Grant Agreement: 287829

Comprehensive Modelling for Advanced Systems of Systems

COMPASS

CML Definition 0

Deliverable Number: D23.1

Version: 1.0

Date: June 15, 2012

Public Document

http://www.compass-research.eu

Contributors:

Ana Cavalcanti, York
Joey Coleman, Aarhus
André Didier, UFPE
Peter Gorm Larsen, Aarhus
Álvaro Miyazawa, York
Marcel Oliveira, UFPE
Jim Woodcock, York

Editors:

Jim Woodcock
Álvaro Miyazawa

Reviewers:

Aarhus
Atego
Newcastle

Document History

Ver	Date	Author	Description
0.1	22-10-2011	Peter Gorm Larsen	Initial document version
0.2	26-10-2011	Jim Woodcock	CSP syntax added
0.3	06-01-2012	Jim Woodcock	CML_{-1} complete.
0.4	26-02-2012	Jim Woodcock	First major overhaul
0.5	06-03-2012	Alvaro Miyazawa	Updated the syntax (e.g., reintroduced implicit functions and operations, integrated CSP actions and VDM statements). Reviewed introduction.
0.6	06-03-2012	Alvaro Miyazawa	Fixed some issues (e.g., assignment of result of operation application. Commented introduction. Updated syntax. Included the lexical specification.
0.7	08-03-2012	Jim Woodcock	Draft for review.
1.0	26-03-2012	Jim Woodcock	Incorporating detailed review comments from Aarhus, Atego, and Newcastle.

Abstract

This deliverable contains the complete initial syntactic design for CML. This is the first contribution to providing the sound underpinnings and effective tool support for the COMPASS Modelling Language, CML. It has been produced to allow tool building for the language to get underway as soon as possible, and to get early feedback from the project's case studies. The language definition is inherited from the following baseline languages: VDM, CSP, and *Circus*. Apart from process algebraic combinators, there are no other architectural combinators. This part of the language design will be informed by practical studies of existing graphical notations and models for systems of systems, such as are being carried out in deliverable D22.1 (*Initial Report on SoS Architectural Models*).

Contents

1 Introduction

CML is the COMPASS Modelling Language, the first language specifically designed for modelling and analysing systems of systems (SoS). It is based on the following baseline languages: VDM [10], CSP [1], and *Circus* [2]. The main objective of workpackage WP 23 of the COMPASS project is to provide a complete design for CML, including integration of the baseline notations' syntax and semantics. This will be used as the basis for the development of analysis techniques and prototype tools development in Theme 3.

The first stage in the development of CML is to construct a syntactic definition, and this preliminary version of the language will allow tool development to commence and experiments to be conducted on industrial case studies.

This report contains the initial version of the language, CML_0. CML will be a modern, integrated formal method. As yet, there is no formal semantics: constructing this is the task of the rest of this workpackage. The syntax of CML_0 provides a language with rich state and operations, concurrency, communication, time, and object orientation. Apart from process algebraic combinators, no other architectural combinators have yet been proposed for CML. This part of the language design will be informed by practical studies of existing graphical notations and models for SoS, such as are being carried out in deliverable D22.1 (*Initial Report on SoS Architectural Models*).

1.1 An example CML process

To illustrate the language, we present a small example: modelling a simple register with arithmetic operations. The example is inspired by an existing example in VDM-SL [10]. The key difference in CML_0 is that the register is *reactive*. The specification in VDM of the register is an abstract datatype, and this is then encapsulated in a CSP-like process. The interface to the abstract datatype is activated through external channel communications.

This example illustrates the addition of CSP-like operators to basic VDM functionality. This additional functionality consists of the following:

- Variable scoping.
- Synchronisation.
- Input and output communications.
- Event prefixing.

- Sequential composition.
- Interleaving and synchronised parallel composition.
- Interrupt operator.

The simple requirements for our example are as follows.

1. The register holds one byte.
2. The register abstract datatype has the following interface:
   - initialise, load, read, add, and subtract.
3. The encapsulated operations are partial: adding and subtracting certain values may overflow or underflow.
4. The encapsulated abstract datatype is given a totalised external interface by signalling errors to the user.

These requirements are formalised by defining a process `RegisterProc` with the following outline:

```
types
...
functions
...
channels
...
chansets
...
process RegisterProc =
  begin
    state
      ...
    initial
      ...
    operations
      ...
    actions
      ...
  @
    ...
  end
```

The global `types` section allows us to define a required type of `Byte` as a subtype of integers:

```
types
  public Byte = nat
```

```
  inv n == (n <= 255);
```

We need to be able to describe the situations when adding two bytes will result in overflow:

```
functions
  oflow: (i,j:Byte) b: bool
  post b = (i+j > 255)
```

and when subtracting one byte from another will result in underflow.

```
  uflow: (i,j:Byte) b: bool
  post b = (i-j < 0)
```

The state encapsulated in our `RegisterProc` process consists of a single instance variable declaration:

```
state reg: Byte
```

We complete the description of the abstract datatype by defining the operations. First, the initialisation has a frame giving write access to the register; it is a total operation (the omitted precondition is by default true), and the postcondition sets the final value of `reg` to zero:

```
initial
  INIT ()
    frame wr reg: Byte
    post reg = 0
```

The load operation takes a single byte parameter `i` as the value to be loaded into the register:

```
operations
  LOAD (i: Byte)
    frame wr reg: Byte
    post reg = i;
```

Reading the register simply copies its value to the operation's output:

```
  READ () j: Byte
    frame rd reg: Byte
    post j = reg;
```

The add operation takes the byte to be added as an input. The precondition is that there will not be any overflow; the postcondition is that the new value of the register `reg` is the input added to the old value `reg~`:

```
  ADD(i: Byte)
```

```
    frame wr reg: Byte
    pre not oflow(reg,i)
    post reg = reg~ + i;
```

Similarly, the subtract operation takes the byte to be subtracted as an input. The precondition is that there will not be any underflow; the postcondition is that the new value of the register `reg` is the input subtracted from the old value `reg~`:

```
  SUB(i: Byte)
    frame wr reg: Byte
    pre not uflow(reg,i)
    post reg = reg~ - i
```

The `RegisterProc` process is connected to its environment with a number of channels, some used for synchronisation, and others used for communicating bytes. The environment starts the register process using the synchronisation channel `init`. The other channels are used to initiate the various operations on the register and indicate errors: the `load`, `add`, and `sub` channels are used to communicate inputs to the process, `read` is used to communicate a result to the environment, and `overflow` and `underflow` are used to signal, respectively, overflow and underflow of the register.

```
init, overflow, underflow
load, read, add, sub : Byte
```

The interface to the register process is made up from these channels and the values that they can communicate, and this is defined by the channel set `I`:

```
I = {| init, overflow, underflow, read, load, add, sub |}
```

The reactive behaviour of the register is given by the following main action for the process:

```
@ init -> INIT(); REG
```

Following the `init` synchronisation event, the process initialises the register using the `INIT` operation. Subsequent behaviour is given by the `REG` action. `REG` offers the environment an external choice between the different operations:

The first possibility is that the environment loads the register, and this is done by using the `load` channel.

```
load?i -> LOAD(i); REG
```

The value communicated by the environment is stored in the local variable `i`, which is then used as the parameter to the `LOAD` operation. There is then a recursive call to `REG`.

