option;
fi
```

The consequence of the non-deterministic choice is that, in the example above, if A is true, *both choices may be taken*. In "traditional" programming, one would understand an *if - if - else* structure sequentially. Here, the *if - double colon - double colon* must be understood as "any one being ready" and if none is ready, only then would the *else* be taken.

```
if
:: value = 3;
:: value = 4;
fi
```

In the example above, value is non-deterministically given the value 3 or 4.

There are two pseudo-statements that can be used as guards: the *timeout* statement and the *else* statement. The *timeout* statement models a special condition that allows a process to abort the waiting for a condition that may never become true. The else statement can be used as the initial statement of the last option sequence in a selection or iteration statement. The *else* is only executable if all other options in the same selection are not executable. Also, the *else* may not be used together with channels.

Repetition (Loop)

A logical extension of the selection structure is the repetition structure. For example:

```
do
:: count = count + 1
:: a = b + 2
:: (count == 0) -> break
od
```

describes a repetition structure in PROMELA. Only one option can be selected at a time. After the option completes, the execution of the structure is repeated. The normal way to terminate the repetition structure is with a *break* statement. It transfers the control to the instruction that immediately follows the repetition structure.

Unconditional Jumps

Another way to break a loop is the goto statement. For example, we can modify the example above as follows:

```
do
:: count = count + 1
:: a = b + 2
:: (count == 0) -> goto done
od
done:
skip;
```

The *goto* in this example jumps to a label named done. A label can only appear before a statement. If we might want to jump at the end of the program, for example, a dummy statement *skip* is useful: it is a place holder that is always executable and has no effect.

Assertions

An important language construct in PROMELA that needs a little explanation is the *assert* statement. Statements of the form:

```
assert(any_boolean_condition)
```

are always executable. If a boolean condition specified holds, the statement has no effect. If, however, the condition does not necessarily hold, the statement will produce an error during verifications with Spin.

Complex Data Structures

A PROMELA *typedef* definition can be used to introduce a new name for a list of data objects of predefined or earlier defined types. The new type name can be used to declare and instantiate new data objects, which can be used in any context in an obvious way:

```
typedef MyStruct
{
    short Field1;
    byte  Field2;
};
```

The access to the fields declared in a *typedef* construction is done in the same manner as in C programming language. For example:

```
MyStruct x;
x.Field1 = 1;
```

is a valid PROMELA sequence that assigns to the field *Field1* of the variable *x* the value *1*.

Active Proctypes

The *active* keyword that can be prefixed to any *proctype* definition. If the keyword is present, an instance of that proctype will be active in the initial system state. Multiple instantiations of that proctype can be specified with an optional array suffix of the keyword. Example:

```
active proctype A() { ... }
active  proctype B() { ... }
```

Executability

The semantics of *executability* provides the basic means in Promela for modeling process synchronizations.

```
mtype = {M_UP, M_DW};
chan Chan_data_down = [0] of {mtype};
chan Chan_data_up   = [0] of {mtype};
proctype P1 (chan Chan_data_in, Chan_data_out)
{
    do
    ::  Chan_data_in  ? M_UP -> skip;
    ::  Chan_data_out ! M_DW -> skip;
    od;
};
proctype P2 (chan Chan_data_in, Chan_data_out)
{
    do
    ::  Chan_data_in  ? M_DW -> skip;
    ::  Chan_data_out ! M_UP -> skip;
    od;
};
init
{
    atomic
    {
        run P1 (Chan_data_down, Chan_data_up);
        run P2 (Chan_data_up,   Chan_data_down);
    }
}
```

In the example, the two processes P1 and P2 have non-deterministic choices of (1) input from the other or (2) output to the other. Two rendezvous handshakes are possible, or *executable*, and one of them is chosen. This repeats forever. Therefore this model will not deadlock.

When Spin analyzes a model like the above, it will verify the choices with a non-deterministic algorithm, where all executable choices will be explored. However, when Spin's *simulator* visualizes possible non verified communication patterns, it may use a random generator to resolve the "non-deterministic" choice. Therefore the simulator may

fail to show a bad execution (in the example there is no bad trail). This illustrates a difference between verification and simulation.

Keywords

The following identifiers are reserved for use as keywords.

```
active      assert      atomic      bit
bool        break       byte        chan
d_step      D_proctype  do          else
empty       enabled     fi          full
goto        hidden      if          init
int         len         mtype       nempty
never       nfull       od          of
pc_value    printf      priority    proctype
provided    run         short       skip
timeout     typedef     unless      unsigned
xr          xs
```

Chapter 11

Petriscript

PetriScript is a modelling language for Petri nets, designed by Alexandre Hamez and Xavier Renault.

The CPN-AMI platform provides many tools to work on Petri nets, such as verifying or model-checking tools. It was easily possible to graphically design simple Petri nets with Macao, but various works made internally at LIP6 reveal that it was needed to automate such task.

Therefore PetriScript has been designed to provide some facilities in modelling places-transition and coloured Petri nets within the CPN-AMI platform.

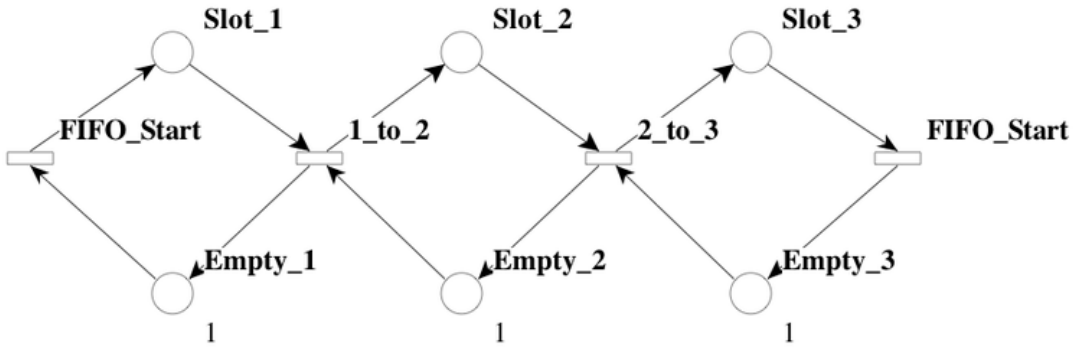
Its main purpose is to automate modelling operations on Petri nets such as merging, creating, and connecting nodes. Thus, it supports almost everything needed like macros, loops control, lists, string and arithmetic expressions, and avoids to the maximum the intervention of the user. Its syntax is more or less Ada-like.

For example, the following script produces a FIFO with three sections:

```
define(FIFO_SIZE,3)
define(FIFO_BASE_X,100)
define(FIFO_BASE_Y,100)
define(FIFO_STEP,120)
int $wave := 0;
for $wave in 1..FIFO_SIZE loop
        create place "Slot_" & '$wave' (x FIFO_BASE_X + FIFO_STEP *
$wave,
                y FIFO_BASE_Y);
        create place "Empty_" & '$wave' (x FIFO_BASE_X + FIFO_STEP *
$wave,
                y FIFO_BASE_Y + 100, marking "1");
```

