

Grant Agreement: 287829

Comprehensive Modelling for Advanced Systems of Systems



Fourth Release of the COMPASS Tool CML Grammar Reference

Deliverable Number: D31.4c

Version: 1.0

Date: September 2014

Public Document



Contributors:

Joey W. Coleman, Aarhus

Editors:

Joey W. Coleman, Aarhus

Reviewers:

Uwe Schulze, Bremen Richard Payne, Newcastle Adrian Larkham, Atego



Document History

Ver	Date	Author	Description
0.1	02-12-2013	JWC	Initial version based on D31.3c
0.2	08-07-2014	JWC	Update based on current grammar file & tool state.
0.3	16-07-2014	JWC	Remove stray action combinator
0.4	19-08-2014	JWC	Editing from internal review
1.0	01-09-2014	JWC	Ready for delivery



Contents

1	Introduction 1.1 Definition (Meta-)Syntax 1.2 Major changes relative to previous versions	5 5
2	Top Level model	7
3	Declarations	7
4	Classes	9
5	Processes	10
6	Actions	12
7	Statements	13
8	Types	15
9	Operations	17
10	Functions	18
11	Expressions	19
12	Patterns	22
13	Lexical Specification	23
Re	ferences	27
Ind	dex	28



1 Introduction

The purpose of this document is to provide a reference for the grammar of the COM-PASS Modelling Language (CML), as it is accepted by the Symphony IDE. This document is a reference document and, as such, makes no attempt to explain the purpose of any of the constructs that the defined syntax corresponds to, as this is not in the scope of the Theme 3 work. For the semantics of CML, please see [BCW14].

This document accompanies version 0.4.0 of the Symphony IDE, corresponding to the COMPASS Deliverable D31.4 release.

This document supports several useful activities:

- 1. users of the tool may use this document as a reference to ensure that their models conform to the format that the tool expects;
- 2. we can compare the syntax of the language the tool accepts against the semantic and syntactic definitions produced in Theme 2 for discrepancies; and,
- 3. members of the project have a basis for discussions regarding the superficial structure of the language that is neither the running code nor the semantics.

The second and third points were critical for maintaining the tool in the face of changes to the CML language as the project progressed. This document, in its previous versions, was used as a reference in to clarify conversations about both the tool and the language.

The first point is meant to be taken as a complement to tutorial materials, not as a replacement for them. The initial tutorials for CML are critical for users to gain an understanding of how to use the language; this document is intended to clarify the specific details of how to express models in CML.

1.1 Definition (Meta-)Syntax

The syntax used in this document to define the CML syntax is a variation of the usual Extended BNF syntax commonly used elsewhere. Rule definitions start with the name of the rule, then an equality symbol, =, then the rule definition body, then a semi-colon.

A rule definition body is a list of alternatives separated by vertical bars, I, and each alternative is a comma-separated list of components. Each component may be a literal string, which is a terminal symbol, a reference to another rule, or a bracketed sublist of components indicating either the optional presence of the sublist in that alternative, or the Kleene closure of the sublist. Examples of this are presented in Figure 1.

The example rule presented in Figure 2 uses all of the features permitted in our grammar format. It is named *example rule*, has two alternative productions, has terminal symbols and rule references, and uses the optional item and sequence braces.



Example	Explanation
'literal'	A literal value indicating the characters between the quota-
	tion marks.
map expression	A reference to the rule "map expression".
'inv', expression	The literal characters inv followed by something satisfying
	the <i>expression</i> rule.
{ <i>bind</i> }	A (possibly empty) sequence of things, each satisfying the
	bind rule.
[':', type]	Either empty or the concatenation of a colon and then some-
	thing that satisfies the <i>type</i> rule.

Figure 1: Examples of definition elements used in this document.

```
example rule →
    'terminal symbol', example rule, { 'optional sequence' }
    I'alternative case', ['optional single']
    ;
```

Figure 2: An example grammar rule.