The second possibility is that the environment reads the register. But we need a variable to receive the output from the `READ` command.

```
dcl j: Byte @ j := READ(); read!j -> REG
```

A local variable `j` is declared; the output from `READ` is assigned to `j`; `j` is output through the `read` channel; and then there is a recursive call to `REG`.

The third possibility is that the environment adds a value to the register:

```
add?i -> ( oflow(reg,i) & overflow -> INIT(); REG
           []
           not oflow(reg,i) & ADD(i); REG )
```

The environment sends a value as input over the `add` channel; this is bound to the local variable `i`. What happens next depends on whether the add operation will overflow or not. The "`&`" syntax introduces a guarded action: if overflow is possible, then the guard `oflow(reg,i)` is true, and the following action is enabled. This signals the `overflow` event; it then resets the register; and finally recurses. However, if there will be no overflow, then the `ADD` operation is called, followed by a recursive call to the `REG` action.

Something similar happens with the fourth possibility: subtraction.

These alternative are put together with external choice:

```
actions
  REG =
    load?i -> LOAD(i); REG
    []
    var j: Byte @ j := READ(); read!j -> REG
    []
    add?i -> ( oflow(reg,i) & overflow -> INIT(); REG
               []
               not oflow(reg,i) & ADD(i); REG )
    []
    sub?i -> ( uflow(reg,i) & underflow -> INIT(); REG
               []
               not uflow(reg,i) & SUB(i); REG )
```

The whole process is given in Fig 1.

To show how we might use this process, we create a pair of named registers:

```
X = RegisterProc[[e <- x.e | e in set I]]
Y = RegisterProc[[e <- y.e | e in set I]]
```

In each definition, the external events in the interface channel set `I` are renamed, so that they are prefixed by `x` and `y`, respectively. The `[[...]]` syntax introduces a renaming comprehension. The two registers are then composed into the `REGS` process:

```
REGS = X ||| Y
```

The pair's behaviours are interleaved, so that

- There is no direct communication between the registers.
- But a user may compute with both.

The user interacts through the interface channel set. For example, if the user process is `COMPUTE`, then the interaction between the user and the registers is defined by

```
COMPUTE [| A union B  |] REGS
```

where

```
A = { x.e | e in set I }
B = { y.e | e in set I }
```

`COMPUTE` can now use both named resources, taking advantage of the communication channels `y.read`, `x.sub`, etc.

1.2 Informal Semantics

The informal semantics of CML_0 is based on the constituent baseline languages.

VDM is described in a very accessible way on Wikipedia [10]. The classic reference is Jones's book [7]. Wider software engineering issues are discussed in [5]. The definitive reference is the ISO Standard [9].

CSP is described, again in very accessible way, on Wikipedia [1]. The classic reference is Hoare's book [4]. A more extensive up-to-date reference is [6].

```
process RegisterProc =
  begin
    state reg: Byte

    initial
      INIT ()
      frame wr reg: Byte
      post reg = 0

    operations
      LOAD (i: Byte)
      frame wr reg: Byte
      post reg = i;

      READ () j: Byte
      frame rd reg: Byte
      post j = reg;

      ADD (i: Byte)
      frame wr reg: Byte
      pre not oflow(reg,i)
      post reg = reg~ + i;

      SUB (i: Byte)
      frame wr reg: Byte
      pre not oflow(reg,i)
      post reg = reg~ - i

    actions
      REG =
        load?i -> LOAD(i); REG
        [] dcl j: Byte @ j := READ(); read!j -> REG
        [] add?i -> ( oflow(reg,i) & overflow -> INIT(); REG
                      []
                      not oflow(reg,i) & ADD(i); REG )
        [] sub?i -> ( uflow(reg,i) & underflow -> INIT(); REG
                      []
                      not uflow(reg,i) & SUB(i); REG )
  @ init -> INIT(); REG
  end
```

Figure 1: The `RegisterProc` process in CML_0

The main tool for CSP is FDR2, and a description of FDR2's machine-readable version of CSP, as used in CML, is described in [3].

Sections 3– 19 contain the current grammar for CML_0. They describe all the main productions. Key syntactic constructs are described informally. Table 1 describes basic process constructors, while Table 2 describes their replicated versions. Table 3 describes the syntax of basic actions, Table 4 describes the different varieties of parallel action constructors, and Table 5 describes the replicated action constructors. Basic VDM statements are described in Table 6, and VDM control statements are described in Table 7. Finally a complete cross-reference is contained in the Index, with a list of all definitions and applied occurrences of productions.

The next version of CML will provide a formal account of the integrated semantics of the two languages.

2 Language issues

The current status of CML is an initial design that makes a first approximation to the unification of the syntax of VDM and CSP, using semantic inspiration from the *Circus* family of languages to guide the syntax of real-time and object-oriented facilities. During this design process, a number of issues were raised that must be addressed in later phases of the language design.

CML type system. There's a difference between types in Z, as used in *Circus*, and types in VDM, as used in CML. The two most striking differences are: (i) types are also sets in Z, whereas they are not in VDM; and (ii) types are maximal in Z, whereas in VDM, they are not. In practice, typechecking in VDM requires discharging proof obligations, and so typechecking is, in effect, also maximal.

Invariants. Invariants are part of operation definitions in Z, but not in VDM. The underlying UTP semantics takes the Z position on invariants. The consquences of the required change to accommodate VDM must be carefully considered.

Polymorphism vs genericity. In Z and so in *Circus*, we have generic constructs, such as generic processes and generic classes: families of definitions indexed by a formal type parameter. This doesn't seem to differ much from VDM's notion of polymorphism, which is rather restricted.