```
end loop;
for $wave in 1..FIFO_SIZE+1 loop
        create transition "t" & '$wave -1' & "_to_" & '$wave' (x
FIFO_BASE_X + FIFO_STEP * $wave - FIFO_STEP / 2,
                y FIFO_BASE_Y + 50);
        if $wave < FIFO_SIZE+1 then
                connect "1" transition "t" &'$wave -1' & "_to_" &
'$wave' to  place "Slot_" & '$wave';
                connect "1" place "Empty_" & '$wave' to transition "t"
&'$wave -1' & "_to_" & '$wave';
        end if;
        if $wave > 1 then
                connect "1" transition "t" &'$wave -1' & "_to_" &
'$wave' to  place "Empty_" & '$wave - 1';
                connect "1" place "Slot_" & '$wave - 1' to transition
"t" &'$wave -1' & "_to_" & '$wave';
        end if;
end loop;
set transition "t0_to_1" to (name "FIFO_Start");
set transition "t" &  'FIFO_SIZE' & "_to_" & 'FIFO_SIZE + 1' to (name
"FIFO_End");
```

Which produces the following graph:

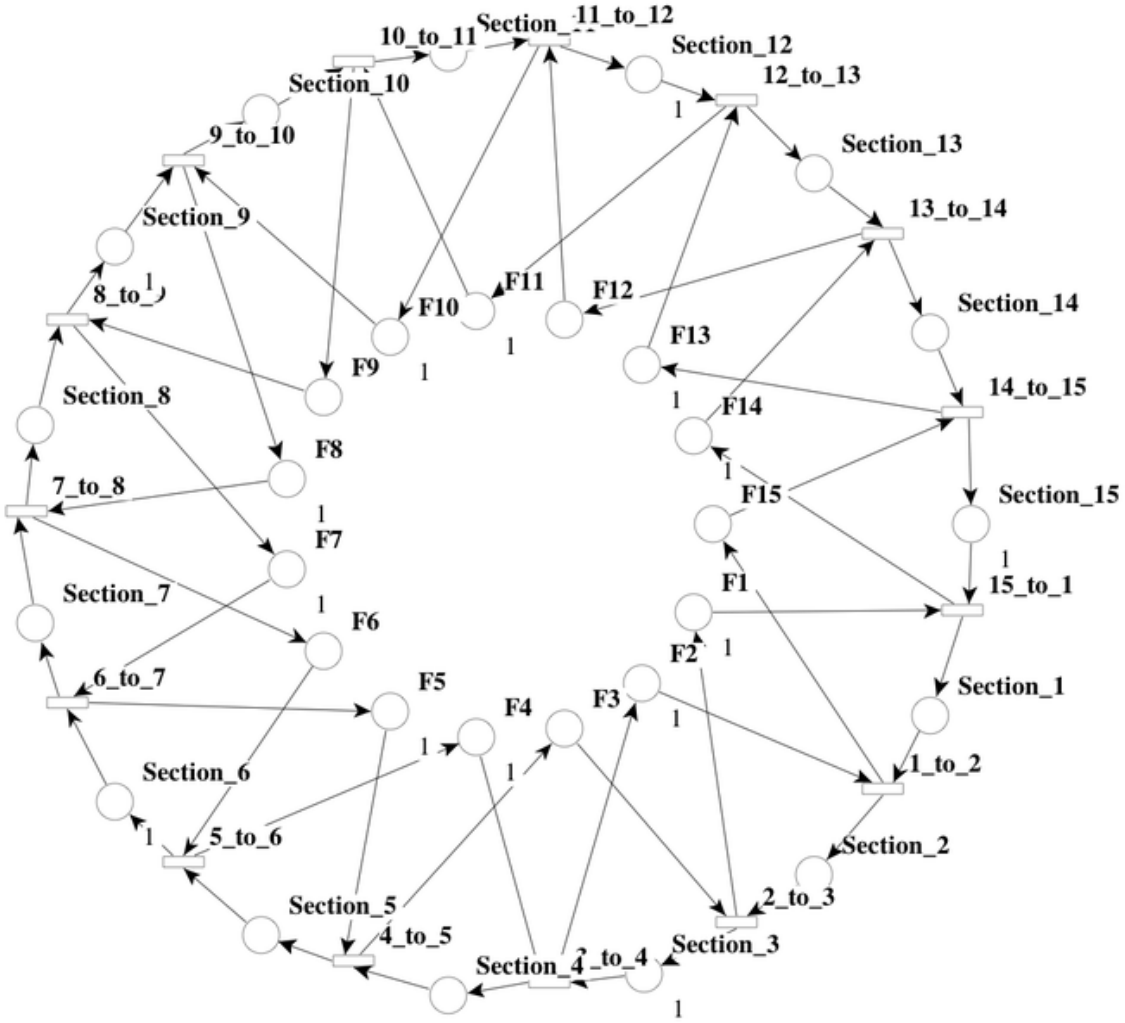


Here is another example that shows the power of PetriScript:

```
define(X,250)
define(Y,350)
define(radius,50)
define(R,150)

define(SECTIONS,15)

define(INNER_ANGLE,360/SECTIONS)
define(OUTER_ANGLE,360/(2*SECTIONS))

int $i := 0;
int $j := 0;

for $i in 1.. SECTIONS loop
        create place "F" & '$i' ( x X, y Y, r radius, t $i *
INNER_ANGLE);
```

```
        create place "Section_" & '$i' ( x X, y Y,  r R, t $i *
INNER_ANGLE);
        create transition "t" & '$i' & "_to_" & '$i mod SECTIONS + 1' (
x X, y Y, r R, t $i * INNER_ANGLE + OUTER_ANGLE);
end loop;


for $i in 1.. SECTIONS loop
        connect place "Section_" & '$i'  to transition "t"&'$i' &
"_to_" & '$i mod SECTIONS + 1';

        connect transition "t" & '$i' & "_to_" & '$i mod SECTIONS + 1'
to place "Section_" & '$i mod SECTIONS + 1';

        if $i /= 1 then
                connect place "F" & '$i'  to transition "t" & '$i-1' &
"_to_" & '$i';
        else
                connect place "F1"  to transition "t" & 'SECTIONS' &
"_to_" & '1';
        end if;

        connect transition "t" &'$i mod SECTIONS + 1' & "_to_" &
'($i+1) mod SECTIONS + 1' to place "F" & '$i';

end loop;