In the initial language design, generics and polymorphism have been removed.

Static definitions. VDM++ allows some static definitions to live outside the definition of a defining class. This seems convenient, but raises issues about modularity and name spaces. For the moment, CML_0 will allow global definitions.

Mathematical toolkit. In Z, and so in *Circus*, there are mechanisms for introducing global, generic, mixed-fixity operators. This means that the basic operators for numbers, sets, relations, functions, sequences, and bags are not part of the kernel language, but belong in a library of definitions called the mathematical toolkit. This modularises the language and reduces the semantics of these operators to mere definitions. It also makes the language more flexible by facilitating extensibility, which is particularly useful for engineering new domains. This approach will not be followed in CML, where extensibility will be restricted to prefix operators and functions, and the mathematical datatypes will remain primitive parts of the language.

New expressions. We have removed `new` expressions, allowing only `new` statements, as nondeterministic expressions are semantically problematic.

Parameters. Parameter-passing mechanisms in VDM is by value only. More general mechanisms are allowed in *Circus*. CML will require a reference semantics for its objects, and this will provide a call-by-reference parameter mechanism. There is still a debate about whether this will be restricted to the constituent-system level, or whether it will permeate the architecture of an SoS.

Logic. VDM uses the Logic of Partial Functions (LPF); Overture-VDM uses McCarthy logic (left-to-right evaluation); Z uses semi-classical logic. What should CML use? Specifically, what is the treatment of undefined expressions and what is the treatment of undefined predicates? It has been decided that CML will use McCarthy logic for compatibility with Overture-VDM. This will entail a thorough treatment of the proof obligations for definedness for the entire language—both the VDM and CSP components.

3 CML_0 Grammar

program =
    program paragraph, { program paragraph };

program paragraph =
    class declaration
    | process declaration
    | channel declaration
    | chanset declaration
    | global declaration;

3.1 Class declaration

class declaration =
    '`class`', identifier, '`=`', class body;

3.2 Process declaration

process declaration =
    '`process`', identifier, '`=`',
        process definition;

3.3 Channel declaration

channel declaration =
    channel definition;

3.4 Chanset declaration

chanset declaration =
    identifier, '`=`', chanset expression ;

3.5 Global declaration

global declaration =

type declaration
|function declaration
|value declaration;

4 Class body

class body =
    [‘`extends`’, identifier],
    ‘`begin`’,
        { class paragraph },
        [‘`state`’, instance variable declaration],
        { class paragraph },
        [‘`initial`’, operation declaration],
        { class paragraph },
    ‘`end`’;

5 Process definition

process definition =
    [declaration, '@'], process;

declaration =
    single type declaration,{ ';', single type declaration };

single type declaration =
    identifier, { ',', identifier }, ':', type;

6 Channel definition

channel definition =
    channel name declaration, { ';', channel name declaration } ;

7 Chanset expression

Currently, `{|...|}` is used for chanset comprehension to avoid confusion with general set comprehensions.

chanset expression =

identifier
| '`{`', [identifier, { '`,`', identifier }], '`}`'
| '`{|`', [identifier, { '`,`', identifier }], '`|}`'
| chanset expression, '`union`', chanset expression
| chanset expression, '`inter`', chanset expression
| chanset expression, '\', chanset expression
| '`{|`', identifier, { '`.`' expression }, '`|`' bind list, ['`@`', expression], '`|}`' ;

8 Class paragraph

class paragraph =
    paragraph;

paragraph =
    type declaration
    | value declaration
    | function declaration
    | operation declaration;

9 Process

process =
    'begin',
        { process paragraph },
        ['state', instance variable declaration],
        { process paragraph }
    '@',
        action,
    'end'
    | process, ';', process
    | process, '[]', process
    | process, '|~|', process
    | process, '[|', chanset expression, '|]', process
    | process, '[', chanset expression, '||', chanset expression, ']', process
    | process, '||', process
    | process, '|||', process
    | process, '/', [expression], '\', process
    | process, '[', [expression], '>', process
    | process, '\', chanset expression
    | process, 'startsby', expression
    | process, 'endsby', expression
    | '(', declaration, '@', process definition, ')', '(', { expression }, ')'
    | identifier, ['(', { expression }, ')']
    | process, renaming expression
    | ';', replication declaration, '@', process
    | '[]', replication declaration, '@', process
    | '|~|', replication declaration, '@', process
    | '[|', chanset expression, '|]', replication declaration, '@', process
    | '||', replication declaration, '@', '[', chanset expression, ']', process
    | '||', replication declaration, '@', process
    | '|||', replication declaration, '@', process
    | '(', process,')';

renaming expression =
    renaming enumeration
    | renaming comprehension ;

renaming enumeration =
    '[[', renaming pair, { ',', renaming pair }, ']]' ;

renaming pair =

identifier, { '.', expression },
'<-',
identifier, { '.', expression } ;

renaming comprehension =
'[[', renaming pair, '|' bind list, ['@', expression], ']]';

The basic process `process P = begin state v: T; ... @ A end` may declare a number of actions, functions, operations, types, values and state variables as well as a mandatory main action (`@ A`). The behaviour of the process is defined by its main action.

operator	syntax	hints of the semantics
sequential composition	`A ; B`	behave like `A` until `A` terminates, then behave like `B`
external choice	`A [] B`	offer the environment the choice between `A` and `B`.
internal choice	`A |~| B`	nondeterministically behave either like `A` or `B`.
generalised parallelism	`A [| cs |] B`	execute `A` and `B` in parallel synchronising on the events in `cs`.
alphabetised parallelism	`A [ X || Y ] B`	execute `A` and `B` in parallel synchronising in the intersection of `X` and `Y`. `A` is only allowed to communicate on `X` and `B` is only allowed to communicate on `Y`.
synchronous parallelism	`A || B`	execute `A` and `B` in parallel synchronising on all events.
interleaving	`A ||| B`	execute `A` and `B` in parallel without synchronising.
interrupt	`A /\ B`	behave as `A` until `B` takes control, then behave like `B`.
timed interrupt	`A /e\ B`	behave as `A` for `e` time units, then behave as `B`.
untimed timeout	`A [> B`	offer `A`, but may nondeterministically stop offering `A` and offer `B` at any time.
timeout	`A [ e > B`	offer `A` for `e` time units, then offer `B`.
hiding	`A \ cs`	behave as `A` with the events in `cs` hidden
start deadline	`A startby e`	`A` must execute an observable event within `e` time units. Otherwise, the process is infeasible.
end deadline	`A endsby e`	`A` must terminate within `e` time units. Otherwise, the process is infeasible
process instantiation	`(v:T @ A)(e)` or `A(e)`	behaves as `A` where the formal parameters (`v`) are instantiated to `e`.
channel renaming	`A[[c<-nc]]`	behaves as `A` with event `c` renamed to `nc`.

Table 1: Process constructors.

operator	syntax	hints of the semantics
replicated sequential composition	`i:e @ A(i)`	`e` must be a sequence, for each `i` in the sequence, `A(i)` is executed in order.
replicated external choice	`[]i:e @ A(i)`	offer the environment the choice of all processes `A(i)` such that `i` is in the set `e`.
replicated internal choice	`|~|i:e @ A(i)`	nondeterministically behave as `A(i)` for any `i` in the set `e`.
replicated generalised parallelism	`[|cs|]i:e @ A(i)`	execute all processes `A(i)` (for `i` in the set `e`) in parallel synchronising on the events in `cs`.
replicated alphabetised parallelism	`||i:e@[cs(i)]A(i)`	execute all processes `A(i)` in parallel synchronising on the intersection of all `cs(i)`. Each process `A(i)` can only perform events in `cs(i)`.
replicated synchronous parallelism	`||i:e @ A(i)`	execute all processes `A(i)` in parallel synchronising on all events.
replicated interleaving	`|||i:e @ A(i)`	execute all processes `A(i)` in parallel without synchronising on any events.

Table 2: Replicated process constructors.

10 Channel name declaration

channel name declaration =
    declaration
    | identifier, { ‘,’, identifier } ;

11 Type declaration

type declaration =
    ‘types’, type definition, { ‘;’, type definition };

type definition =
    [qualifier], identifier, ‘=’, type, [invariant]
    | [qualifier], identifier, ‘::’, field list, [invariant];

qualifier =
    ‘private’
    | ‘protected’
    | ‘public’
    | ‘logical’;

type =
    bracketed type
    | basic type
    | quote type
    | composite type
    | union type
    | product type
    | optional type
    | set type
    | seq type
    | map type
    | function type
    | type name ;

bracketed type =
    ‘(’, type, ‘)’;

basic type =
    ‘bool’ | ‘nat’ | ‘nat1’ | ‘int’ | ‘rat’ | ‘real’ | ‘char’ | ‘token’;

quote type =
    quote literal;

composite type =
    ‘compose’, identifier, ‘of’, field list, ‘end’;

field list =
    { field };

field =

[identifier, ‘:’], type
| [identifier, ‘:-’], type;

union type =
type, ‘|’, type, { ‘|’, type };

product type =
type, ‘*’, type, { ‘*’, type };

optional type =
‘[’, type, ‘]’;

set type =
‘set of’, type;

seq type =
seq0 type
| seq1 type;

seq0 type =
‘seq of’, type;

seq1 type =
‘seq1 of’, type;

map type =
general map type
| injective map type;

general map type =
‘map’, type, ‘to’, type;

injective map type =
‘inmap’, type, ‘to’, type;

function type =
partial function type
| total function type;

partial function type =
discretionary type, ‘+>’, type;

total function type =
discretionary type, ‘->’, type;

discretionary type =
type
| ‘(’, ‘)’;

type name =
    name;

invariant =
    ‘inv’, invariant initial function;

invariant initial function =
    pattern, ‘==’, expression;

12 Value declaration

value declaration =
    ‘`values`’, value definition, { ‘`;`’, value definition };

value definition =
    [qualifier], pattern, [‘`:`’, type], ‘`=`’, expression
    |[qualifier], pattern, [‘`:`’, type], ‘`in`’, expression;

13 Function declaration

function declaration =
‘functions’,[function definition], { ‘;’, function definition };

function definition =
explicit function definition
|implicit function definition;

explicit function definition =
[qualifier], identifier, ‘:’, function type,
identifier, parameters list, ‘==’, function body,
[‘pre’, expression],
[‘post’, expression],
[‘measure’, name];

implicit function definition =
[qualifier], identifier, parameter types, identifier type pair list,
[‘pre’, expression], ‘post’, expression ;

parameters list =
parameters, { parameters };

parameters =
‘(’, [pattern list], ‘)’;

function body =
expression
| ‘is subclass responsibility’
| ‘is not yet specified’;

parameter types =
‘(’, [pattern list, ‘:’, type, { ‘,’, pattern list, ‘:’, type }], ‘)’;

identifier type pair list =
identifier,‘:’,type, { ‘,’, identifier,‘:’,type } ;

14 Operation declaration

Operations do not include reactive constructs.

operation declaration =
‘`operations`’, operation definition, { ‘`;`’, operation definition };

operation definition =
explicit operation definition
|implicit operation definition ;

explicit operation definition =
[qualifier], identifier, ‘`:`’, operation type,
identifier, parameters, ‘`==`’, operation body,
[‘`pre`’, expression],
[‘`post`’, expression];

operation type =
discretionary type, ‘`==>`’, discretionary type;

operation body =
action
| ‘`is subclass responsibility`’
| ‘`is not yet specified`’;

externals =
‘`frame`’, var information, { var information };

var information =
mode, name list, [‘`:`’, type];

mode =
‘`rd`’ | ‘`wr`’;

implicit operation definition =
[qualifier], identifier, parameter types,
[identifier type pair list],
[externals],
[‘`pre`’, expression],
‘`post`’, expression;