for $i in 1.. SECTIONS loop
        if $i mod 3 = 0 then
                set place "Section_" & '$i' to marking "1";
        else
                set place "F" & '$i' to marking "1";
        end if;
end loop;
```

Which produces the following graph:


10_to_11
Section_11_to_12
Section_12
12_to_13
Section_10
9_to_10
Section_13
13_to_14
Section_9
8_to_9
F10
F11
F12
Section_14
F13
F9
14_to_15
Section_8
F14
F8
7_to_8
F15
Section_15
F7
F1
15_to_1
Section_7
F6
F2
6_to_7
F5
F4
F3
Section_1
Section_6
1_to_2
5_to_6
Section_2
Section_5
2_to_3
4_to_5
Section_4
3_to_4
Section_3
1

Chapter 12

WebML

WebML (**Web Modeling Language**) is a visual notation for designing complex data-intensive Web applications. It provides graphical, yet formal, specifications, embodied in a complete design process, which can be assisted by visual design tools, like WebRatio.

This method has five models: structure, derivation, composition, navigation and presentation. These models are developed in an iterative process.

Concepts

WebML enables designers to express the core features of a site at a high level, without committing to detailed architectural details. WebML concepts are associated with an intuitive graphic representation, which can be easily supported by CASE tools and effectively communicated to the non-technical members of the site development team (e.g., with the graphic designers and the content producers). WebML also supports an XML syntax, which instead can be fed to software generators for automatically producing the implementation of a Web site. The specification of a site in WebML consists of four orthogonal perspectives:

1. **Structural Model:** it expresses the data content of the site, in terms of the relevant entities and relationships. WebML does not propose yet another language for data modeling, but is compatible with classical notations like the E/R model, the ODMG object-oriented model, and UML class diagrams.
2. **Hypertext Model:** it describes one or more hypertexts that can be published in the site. Each different hypertext defines a so-called site view (see Figure 2). Site view descriptions in turn consist of two sub-models.
    - **Composition Model:** it specifies which pages compose the hypertext, and which content units make up a page.
    - **Navigation Model:** it expresses how pages and content units are linked to form the hypertext. Links are either non-contextual, when they connect

semantically independent pages (e.g., the page of an artist to the home page of the site), or contextual, when the content of the destination unit of the link depends on the content of the source unit.

3. **Presentation Model:** it expresses the layout and graphic appearance of pages, independently of the output device and of the rendition language, by means of an abstract XML syntax. Presentation specifications are either page-specific or generic.
4. **Personalization Model:** users and user groups are explicitly modeled in the structure schema in the form of predefined entities called User and Group. The features of these entities can be used for storing group-specific or individual content, like shopping suggestions, list of favorites, and resources for graphic customization.

Design Process

A typical design process using WebML proceeds by iterating the following steps for each design cycle:

- **Requirements Collection.** Application requirements are gathered, which include the main objectives of the site, its target audience, examples of content, style guidelines, required personalization and constraints due to legacy data.
- **Data Design.** The data expert designs the structural model, possibly by reverse-engineering the existing logical schemas of legacy data sources.
- **Hypertext Design "in the large"**. The Web application architect defines the structure "in the large" of the hypertext, by identifying pages and units, linking them, and mapping units to the main entities and relationships of the structure schema. In this way, he develops a "skeleton" site view, and then iteratively improves it.
- **Hypertext Design "in the small"**. The Web application architect concentrates next in the design "in the small" of the hypertext, by considering each page and unit individually. At this stage, he may add non-contextual links between pages, consolidate the attributes that should be included within a unit, and introduce novel pages or units for special requirements (e.g., alternative index pages to locate objects, filters to search the desired information, and so on).
- **Presentation Design.** Once all pages are sufficiently stable, the Web style architect adds to each page a presentation style.
- **User and Group Design.** The Web administrator defines the features of user profiles, based on personalization requirements. Potential users and user groups are mapped to WebML users and groups, and possibly a different site view is created for each group. The design cycle is next iterated for each of the identified site views.
- **Customization Design.** The Web administrator identifies profile-driven data derivations and business rules, which may guarantee an effective personalization of the site.

Structural Model

The fundamental elements of WebML structure model are entities, which are containers of data elements, and relationships, which enable the semantic connection of entities. Entities have named attributes, with an associated type; properties with multiple occurrences can be organized by means of multi-valued components, which corresponds to the classical part-of relationship. Entities can be organized in generalization hierarchies. Relationships may be given cardinality constraints and role names.

Derivation Model

- In other words it is similar to **VIEWS** in databases modelling. Like VIEW in Oracle or MySQL.
- For each page there is **One abstract Table** of datas. But it is merged from other tables.
- Uses WebML-OQL (WebML-Object Query Language)

HyperText Model

- The most important model of the WebML methodology
- It models the navigation of user on the web.
- HyperText Model is compounded from 2 models: Composition and Navigational model.

Composition Model

The purpose of composition modeling is to define which nodes make up the hypertext contained in the Web site. More precisely, composition modeling specifies content units (units for short), i.e., the atomic information elements that may appear in the Web site, and pages, i.e., containers by means of which information is actually clustered for delivery to the user. In a concrete setting, e.g., an HTML or WML implementation of a WebML site, pages and units are mapped to suitable constructs in the delivery language, e.g., units may map to HTML files and pages to HTML frames organizing such files on the screen.

WebML supports six types of unit to compose an hypertext:

- **Data units** (show information about a single object).
- **Multidata units** (show information about a set of objects).
- **Index units** (show a list of objects without presenting the detailed information of each object).
- **Scroller units** (show commands for accessing the elements of an ordered set of objects).
- **Filter units** (show edit fields for inputting values used for searching within a set of object(s) those ones that meet a condition).

- **Direct units** (do not display information, but are used to denote the connection to a single object that is semantically related to another object).

Elements

- Data unit
- MultiData unit
- Index unit
- Multichoice index unit
- Hierarchical unit
- Scroller unit
- Entry unit

Navigational Model

Units and pages do not exist in isolation, but must be connected to form a hypertext structure. The purpose of navigation modeling is to specify the way in which the units and pages are linked to form a hypertext. To this purpose, WebML provides the notion of link. There are two variants of links:

- **Contextual links** (connect units in a way coherent to the semantics expressed by the structure schema of the application. Carries some information (called context) from the source unit to the destination unit. Context is used to determine the actual object or set of objects to be shown in the destination unit).
- **Non-contextual links** (connect pages in a totally free way, i.e., independently of the units they contain and of the semantic relations between the structural concepts included in those units. Syntactically, contextual and non-contextual links are denoted by element INFOLINK and HYPERLINK, respectively nested within units and pages).

Elements

- Web pages
- Links between pages

  context
  noncontext

Personalization Model

Personalization is the definition of content or presentation style based on user profile data. In WebML, units, pages, their presentation styles, and site views can be defined so to take user- or group-specific data into account. This can be done in two complementary ways:

- **Declarative personalization:** the designer defines derived concepts (e.g., entities, attributes, multi-valued components) whose definition depends on user-specific data. In this way, customization is specified declaratively; the system fills in the information relative to each user when computing the content of units.
- **Procedural personalization:** WebML includes an XML syntax for writing business rules that compute and store user-specific information. A business rule is a triple event-condition-action, which specifies the event to be monitored, the precondition to be checked when the event occurs, and the action to be taken when the condition is found true. Typical tasks performed by business rules are the assignment of users to user groups based on dynamically collected information, the notification of messages to users upon the update of the information base (push technology), the logging of user actions into user-specific data structures, and so on.

Presentational Model

Presentation modeling is concerned with the actual look and feel of the pages identified by composition modeling. WebML pages are rendered according to a style sheet. A style sheet dictates the layout of pages and the content elements to be inserted into such layout, and is independent of the actual language used for page rendition. For better reusability, two categories of style sheets are provided: untyped style sheets (also called models) describe the page layout independently of its content, and thus can be applied regardless of the mapping of the page to a given concept; typed style sheets are specified at a finer granularity and thus apply only to pages describing specific concepts.

Other possibilities

- UML-based Web Engineering (UWE)
- HDM
- RMM
- EORM
- OOHDM
- WSDM
- Araneus
- OO-H
- UML WAE
- Hera

Chapter 13

Tefkat & Z notation

Tefkat

Tefkat is a Model Transformation Language and a model transformation engine. The language is based on F-logic and the theory of stratified logic programs. The engine is an Eclipse plug-in for the Eclipse Modeling Framework (EMF).

History

Tefkat was one of the sub-projects of the Pegamento project at the Distributed Systems Technology Centre (DSTC), Australia. Although the project was already underway, the most active research occurred for the submission of a response to the OMG's MOF 2.0 Queries / Views / Transformations Request for Proposals.

Tefkat was open-sourced before the closure of the DSTC in June 2006, and is still under active development.

Brief Description

Tefkat defines a mapping from a set of source metamodels to a set of target metamodels. A Tefkat transformation consists of *rules*, *patterns* and *templates*. Rules contain a *source term* and a *target term*. Patterns are simply named composite source terms, and templates are simply named composite target terms. These elements are based on F-logic and pure logic programming, however the absence of function symbols means a significant reduction in complexity.

Tefkat has two more significant language elements: *trackings* and *injections*. Trackings allow arbitrary relationships to be preserved in a trace model. Injections allow the identity of target objects to be specified in terms of a function symbol. Thus injections are similar (but more powerful) to QVT's keys, which specify a target object's identity to be a function of its type and some of its properties.

The declarative semantics of a Tefkat transformation is the *perfect* model of traces and targets that satisfies all the rules. A more imperative semantics of a Tefkat transformation is the *iterated least fixed-point* of the immediate consequence of each rule. Due to stratification, these semantics are equivalent and unambiguous. Tefkat does not use explicit rule-calling; all (non-abstract) rules fire independently from all others, however rules can be loosely coupled using trackings, injections, rule extension and/or rule superseding.

Concrete Syntax

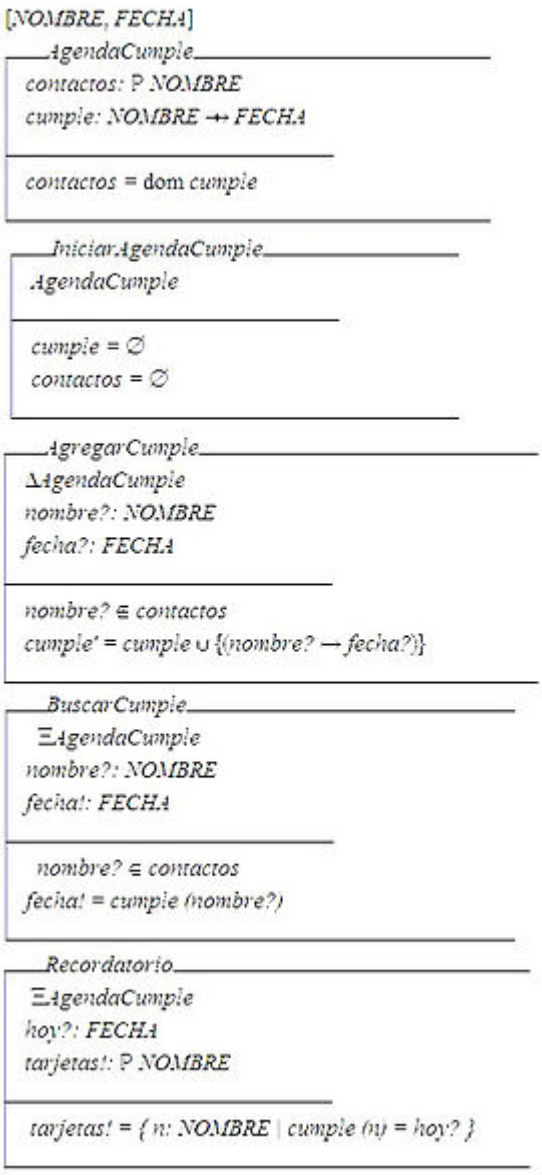
Tefkat has an SQL-like concrete syntax designed to concisely convey the intent of each rule, pattern or template.

```
RULE ClassToTable
FORALL  Class c { name: n; }
MAKE    Table t { name: n; }
;
```

Compliance

The Tefkat language is defined in terms of (E)MOF 2.0, however the engine is implemented in terms of Ecore, the EMOF-like metametamodel at the centre of EMF. The language is very similar to the Relations package of QVT, however it is not strictly compliant.

Z Notation



An example of a formal specification using the Z notation.

The **Z notation** named after Zermelo–Fraenkel set theory, is a formal specification language used for describing and modelling computing systems. It is targeted at the clear specification of computer programs and computer-based systems in general.

History

In 1974, Jean-Raymond Abrial published "Data Semantics". He used a notation that would later be taught in the University of Grenoble until the end of the 1980s. While at

EDF (Électricité de France), Abrial wrote internal notes on Z. The Z notation is used in the 1980 book *Méthodes de programmation*.

Z was originally proposed by Abrial in 1977 with the help of Steve Schuman and Bertrand Meyer. It was developed further at the Programming Research Group at Oxford University, where Abrial worked in the early 1980s, having arrived at Oxford in September 1979.

Abrial answers the question "Why Z?" with "Because it is the ultimate language!"

Usage and notation

Z is based on the standard mathematical notation used in axiomatic set theory, lambda calculus, and first-order predicate logic. All expressions in Z notation are typed, thereby avoiding some of the paradoxes of naive set theory. Z contains a standardized catalog (called the *mathematical toolkit*) of commonly used mathematical functions and predicates.

Although Z notation (just like the APL language, long before it) uses many non-ASCII symbols, the specification includes suggestions for rendering the Z notation symbols in ASCII and in LaTeX.

Standards

The ISO completed a Z standardization effort in 2002. This standard can be obtained directly from ISO.

Chapter 14

Programming Language

A **programming language** is an artificial language designed to express computations that can be performed by a machine, particularly a computer. Programming languages can be used to create programs that control the behavior of a machine, to express algorithms precisely, or as a mode of human communication.

The earliest programming languages predate the invention of the computer, and were used to direct the behavior of machines such as Jacquard looms and player pianos. Thousands of different programming languages have been created, mainly in the computer field, with many more being created every year. Most programming languages describe computation in an imperative style, i.e., as a sequence of commands, although some languages, such as those that support functional programming or logic programming, use alternative forms of description.

A programming language is usually split into the two components of syntax (form) and semantics (meaning) and many programming languages have some kind of written specification of their syntax and/or semantics. Some languages are defined by a specification document, for example, the C programming language is specified by an ISO Standard, while other languages, such as Perl, have a dominant implementation that is used as a reference.

Definitions

A programming language is a notation for writing programs, which are specifications of a computation or algorithm. Some, but not all, authors restrict the term "programming language" to those languages that can express *all* possible algorithms. Traits often considered important for what constitutes a programming language include:

- *Function and target:* A *computer programming language* is a language used to write computer programs, which involve a computer performing some kind of computation or algorithm and possibly control external devices such as printers, disk drives, robots, and so on. For example PostScript programs are frequently

created by another program to control a computer printer or display. More generally, a programming language may describe computation on some, possibly abstract, machine. It is generally accepted that a complete specification for a programming language includes a description, possibly idealized, of a machine or processor for that language. In most practical contexts, a programming language involves a computer; consequently programming languages are usually defined and studied this way. Programming languages differ from natural languages in that natural languages are only used for interaction between people, while programming languages also allow humans to communicate instructions to machines.

- *Abstractions:* Programming languages usually contain abstractions for defining and manipulating data structures or controlling the flow of execution. The practical necessity that a programming language support adequate abstractions is expressed by the abstraction principle; this principle is sometimes formulated as recommendation to the programmer to make proper use of such abstractions.
- *Expressive power:* The theory of computation classifies languages by the computations they are capable of expressing. All Turing complete languages can implement the same set of algorithms. ANSI/ISO SQL and Charity are examples of languages that are not Turing complete, yet often called programming languages.

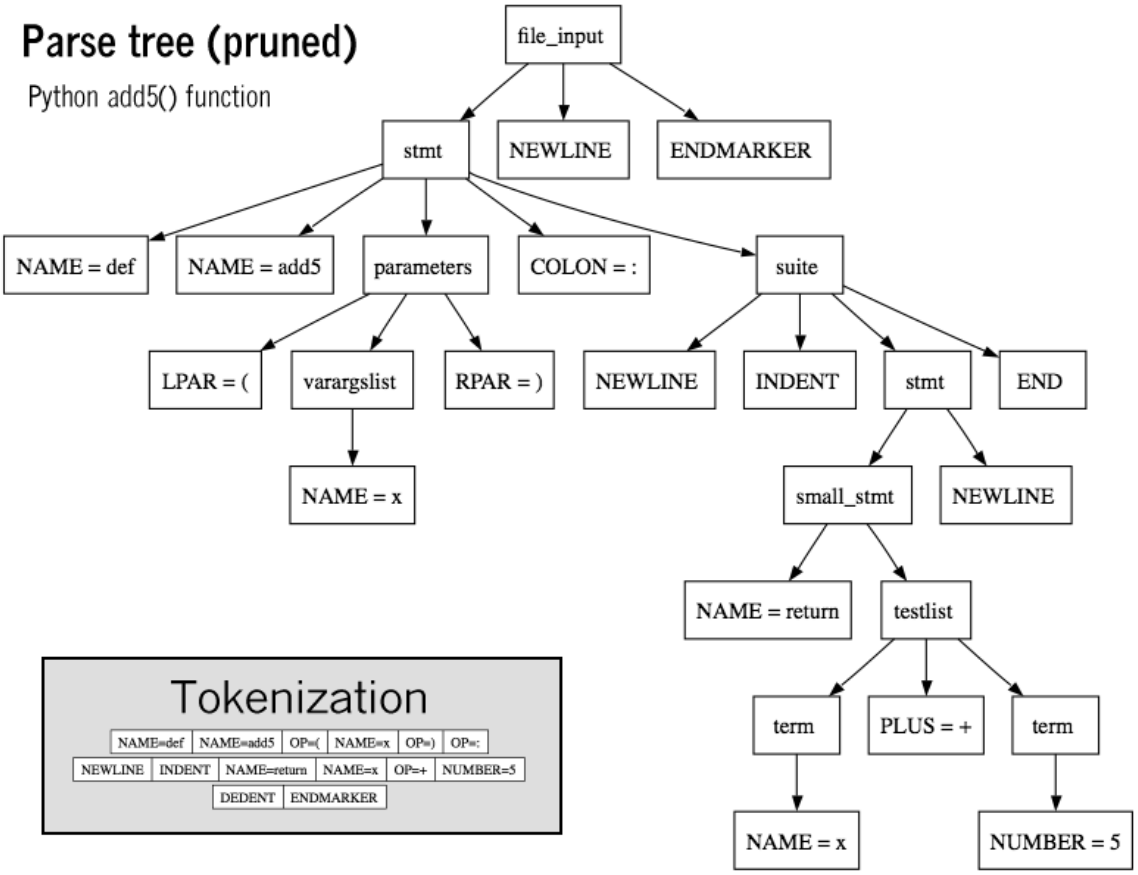
Markup languages like XML, HTML or troff, which define structured data, are not generally considered programming languages. Programming languages may, however, share the syntax with markup languages if a computational semantics is defined. XSLT, for example, is a Turing complete XML dialect. Moreover, LaTeX, which is mostly used for structuring documents, also contains a Turing complete subset.

The term *computer language* is sometimes used interchangeably with programming language. However, the usage of both terms varies among authors, including the exact scope of each. One usage describes programming languages as a subset of computer languages. In this vein, languages used in computing that have a different goal than expressing computer programs are generically designated computer languages. For instance, markup languages are sometimes referred to as computer languages to emphasize that they are not meant to be used for programming. Another usage regards programming languages as theoretical constructs for programming abstract machines, and computer languages as the subset thereof that runs on physical computers, which have finite hardware resources. John C. Reynolds emphasizes that formal specification languages are just as much programming languages as are the languages intended for execution. He also argues that textual and even graphical input formats that affect the behavior of a computer are programming languages, despite the fact they are commonly not Turing-complete, and remarks that ignorance of programming language concepts is the reason for many flaws in input formats.

Elements

All programming languages have some primitive building blocks for the description of data and the processes or transformations applied to them (like the addition of two numbers or the selection of an item from a collection). These primitives are defined by syntactic and semantic rules which describe their structure and meaning respectively.

Syntax



Parse tree of Python code with inset tokenization