15 Process paragraph

process paragraph =
    paragraph
    | identifier, '=', paragraph action
    | nameset, identifier, '=', nameset expression ;

paragraph action =
    action
    | declaration, '@', paragraph action ;

action =
'Skip' | 'Stop' | 'Chaos' | 'Div'
| 'Wait', expression
| communication, '->', action
| expression, '&', action
| action, ';', action
| action, '[]', action
| action, '|~|', action
| action, '/', [expression], '\', action
| action, '[', [expression], '>', action
| action, '\', chanset expression
| action, 'startby', expression
| action, 'endby', expression
| action, renaming expression
| 'mu', identifier, { ',', identifier }, '@', action, { ',', action }
| parallel actions
| parametrised actions
| instantiated actions
| replicated actions
| let statement
| block statement
| control statements
| '(', action, ')' ;

communication =
identifier, { communication parameter } ;

communication parameter =
'?', parameter
|'?', parameter, ':', expression
|'!', expression
|'.', expression ;

parameter =
identifier
|'mk_', '(', parameter, { ',', parameter }, ')'
|'mk_', name, '(', { parameter }, ')';

parallel actions =

operator	syntax	hints of the semantics
termination	`Skip`	terminate immediately
deadlock	`Stop`	
chaos	`Chaos`	can always choose to communicate or reject any event
divergence	`Div`	
delay	`Wait e`	does nothing until `e` time units have passed, and then terminates.
prefixing	`c!e?x:P(x) -> A`	offers the environment a choice of events of the form `c.e.p`, where `p in set {x | x: T @ P(x)}`.
guarded action	`g&A`	if `g` is true, behave like `A`, otherwise, behave like `Stop`.
sequential composition	`A ; B`	behave like `A` until `A` terminates, then behave like `B`
external choice	`A [] B`	offer the environment the choice between `A` and `B`.
internal choice	`A |~| B`	nondeterministically behave either like `A` or `B`.
interrupt	`A /\ B`	behave as `A` until `B` takes control, then behave like `B`.
timed interrupt	`A /e\ B`	behave as `A` for `e` time units, then behave as `B`.
untimed timeout	`A [> B`	offer `A`, but may nondeterministically stop offering `A` and offer `B` at any time.
timeout	`A [ e > B`	offer `A` for `e` time units, then offer `B`.
hiding	`A \ cs`	behave as `A` with the events in `cs` hidden
start deadline	`A startby e`	`A` must execute an observable event within `e` time units. Otherwise, the process is infeasible.
end deadline	`A endsby e`	`A` must terminate within `e` time units. Otherwise, the process is infeasible
channel renaming	`A[[c<-nc]]`	behaves as `A` with event `c` renamed to `nc`.
recursion	`mu X,Y @ F(X,Y),G(X,Y)`	explicit definition of mutually recursive actions.

Table 3: Action constructors.

action, '[||', chanset expression, '|', chanset expression, '||]', action
| action, '|||', action
| action '[|', nameset expression, '|', nameset expression, '|]', action
| action '||' action,
| action '[', nameset expression, '|', chanset expression, '||',
chanset expression, '|', nameset expression, ']', action
| action '[', chanset expression, '||', chanset expression, ']', action
| action '[|', nameset expression, '|', chanset expression, '|',
nameset expression, '|]', action
| action '[|', chanset expression, '|]', action;

parametrised actions =
parametrisation, { ';', parametrisation }, '@', action;

parametrisation =
('val' |'res' |'vres'), single type declaration;

instantiated actions =
'(', declaration, '@', action, ')', '(', expression, { ',', expression }, ')'
| '(', parametrised action, ')', '(', expression, { ',', expression }, ')';

operator	syntax	hints of the semantics
interleaving	`A [|| ns1 | ns2 ||] B`	execute `A` and `B` in parallel without synchronising. `A` can modify the state components in `ns1` and `B` can modify the state components in `ns2`.
interleaving (no state)	`A ||| B`	execute `A` and `B` in parallel without synchronising. Neither `A` nor `B` change the state.
synchronous parallelism	`A [| ns1 | ns2|] B`	execute `A` and `B` in parallel synchronising on all events. `A` can modify the state components in `ns1` and `B` can modify the state components in `ns2`.
synchronous parallelism (no state)	`A || B`	execute `A` and `B` in parallel synchronising on all events. Neither `A` nor `B` change the state.
alphabetised parallelism	`A [ns1| X || Y |ns2] B`	execute `A` and `B` in parallel synchronising in the intersection of `X` and `Y`. `A` in only allowed to communicate on `X` and `B` is only allowed to communicate on `Y`. `A` can modify the state components in `ns1` and `B` can modify the state components in `ns2`.
alphabetised parallelism (no state)	`A [X || Y] B`	execute `A` and `B` in parallel synchronising in the intersection of `X` and `Y`. `A` in only allowed to communicate on `X` and `B` is only allowed to communicate on `Y`. Neither `A` nor `B` change the state.
generalised parallelism	`A [|ns1 | cs | ns2|] B`	execute `A` and `B` in parallel synchronising on the events in `cs`. `A` can modify the state components in `ns1` and `B` can modify the state components in `ns2`.
generalised parallelism (no state)	`A [| cs |] B`	execute `A` and `B` in parallel synchronising on the events in `cs`. Neither `A` nor `B` change the state.

Table 4: Parallel action constructors.

replicated actions =
‘;’, replication declaration, ‘@’, action
| ‘[]’, replication declaration, ‘@’, action
| ‘|~|’, replication declaration, ‘@’, action
| ‘[||’, nameset expression, ‘||]’, replication declaration, ‘@’, action
| ‘|||’, replication declaration, ‘@’, ‘[’, nameset expression, ‘]’, action
| ‘[|’, chanset expression ‘|]’, replication declaration, ‘@’,
‘[’ , nameset expression, ‘]’, action
| ‘||’, replication declaration, ‘@’,
‘[’, nameset expression, ‘|’, chanset expression, ‘]’, action
| ‘||’, replication declaration, ‘@’, ‘[’, nameset expression, ‘]’, action
;

replication declaration =
(single type declaration | single expression declaration), ‘;’,
{ (single type declaration | single expression declaration) } ;

single expression declaration =
identifier, { ‘,’, identifier }, ‘:’, expression ;

15.1 Local Binding Statements

let statement =
‘let’, local definition, { ‘,’, local definition },
‘in’, action;

local definition =
value definition
| function definition;

15.2 Block and Assignment Statements

To be clarified

block statement =
‘(’, [dcl statement], action, ‘)’;

dcl statement =
‘dcl’, assignment definition, { ‘,’, assignment definition }, ‘@’;

assignment definition =
identifier, ‘:’, type, [‘:=’, expression]
|identifier, ‘:’, type, [‘in’, expression];

operator	syntax	hints of the semantics
replicated sequential composition	`i:e @ A(i)`	`e` must be a sequence, for each `i` in the sequence, `A(i)` is executed in order.
replicated external choice	`[]i:e @ A(i)`	offer the environment the choice of all actions `A(i)` such that `i` is in the set verb"e".
replicated internal choice	`|~|i:e @ A(i)`	nondeterministically behave as `A(i)` for any `i` in the set `e`.
replicated interleaving	`|||i:e @ [ns(i)]A(i)`	execute all actions `A(i)` in parallel without synchronising on any events. Each action `A(i)` can only modify the state components on `ns(i)`.
replicated generalised parallelism	`[|cs|]i:e @ [ns(i)] A(i)`	execute all action `A(i)` (for `i` in the set `e`) in parallel synchronising on the events in `cs`.Each action `A(i)` can only modify the state components on `ns(i)`.
replicated alphabetised parallelism	`||i:e@[ns(i) | cs(i)]A(i)`	execute all processes `A(i)` in parallel synchronising on the intersection of all `cs(i)`. Each process `A(i)` can only perform events in `cs(i)` and can only modify the state components in `ns(i)`.
replicated synchronous parallelism	`||i:e @ [ns(i)] A(i)`	execute all processes `A(i)` in parallel synchronising on all events. Each action `A(i)` can only modify the state components on `ns(i)`.

Table 5: Replicated action constructors.

general assign statement =
    assign statement
    | multiple assign statement;

assign statement =
    state designator, ‘:=’, expression;

multiple assign statement =
    ‘atomic’, ‘(’,
      assign statement, ‘;’, assign statement, [{ ‘;’,assign statement }],
    ‘)’;

15.3 Call and Return Statements

call statement =
    call
    | state designator, ‘:=’, call;

call =
    [object designator, ‘.’], name, ‘(’, [expression list], ‘)’;

object designator =
    name
    | self expression
    | object field reference
    | object apply;

object field reference =
    object designator, ‘.’, identifier;

object apply =
    object designator, ‘(’, [expression list], ‘)’;

return statement =
    ‘return’, [expression];

15.4 The Specification Statement

specification statement =
    ‘[’, implicit operation body, ‘]’;

implicit operation body =

[externals],
[‘pre’, expression],
‘post’, expression;

15.5 The New Statement

new statement =
state designator, ‘:=’, ‘new’, name, ‘(’, [expression list], ‘)’;

15.6 Control Statements

control statements =
non-deterministic if statement
| if statement
| cases statement
| general assign statement
| call statement
| specification statement
| return statement
| new statement
| non-deterministic do statement
| sequence for loop
| set for loop
| index for loop
| while loop ;

15.7 Dijkstra’s guarded commands

non-deterministic if statement =
‘if’ expression, ‘->’, action, [{ ‘|’, expression, ‘->’, action }], ‘end’;

non-deterministic do statement =
‘do’ expression, ‘->’, action, [{ ‘|’, expression, ‘->’, action }], ‘end’;

15.8 Conditional Statements

if statement =

operator	syntax	hints of the semantics
let	`let p=e in a`	evaluate the action `a` in the environment where `p` is associated to `e`.
block	`(dcl v: T := e; a)`	declare the local variable `v` of type `T` (optionally) initialised to `e` and evaluate action `a` in this context.
assignment	`v:=e`	
multiple assignment	`atomic (v1:=e1,...,vn:=en)`	assignments are executed atomically with respect to the state invariant.
call	`obj.op(p) or op(p) or act(p) or v := obj.op(p) or v := op(p)`	execute operation `op` of an object `obj` (1) or of the current object `obj` (2) with the parameters `p`. (3) execute action `act` with parameters `p`. Optionally, assign the value returned by an operations to a variable.
return	`return e` or `return`	terminates the evaluation of an operation possibly yielding a value `e`.
specification	`[frame wr v1: T1 rd v2: T2 pre P1(v1,v2) post P2(v1,v1~,v2,v2~)]`	`frame` clause lists variables that are manipulated by the specification statement (read or write). The `pre` clause defines a predicate that describes the precondition of the statement in terms of the initial values of the variables in the frame, and the `post` clause defines a predicate that describes the post condition of the statement in terms of the initial and final values of the variables in the frame. The initial value of a variable `v1` is denoted by `v1~`.
new	v:= new c	instantiate a new object of class `c` and assign it to `v`.

Table 6: VDM statements.

‘if’, expression, ‘then’,
action,
{ elseif statement },
[‘else’,
action];

elseif statement =
‘elseif’, expression, ‘then’,
action;

cases statement =
‘cases’, expression, ‘:’,
cases statement alternatives,
[‘,’, others statement],
‘end’;

cases statement alternatives =
cases statement alternative,
{ ‘,’, cases statement alternative };

cases statement alternative =
pattern list, ‘->’, action;

others statement =
‘others’, ‘->’, action;

15.9 Loop Statements

sequence for loop =
‘for’, pattern bind, ‘in’, [‘reverse’], expression, ‘do’, action;

set for loop =
‘for’, ‘all’, pattern, ‘in set’, expression, ‘do’, action;

index for loop =

‘for’, identifier, ‘=’, expression, ‘to’, expression, [‘by’, expression], ‘do’, action ;

while loop =
‘while’, expression, ‘do’, action;

operator	syntax	hints of the semantics

nondeterministic if statement	`if e1 -> a1`
	`| e2 -> a2`
	`| ...`
	`end`
		evaluate all guards `ei`, if none is true, the statement diverges. If one or more guards are true, one of the associated actions is executed nondeterministically.

if statement	`if e1 then a1`
	`elseif e2 then a2`
	`...`
	`else an`
		the boolean expressions `ei` are evaluated in order. When the first `ei` is evaluated to true, the associated action is executed. If no `ei` is evaluate to true, the action in the `else` clause is executed.

cases statement	`cases e:`
	`  p1 -> a1,`
	`  p2 -> a2,`
	`  ...,`
	`  others -> an`
	`end`
		The expression `e` is matched against each pattern `pi` in order. The result of the cases statement is the first action `ai` whose associated pattern `pi` matches the expression `e`. If not pattern is matched, the `others` clause is executed.

nondeterministic do statement	`do e1 -> a1`
	`| e2 -> a2`
	`| ...`
	`end`
		if all guards `ei` evaluate to false, terminate. Otherwise, chose nondeterministically one guard that evaluates to true, execute the associated action, and repeat the do statement.

sequence for loop	`for e in s do a`	for each expression `e` in the sequence `s`, execute action `a`.
set for loop	`for all e in set S do a`	for each expression `e` in the set `S`, execute action `a`.
index for loop	`for i=e1 to e2 by e3 do a`	execute action `a` for each integer `i` in the range `{e1,...,e2}` such that `i = e1+(n*e3)` (where `n` is a natural number).
while loop	`while e do a`	execute action `a` while the boolean expression `e` evaluates to true.

Table 7: Control statements.