```
def add5(x):
   return x+5

def dotwrite(ast):
   nodename = getNodename()
   label=symbol.sym_name.get(int(ast[0]),ast[0])
   print '    %s [label="%s' % (nodename, label),
   if isinstance(ast[1], str):
      if ast[1].strip():
         print '= %s"];' % ast[1]
      else:
         print '"]'
    else:
      print '"];'
      children = []
      for  in n, childenumerate(ast[1:]):
         children.append(dotwrite(child))
      print ,'    %s -> {' % nodename
      for  in :namechildren
         print '%s' % name,
```

Syntax highlighting is often used to aid programmers in recognizing elements of source code. The language above is Python.

A programming language's surface form is known as its syntax. Most programming languages are purely textual; they use sequences of text including words, numbers, and punctuation, much like written natural languages. On the other hand, there are some programming languages which are more graphical in nature, using visual relationships between symbols to specify a program.

The syntax of a language describes the possible combinations of symbols that form a syntactically correct program. The meaning given to a combination of symbols is handled by semantics (either formal or hard-coded in a reference implementation). Since most languages are textual.

Programming language syntax is usually defined using a combination of regular expressions (for lexical structure) and Backus–Naur Form (for grammatical structure). Below is a simple grammar, based on Lisp:

```
expression ::= atom | list
atom ::= number | symbol
number ::= [+-]?['0'-'9']+
symbol ::= ['A'-'Z''a'-'z'].*
list ::= '(' expression* ')'
```

This grammar specifies the following:

- an *expression* is either an *atom* or a *list*;
- an *atom* is either a *number* or a *symbol*;
- a *number* is an unbroken sequence of one or more decimal digits, optionally preceded by a plus or minus sign;
- a *symbol* is a letter followed by zero or more of any characters (excluding whitespace); and
- a *list* is a matched pair of parentheses, with zero or more *expressions* inside it.

The following are examples of well-formed token sequences in this grammar: '`12345`', '`()`','`(a b c232 (1))`'

Not all syntactically correct programs are semantically correct. Many syntactically correct programs are nonetheless ill-formed, per the language's rules; and may (depending on the language specification and the soundness of the implementation) result in an error on translation or execution. In some cases, such programs may exhibit undefined behavior. Even when a program is well-defined within a language, it may still have a meaning that is not intended by the person who wrote it.

Using natural language as an example, it may not be possible to assign a meaning to a grammatically correct sentence or the sentence may be false:

- "Colorless green ideas sleep furiously." is grammatically well-formed but has no generally accepted meaning.
- "John is a married bachelor." is grammatically well-formed but expresses a meaning that cannot be true.

The following C language fragment is syntactically correct, but performs an operation that is not semantically defined (because `p` is a null pointer, the operations `p->real` and `p->im` have no meaning):