16 Instance variable declaration

instance variable declaration =
    instance variable definition, { ';', instance variable definition };

instance variable definition =
    [qualifier], assignment definition
    | invariant definition;

invariant definition =
    'inv', expression;

17 Expressions

expression list =
    expression, { ‘,’, expression };

expression =
    bracketed expression
    | let expression
    | if expression
    | cases expression
    | unary expression
    | binary expression
    | quantified expression
    | set enumeration
    | set comprehension
    | set range expression
    | sequence enumeration
    | sequence comprehension
    | subsequence
    | map enumeration
    | map comprehension
    | tuple constructor
    | record constructor
    | apply
    | field select
    | tuple select
    | lambda expression
    | self expression
    | general is expression
    | precondition expression
    | isofclass expression
    | name
    | old name
    | symbolic literal;

17.1 Bracketed Expressions

bracketed expression =
    ‘(’, expression, ‘)’;

17.2 Local Binding Expressions

let expression =
    'let', local definition, { ',', local definition },
    'in', expression ;

17.3 Conditional Expressions

if expression =
    'if', expression, 'then',
      expression,
    { elseif expression },
    'else',
      expression ;

elseif expression =
    'elseif', expression, 'then',
      expression;

cases expression =
    'cases', expression, ':',
      cases expression alternatives,
      [',', others expression],
    'end';

cases expression alternatives =
    cases expression alternative, { ',', cases expression alternative };

cases expression alternative =
    pattern list, '->', expression;

others expression =
    'others', '->', expression;

17.4 Unary Expressions

unary expression =
    prefix expression
    | map inverse;

prefix expression =
    unary operator, expression;

unary operator =
    unary plus
    | unary minus
    | arithmetic abs
    | floor
    | not
    | set cardinality
    | finite power set
    | distributed set union
    | distributed set intersection
    | sequence head
    | sequence tail
    | sequence length
    | sequence elements
    | sequence indices
    | sequence reverse
    | distributed sequence concatenation
    | map domain
    | map range
    | distributed map merge;

unary plus =
    '+';

unary minus =
    '-';

arithmetic abs =
    'abs';

floor =
    'floor';

not =
    'not';

set cardinality =
    'card';

finite power set =
    'power';

distributed set union =
    'dunion';

distributed set intersection =
    'dinter';

sequence head =
    'hd';

sequence tail =
    'tl';

sequence length =
    'len';

sequence elements =
    'elems';

sequence indices =
    'inds';

sequence reverse =
    'reverse';

distributed sequence concatenation =
    'conc';

map domain =
    'dom';

map range =
    'rng';

distributed map merge =
    'merge';

map inverse =
    'inverse', expression;

17.5 Binary Expressions

binary expression =
    expression, binary operator, expression;

binary operator =

arithmetic plus
| arithmetic minus
| arithmetic multiplication
| arithmetic divide
| arithmetic integer division
| arithmetic rem
| arithmetic mod
| less than
| less than or equal
| greater than
| greater than or equal
| equal
| not equal
| or
| and
| imply
| logical equivalence
| in set
| not in set
| subset
| proper subset
| set union
| set difference
| set intersection
| sequence concatenate
| map or sequence modify
| map merge
| map domain restrict to
| map domain restrict by
| map range restrict to
| map range restrict by
| composition
| iterate;

arithmetic plus =
‘+’;

arithmetic minus =
‘-’;

arithmetic multiplication =
‘*’;

arithmetic divide =
    '/';

arithmetic integer division =
    'div';

arithmetic rem =
    'rem';

arithmetic mod =
    'mod';

less than =
    '<';

less than or equal =
    '<=';

greater than =
    '>';

greater than or equal =
    '>=';

equal =
    '=';

not equal =
    '<>';

or =
    'or';

and =
    'and';

imply =
    '=>';

logical equivalence =
    '<=>';

in set =
    'in set';

not in set =
    'not in set';

subset =
    'subset';

proper subset =
    'psubset';

set union =
    'union';

set difference =
    '\';

set intersection =
    'inter';

sequence concatenate =
    '^';

map or sequence modify =
    '++';

map merge =
    'munion';

map domain restrict to =
    '<:';

map domain restrict by =
    '<-:';

map range restrict to =
    ':>';

map range restrict by =
    ':->';

composition =
    'comp';

iterate =
    '**';

17.6 Quantified Expressions

quantified expression =

all expression
| exists expression
| exists unique expression
| iota expression;

all expression =
‘forall’, bind list, ‘@’, expression;

exists expression =
‘exists’, bind list, ‘@’, expression;

exists unique expression =
‘exists1’, bind, ‘@’, expression;

iota expression =
‘iota’, bind, ‘@’, expression;

17.7 Set Expressions

set enumeration =
‘{’, [expression list], ‘}’;

set comprehension =
‘{’, expression, ‘|’, bind list, [‘@’, expression], ‘}’;

set range expression =
‘{’, expression, ‘,’, ‘...’, ‘,’, expression, ‘}’;

17.8 Sequence Expressions

sequence enumeration =
‘[’, [expression list], ‘]’;

sequence comprehension =
‘[’, expression, ‘|’, set bind, [‘@’, expression], ‘]’;

subsequence =
expression, ‘(’, expression, ‘,’, ‘...’, ‘,’, expression, ‘)’;

17.9 Map Expressions

map enumeration =

‘{’, maplet, { ‘,’, maplet }, ‘}’
| ‘{’, ‘|->’, ‘}’;

maplet =
expression, ‘|->’, expression;

map comprehension =
‘{’, maplet, ‘|’, bind list, [‘@’, expression], ‘}’;

17.10 The Tuple Constructor Expression

tuple constructor =
‘mk_’, ‘(’, expression, ‘,’, expression list, ‘)’;

17.11 Record Expressions

record constructor =
‘mk_’,[1]name, ‘(’, [expression list], ‘)’;

17.12 Apply Expressions

apply =
expression, ‘(’, [expression list], ‘)’;

field select =
expression, ‘.’, identifier;

tuple select =
expression, ‘.#’, numeral;

17.13 The Lambda Expression

lambda expression =
‘lambda’, type bind list, ‘@’, expression;

[1] **Note:** no delimiter is allowed

17.14 The Self Expression

self expression =
    'self';

17.15 The Is Expression

general is expression =
    is expression
    | type judgement;

is expression =
    'is_',[2]name, '(', expression, ')'
    | is basic type, '(', expression, ')';

type judgement =
    'is_', '(', expression, ',', type, ')';

17.16 The Precondition Expression

pre-condition expression =
    'pre_', '(', expression, [{ ',', expression }], ')';

17.17 Class Membership

isofclass expression =
    'isofclass', '(', name, expression, ')';

17.18 Names

name =
    identifier, ['.', identifier];

name list =
    name, { ',', name };

old name =
    identifier, '~';

[2] **Note:** no delimiter is allowed

18 State Designators

state designator =
    name
    | field reference
    | map or sequence reference;

field reference =
    state designator, '.', identifier;

map or sequence reference =
    state designator, '(', expression, ')';

19 Patterns and Bindings

19.1 Patterns

pattern =
    pattern identifier
    | match value
    | tuple pattern
    | record pattern;

pattern identifier =
    identifier | '-';

match value =
    '(', expression, ')'
    | symbolic literal;

tuple pattern =
    'mk_', '(', pattern, ',', pattern list, ')';

record pattern =
    'mk_',[3]name, '(', [pattern list], ')';

pattern list =
    pattern, { ',', pattern };

19.2 Bindings

pattern bind =
    pattern | bind;

bind =
    set bind | type bind;

set bind =
    pattern, 'in set', expression;

type bind =
    pattern, ':', type;

bind list =
    multiple bind, { ',', multiple bind };

[3] **Note:** no delimiter is allowed

multiple bind =
    multiple set bind
    | multiple type bind;

multiple set bind =
    pattern list, '`in set`', expression;

multiple type bind =
    pattern list, '`:`', type;

type bind list =
    type bind, { '`,`', type bind };

keyword =
'abs' | 'all' | 'and' | 'atomic' | 'begin' | 'bool' | 'by' | 'card'
| 'cases' | 'chanset' | 'Chaos' | 'char' | 'class'
| 'comp' | 'compose' | 'conc' | 'dcl' | 'dinter' | 'div' | 'Div'
| 'do' | 'dom' | 'dunion' | 'elems' | 'else' | 'elseif' | 'end'
| 'endsby' | 'exists' | 'exists1' | 'extends' | 'false'
| 'floor' | 'for' | 'forall' | 'frame' | 'functions' | 'hd' | 'if' | 'in'
| 'inds' | 'inmap' | 'instance' | 'int' | 'inter' | 'initial'
| 'inv' | 'inverse' | 'iota' | 'is' | 'isofbaseclass' | 'isofclass'
| 'lambda' | 'len' | 'let' | 'logical' | 'map' | 'measure' | 'merge'
| 'mod' | 'mu' | 'munion' | 'nat' | 'nat1' | 'new' | 'nil' | 'not' | 'of'
| 'operations' | 'or' | 'others' | 'post' | 'power' | 'pre' | 'private'
| 'process' | 'protected' | 'psubset' | 'public' | 'rat' | 'rd'
| 'real' | 'rem' | 'res' | 'responsibility' | 'return' | 'reverse'
| 'rng' | 'samebaseclass' | 'sameclass' | 'self' | 'seq' | 'seq1'
| 'set' | 'Skip' | 'specified' | 'startsby' | 'state' | 'Stop'
| 'subclass' | 'subset' | 'then' | 'tl' | 'to' | 'token' | 'true'
| 'types' | 'undefined' | 'union' | 'val' | 'values' | 'vres'
| 'Wait' | 'while' | 'wr'
| 'mk_'
| 'pre_'
| 'post_'
| 'init_'
| 'is_' ;

identifier =
initial letter, { following letter } ;

is basic type =
'is_',('bool' | 'nat' | 'nat1' | 'int' | 'rat' | 'real' | 'char' | 'token') ;

symbolic literal =
numeric literal | boolean literal
| nil literal | character literal
| text literal | quote literal ;

numeral =
digit, { digit };

numeric literal =
decimal literal | hexadecimal literal;

exponent =
('E' |'e'), ['+' | '-'], numeral;

decimal literal =
numeral, ['.', digit, { digit }], [exponent];

hexadecimal literal =
('0x' | '0X'), hexadecimal digit, { hexadecimal digit };

boolean literal =
'true' | 'false';

nil =
'nil';

character literal =
'', (character | escape sequence), '';

escape sequence =

'\\' | '\r' | '\n' | '\t' | '\f' | '\e' | '\a'
| '\x', hexadecimal digit, hexadecimal digit
| '\u', hexadecimal digit, hexadecimal digit, hexadecimal digit, hexadecimal digit
| '\c', character
| '\', octal digit, octal digit, octal digit
| '\"' |'\'';

text literal =
'"', { '\"' | character | escape sequence }, '"' ;

quote literal =
'<', identifier, '>' ;

Single-line comment =
'-''-', { character – newline }, newline ;

Multiple-line comment =
'/*', { character }, '*/';

newline =
'\r' | '\n' | '\r\n' ;

References

[1] Communicating Sequential Processes. `en.wikipedia.org/wiki/Communicating_sequential_processes` Accessed June 15, 2012.

[2] *Circus*, `http://www.cs.york.ac.uk/circus/` Accessed June 15, 2012.

[3] Formal Systems (Europe) Ltd, *Failures-Divergence Refinement*, 19th October, 2010. `www.fsel.com/documentation/fdr2/fdr2manual.pdf` Accessed June 15, 2012.

[4] C. A. R. Hoare, *Communicating Sequential Processes*, Prentice Hall, 1986.

[5] J. S. Fitzgerald and P. G. Larsen, *Modelling Systems: Practical Tools and Techniques in Software Engineering*, Cambridge University Press, ISBN 0-521-62348-0, 1998.

[6] A. W. Roscoe, *The Theory and Practice of Concurrency*, Prentice Hall, ISBN 0-13-674409-5, 1997

[7] C. B. Jones, *Systematic Software Development using VDM*, Prentice Hall, International Series in Computer Science, 2nd ed., ISBN 0-13-880733-7, 1990. `www.csr.ncl.ac.uk/vdm/ssdvdm.pdf.zip` Accessed June 15, 2012.

[8] Peter Gorm Larsen and Kenneth Lausdahl and Nick Battle, *VDM-10 Language Manual*, Overture—Open-source Tools for Formal Modelling, TR-2010-06, April 2010.

[9] ISO, *Information technology: Programming languages, their environments and system software interfaces — Vienna Development Method — Specification Language — Part 1: Base language International Standard*, International Organization for Standardization, ISO/IEC 13817-1, December, 1996.

[10] Vienna Development Method, `en.wikipedia.org/wiki/Vienna_Development_Method`, Accessed June 15, 2012.