```
complex *p = NULL;
complex abs_p = sqrt (p->real * p->real + p->im * p->im);
```

If the type declaration on the first line were omitted, the program would trigger an error on compilation, as the variable "p" would not be defined. But the program would still be syntactically correct, since type declarations provide only semantic information.

The grammar needed to specify a programming language can be classified by its position in the Chomsky hierarchy. The syntax of most programming languages can be specified using a Type-2 grammar, i.e., they are context-free grammars. Some languages, including Perl and Lisp, contain constructs that allow execution during the parsing phase. Languages that have constructs that allow the programmer to alter the behavior of the parser make syntax analysis an undecidable problem, and generally blur the distinction between parsing and execution. In contrast to Lisp's macro system and Perl's `BEGIN`

blocks, which may contain general computations, C macros are merely string replacements, and do not require code execution.

Semantics

The term *semantics* refers to the meaning of languages, as opposed to their form (syntax).

Static semantics

The static semantics defines restrictions on the structure of valid texts that are hard or impossible to express in standard syntactic formalisms. For compiled languages, static semantics essentially include those semantic rules that can be checked at compile time. Examples include checking that every identifier is declared before it is used (in languages that require such declarations) or that the labels on the arms of a case statement are distinct. Many important restrictions of this type, like checking that identifiers are used in the appropriate context (e.g. not adding a integer to a function name), or that subroutine calls have the appropriate number and type of arguments can be enforced by defining them as rules in a logic called a type system. Other forms of static analyses like data flow analysis may also be part of static semantics. Newer programming languages like Java and C# have definite assignment analysis, a form of data flow analysis, as part of their static semantics.

Dynamic semantics

Once data has been specified, the machine must be instructed to perform operations on the data. For example, the semantics may define the strategy by which expressions are evaluated to values, or the manner in which control structures conditionally execute statements. The *dynamic semantics* (also known as *execution semantics*) of a language defines how and when the various constructs of a language should produce a program behavior. There are many ways of defining execution semantics. Natural language is often used to specify the execution semantics of languages commonly used in practice. A significant amount of academic research went into formal semantics of programming languages, which allow execution semantics to be specified in a formal manner. Results from this field of research have seen limited application to programming language design and implementation outside academia.

Type system

A type system defines how a programming language classifies values and expressions into *types*, how it can manipulate those types and how they interact. The goal of a type system is to verify and usually enforce a certain level of correctness in programs written in that language by detecting certain incorrect operations. Any decidable type system involves a trade-off: while it rejects many incorrect programs, it can also prohibit some correct, albeit unusual programs. In order to bypass this downside, a number of languages have *type loopholes*, usually unchecked casts that may be used by the programmer to explicitly allow a normally disallowed operation between different types. In most typed

languages, the type system is used only to type check programs, but a number of languages, usually functional ones, infer types, relieving the programmer from the need to write type annotations. The formal design and study of type systems is known as *type theory*.

Typed versus untyped languages

A language is *typed* if the specification of every operation defines types of data to which the operation is applicable, with the implication that it is not applicable to other types. For example, the data represented by "`this text between the quotes`" is a string. In most programming languages, dividing a number by a string has no meaning. Most modern programming languages will therefore reject any program attempting to perform such an operation. In some languages, the meaningless operation will be detected when the program is compiled ("static" type checking), and rejected by the compiler, while in others, it will be detected when the program is run ("dynamic" type checking), resulting in a runtime exception.

A special case of typed languages are the *single-type* languages. These are often scripting or markup languages, such as REXX or SGML, and have only one data type—most commonly character strings which are used for both symbolic and numeric data.

In contrast, an *untyped language*, such as most assembly languages, allows any operation to be performed on any data, which are generally considered to be sequences of bits of various lengths. High-level languages which are untyped include BCPL and some varieties of Forth.

In practice, while few languages are considered typed from the point of view of type theory (verifying or rejecting *all* operations), most modern languages offer a degree of typing. Many production languages provide means to bypass or subvert the type system.

Static versus dynamic typing

In *static typing* all expressions have their types determined prior to the program being run (typically at compile-time). For example, 1 and (2+2) are integer expressions; they cannot be passed to a function that expects a string, or stored in a variable that is defined to hold dates.

Statically typed languages can be either *manifestly typed* or *type-inferred*. In the first case, the programmer must explicitly write types at certain textual positions (for example, at variable declarations). In the second case, the compiler *infers* the types of expressions and declarations based on context. Most mainstream statically typed languages, such as C++, C# and Java, are manifestly typed. Complete type inference has traditionally been associated with less mainstream languages, such as Haskell and ML. However, many manifestly typed languages support partial type inference; for example, Java and C# both infer types in certain limited cases.

Dynamic typing, also called *latent typing*, determines the type-safety of operations at runtime; in other words, types are associated with *runtime values* rather than *textual expressions*. As with type-inferred languages, dynamically typed languages do not require the programmer to write explicit type annotations on expressions. Among other things, this may permit a single variable to refer to values of different types at different points in the program execution. However, type errors cannot be automatically detected until a piece of code is actually executed, potentially making debugging more difficult. Ruby, Lisp, JavaScript, and Python are dynamically typed.

Weak and strong typing

Weak typing allows a value of one type to be treated as another, for example treating a string as a number. This can occasionally be useful, but it can also allow some kinds of program faults to go undetected at compile time and even at runtime.

Strong typing prevents the above. An attempt to perform an operation on the wrong type of value raises an error. Strongly typed languages are often termed *type-safe* or *safe*.

An alternative definition for "weakly typed" refers to languages, such as Perl and JavaScript, which permit a large number of implicit type conversions. In JavaScript, for example, the expression `2 * x` implicitly converts `x` to a number, and this conversion succeeds even if `x` is `null`, `undefined`, an `Array`, or a string of letters. Such implicit conversions are often useful, but they can mask programming errors. *Strong* and *static* are now generally considered orthogonal concepts, but usage in the literature differs. Some use the term *strongly typed* to mean *strongly, statically typed*, or, even more confusingly, to mean simply *statically typed*. Thus C has been called both strongly typed and weakly, statically typed.

Standard library and run-time system

Most programming languages have an associated core library (sometimes known as the 'standard library', especially if it is included as part of the published language standard), which is conventionally made available by all implementations of the language. Core libraries typically include definitions for commonly used algorithms, data structures, and mechanisms for input and output.

A language's core library is often treated as part of the language by its users, although the designers may have treated it as a separate entity. Many language specifications define a core that must be made available in all implementations, and in the case of standardized languages this core library may be required. The line between a language and its core library therefore differs from language to language. Indeed, some languages are designed so that the meanings of certain syntactic constructs cannot even be described without referring to the core library. For example, in Java, a string literal is defined as an instance of the `java.lang.String` class; similarly, in Smalltalk, an anonymous function expression (a "block") constructs an instance of the library's `BlockContext` class. Conversely, Scheme contains multiple coherent subsets that suffice to construct the rest

of the language as library macros, and so the language designers do not even bother to say which portions of the language must be implemented as language constructs, and which must be implemented as parts of a library.

Design and implementation

Programming languages share properties with natural languages related to their purpose as vehicles for communication, having a syntactic form separate from its semantics, and showing *language families* of related languages branching one from another. But as artificial constructs, they also differ in fundamental ways from languages that have evolved through usage. A significant difference is that a programming language can be fully described and studied in its entirety, since it has a precise and finite definition. By contrast, natural languages have changing meanings given by their users in different communities. While constructed languages are also artificial languages designed from the ground up with a specific purpose, they lack the precise and complete semantic definition that a programming language has.

Many languages have been designed from scratch, altered to meet new needs, combined with other languages, and eventually fallen into disuse. Although there have been attempts to design one "universal" programming language that serves all purposes, all of them have failed to be generally accepted as filling this role. The need for diverse programming languages arises from the diversity of contexts in which languages are used:

- Programs range from tiny scripts written by individual hobbyists to huge systems written by hundreds of programmers.
- Programmers range in expertise from novices who need simplicity above all else, to experts who may be comfortable with considerable complexity.
- Programs must balance speed, size, and simplicity on systems ranging from microcontrollers to supercomputers.
- Programs may be written once and not change for generations, or they may undergo continual modification.
- Finally, programmers may simply differ in their tastes: they may be accustomed to discussing problems and expressing them in a particular language.

One common trend in the development of programming languages has been to add more ability to solve problems using a higher level of abstraction. The earliest programming languages were tied very closely to the underlying hardware of the computer. As new programming languages have developed, features have been added that let programmers express ideas that are more remote from simple translation into underlying hardware instructions. Because programmers are less tied to the complexity of the computer, their programs can do more computing with less effort from the programmer. This lets them write more functionality per time unit.

Natural language processors have been proposed as a way to eliminate the need for a specialized language for programming. However, this goal remains distant and its

benefits are open to debate. Edsger W. Dijkstra took the position that the use of a formal language is essential to prevent the introduction of meaningless constructs, and dismissed natural language programming as "foolish". Alan Perlis was similarly dismissive of the idea. Hybrid approaches have been taken in Structured English and SQL.

A language's designers and users must construct a number of artifacts that govern and enable the practice of programming. The most important of these artifacts are the language *specification* and *implementation*.

Specification

The **specification** of a programming language is intended to provide a definition that the language users and the implementors can use to determine whether the behavior of a program is correct, given its source code.

A programming language specification can take several forms, including the following:

- An explicit definition of the syntax, static semantics, and execution semantics of the language. While syntax is commonly specified using a formal grammar, semantic definitions may be written in natural language (e.g., as in the C language), or a formal semantics (e.g., as in Standard ML and Scheme specifications).
- A description of the behavior of a translator for the language (e.g., the C++ and Fortran specifications). The syntax and semantics of the language have to be inferred from this description, which may be written in natural or a formal language.
- A *reference* or *model* implementation, sometimes written in the language being specified (e.g., Prolog or ANSI REXX). The syntax and semantics of the language are explicit in the behavior of the reference implementation.

Implementation

An **implementation** of a programming language provides a way to execute that program on one or more configurations of hardware and software. There are, broadly, two approaches to programming language implementation: *compilation* and *interpretation*. It is generally possible to implement a language using either technique.

The output of a compiler may be executed by hardware or a program called an interpreter. In some implementations that make use of the interpreter approach there is no distinct boundary between compiling and interpreting. For instance, some implementations of BASIC compile and then execute the source a line at a time.

Programs that are executed directly on the hardware usually run several orders of magnitude faster than those that are interpreted in software.

One technique for improving the performance of interpreted programs is just-in-time compilation. Here the virtual machine, just before execution, translates the blocks of bytecode which are going to be used to machine code, for direct execution on the hardware.

Usage

Thousands of different programming languages have been created, mainly in the computing field. Programming languages differ from most other forms of human expression in that they require a greater degree of precision and completeness. When using a natural language to communicate with other people, human authors and speakers can be ambiguous and make small errors, and still expect their intent to be understood. However, figuratively speaking, computers "do exactly what they are told to do", and cannot "understand" what code the programmer intended to write. The combination of the language definition, a program, and the program's inputs must fully specify the external behavior that occurs when the program is executed, within the domain of control of that program.

A programming language provides a structured mechanism for defining pieces of data, and the operations or transformations that may be carried out automatically on that data. A programmer uses the abstractions present in the language to represent the concepts involved in a computation. These concepts are represented as a collection of the simplest elements available (called primitives). *Programming* is the process by which programmers combine these primitives to compose new programs, or adapt existing ones to new uses or a changing environment.

Programs for a computer might be executed in a batch process without human interaction, or a user might type commands in an interactive session of an interpreter. In this case the "commands" are simply programs, whose execution is chained together. When a language is used to give commands to a software application (such as a shell) it is called a scripting language.

Measuring language usage

It is difficult to determine which programming languages are most widely used, and what usage means varies by context. One language may occupy the greater number of programmer hours, a different one have more lines of code, and a third utilize the most CPU time. Some languages are very popular for particular kinds of applications. For example, COBOL is still strong in the corporate data center, often on large mainframes; FORTRAN in scientific and engineering applications; C in embedded applications and operating systems; and other languages are regularly used to write many different kinds of applications.

Various methods of measuring language popularity, each subject to a different bias over what is measured, have been proposed:

- counting the number of job advertisements that mention the language
- the number of books sold that teach or describe the language
- estimates of the number of existing lines of code written in the language—which may underestimate languages not often found in public searches
- counts of language references (i.e., to the name of the language) found using a web search engine.

Combining and averaging information from various internet sites, langpop.com claims that in 2008 the 10 most cited programming languages are (in alphabetical order): C, C++, C#, Java, JavaScript, Perl, PHP, Python, Ruby, and SQL.

Taxonomies

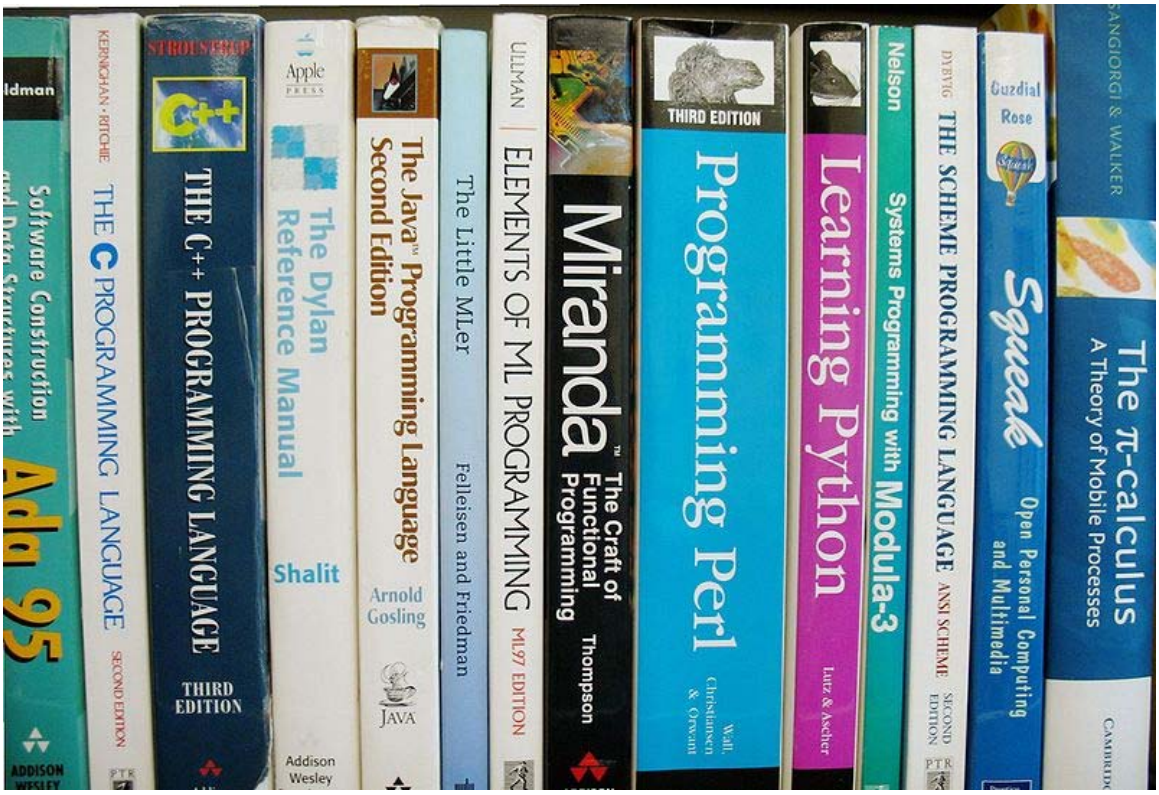
There is no overarching classification scheme for programming languages. A given programming language does not usually have a single ancestor language. Languages commonly arise by combining the elements of several predecessor languages with new ideas in circulation at the time. Ideas that originate in one language will diffuse throughout a family of related languages, and then leap suddenly across familial gaps to appear in an entirely different family.

The task is further complicated by the fact that languages can be classified along multiple axes. For example, Java is both an object-oriented language (because it encourages object-oriented organization) and a concurrent language (because it contains built-in constructs for running multiple threads in parallel). Python is an object-oriented scripting language.

In broad strokes, programming languages divide into *programming paradigms* and a classification by *intended domain of use*. Traditionally, programming languages have been regarded as describing computation in terms of imperative sentences, i.e. issuing commands. These are generally called imperative programming languages. A great deal of research in programming languages has been aimed at blurring the distinction between a program as a set of instructions and a program as an assertion about the desired answer, which is the main feature of declarative programming. More refined paradigms include procedural programming, object-oriented programming, functional programming, and logic programming; some languages are hybrids of paradigms or multi-paradigmatic. An assembly language is not so much a paradigm as a direct model of an underlying machine architecture. By purpose, programming languages might be considered general purpose, system programming languages, scripting languages, domain-specific languages, or concurrent/distributed languages (or a combination of these). Some general purpose languages were designed largely with educational goals.

A programming language may also be classified by factors unrelated to programming paradigm. For instance, most programming languages use English language keywords, while a minority do not. Other languages may be classified as being esoteric or not.

History



A selection of textbooks that teach programming, in languages both popular and obscure. These are only a few of the thousands of programming languages and dialects that have been designed in history.

Early developments

The first programming languages predate the modern computer. The 19th century had "programmable" looms and player piano scrolls which implemented what are today recognized as examples of domain-specific languages. By the beginning of the twentieth century, punch cards encoded data and directed mechanical processing. In the 1930s and 1940s, the formalisms of Alonzo Church's lambda calculus and Alan Turing's Turing machines provided mathematical abstractions for expressing algorithms; the lambda calculus remains influential in language design.

In the 1940s, the first electrically powered digital computers were created. The first high-level programming language to be designed for a computer was Plankalkül, developed for the German Z3 by Konrad Zuse between 1943 and 1945. However, it was not implemented until 1998 and 2000.

Programmers of early 1950s computers, notably UNIVAC I and IBM 701, used machine language programs, that is, the first generation language (1GL). 1GL programming was quickly superseded by similarly machine-specific, but mnemonic, second generation languages (2GL) known as assembly languages or "assembler". Later in the 1950s,

assembly language programming, which had evolved to include the use of macro instructions, was followed by the development of "third generation" programming languages (3GL), such as FORTRAN, LISP, and COBOL. 3GLs are more abstract and are "portable", or at least implemented similarly on computers that do not support the same native machine code. Updated versions of all of these 3GLs are still in general use, and each has strongly influenced the development of later languages. At the end of the 1950s, the language formalized as ALGOL 60 was introduced, and most later programming languages are, in many respects, descendants of Algol. The format and use of the early programming languages was heavily influenced by the constraints of the interface.

Refinement

The period from the 1960s to the late 1970s brought the development of the major language paradigms now in use, though many aspects were refinements of ideas in the very first Third-generation programming languages:

- APL introduced *array programming* and influenced functional programming.
- PL/I (NPL) was designed in the early 1960s to incorporate the best ideas from FORTRAN and COBOL.
- In the 1960s, Simula was the first language designed to support *object-oriented programming*; in the mid-1970s, Smalltalk followed with the first "purely" object-oriented language.
- C was developed between 1969 and 1973 as a *system programming* language, and remains popular.
- Prolog, designed in 1972, was the first *logic programming* language.
- In 1978, ML built a polymorphic type system on top of Lisp, pioneering *statically typed functional programming* languages.

Each of these languages spawned an entire family of descendants, and most modern languages count at least one of them in their ancestry.

The 1960s and 1970s also saw considerable debate over the merits of *structured programming*, and whether programming languages should be designed to support it. Edsger Dijkstra, in a famous 1968 letter published in the Communications of the ACM, argued that GOTO statements should be eliminated from all "higher level" programming languages.

The 1960s and 1970s also saw expansion of techniques that reduced the footprint of a program as well as improved productivity of the programmer and user. The card deck for an early 4GL was a lot smaller for the same functionality expressed in a 3GL deck.

Consolidation and growth

The 1980s were years of relative consolidation. C++ combined object-oriented and systems programming. The United States government standardized Ada, a systems

programming language derived from Pascal and intended for use by defense contractors. In Japan and elsewhere, vast sums were spent investigating so-called "fifth generation" languages that incorporated logic programming constructs. The functional languages community moved to standardize ML and Lisp. Rather than inventing new paradigms, all of these movements elaborated upon the ideas invented in the previous decade.

One important trend in language design for programming large-scale systems during the 1980s was an increased focus on the use of *modules*, or large-scale organizational units of code. Modula-2, Ada, and ML all developed notable module systems in the 1980s, although other languages, such as PL/I, already had extensive support for modular programming. Module systems were often wedded to generic programming constructs.

The rapid growth of the Internet in the mid-1990s created opportunities for new languages. Perl, originally a Unix scripting tool first released in 1987, became common in dynamic websites. Java came to be used for server-side programming, and bytecode virtual machines became popular again in commercial settings with their promise of "Write once, run anywhere" (UCSD Pascal had been popular for a time in the early 1980s). These developments were not fundamentally novel, rather they were refinements to existing languages and paradigms, and largely based on the C family of programming languages.

Programming language evolution continues, in both industry and research. Current directions include security and reliability verification, new kinds of modularity (mixins, delegates, aspects), and database integration such as Microsoft's LINQ.

The 4GLs are examples of languages which are domain-specific, such as SQL, which manipulates and returns sets of data rather than the scalar values which are canonical to most programming languages. Perl, for example, with its 'here document' can hold multiple 4GL programs, as well as multiple JavaScript programs, in part of its own perl code and use variable interpolation in the 'here document' to support multi-language programming.