1.2 Major changes relative to previous versions

The proposed syntax for configuration blocks that previously appeared in Deliverable D31.3c [Col13] has been removed from this version. The impact should be minimal to non-existent: no models in the project used the syntax, only the parser implemented the necessary functionality, and only in an experimental branch.

All forms of synchronous parallelism –parallel combinators that do not have specify chansets, but not including interleaving– have been removed. Their inclusion was an oversight, as they were not intended to be a part of the CML language.

The divergent action keyword 'Div' has been renamed 'Diverge' due to a parsing/recognition problem. The chaotic action keyword 'Chaos' has been removed from the language, in part because of a difference between the definition of that action between Hoare's and Roscoe's semantics for CSP. Using just 'Diverge' suits the needs of the project and language, and avoids confusion.



2 Top Level model

```
model →
    model paragraph, { model paragraph }
    ;

model paragraph →
    type declarations
    | function declarations
    | value declarations
    | channel declarations
    | chanset declarations
    | class declaration
    | process declaration
    ;
```

3 Declarations



```
'chansets', { chanset definition }
chanset definition \rightarrow
   identifier, '=', chanset expression
nameset\ declarations \rightarrow
   'namesets', { nameset definition }
   ;
nameset\ definition \rightarrow
   identifier, '=', nameset expression
   ;
state\ declarations \rightarrow
   'state', { instance variable definition }
instance \ variable \ definition \rightarrow
   [ qualifier ], assignment definition
   I invariant definition
   ;
assignment definition \rightarrow
   identifier, ':', type, [ ':=', expression ]
   ;
invariant definition \rightarrow
   'inv', expression
   ;
process\ declaration \rightarrow
   'process', identifier, '=', [parametrisation, { ', ', parametrisation}, '@'],
   process
```

Note: the only parametrisation qualifier allowed in a process declaration is 'val'. (Omitting a parametrisation qualifier defaults to 'val', and is permitted as well.)

```
parametrisation \rightarrow
   [ parametrisation qualifier ], identifier, { ', ', identifier }, ':', type
parametrisation qualifier \rightarrow
   'val'|'res'|'vres'
action\ declarations 
ightarrow
   'actions', { action definition }
   ;
action definition \rightarrow
   identifier, '=', [ parametrisation, { ', ', parametrisation }, '@' ], action
   ;
chanset\ expression 
ightarrow
   identifier
   | '{', [ identifier, { ', ', identifier } ], '}'
   | '{ | ', [ identifier, { ', ', identifier } ], '| }'
   ['{|', identifier, { '.', expression }, '|' bind list, ['@', expression ], '|}'
   I chanset expression, 'union', chanset expression
   I chanset expression, 'inter', chanset expression
   I chanset expression, '\', chanset expression
   ;
nameset\ expression 
ightarrow
   chanset expression
   ;
```

4 Classes

```
class declaration →
    'class', identifier, [ 'extends', identifier ], '=', 'begin', { class paragraph },
    'end'
   ;
```



```
class paragraph →
type declarations
| value declarations
| function declarations
| operation declarations
| state declarations
| 'initial', operation definition
.
```

5 Processes

```
process \rightarrow
   action process
   | process, ';', process
   | process, '[]', process
   | process, '|~|', process
   | process, '[|', chanset expression, '|]', process
   I process, '[', chanset expression, '||', chanset expression, ']', process
   | process, '| | | ', process
   | process, '/_\', process
   | process, '/_', expression, '_\', process
   | process, '[_>', process
   | process, '[_', expression, '_>', process
   | process, '\\', chanset expression
   I process, 'startsby', expression
   I process, 'endsby', expression
   l'(', parametrisation, {',', parametrisation}, '@', process, ')', '(', expression, {
   ',', expression }, ')'
   | identifier, ['(', [expression, {',', expression }],')']
   I process, renaming expression
   | replicated process
   1 '(', process, ')'
action\ process \rightarrow
   'begin', { action paragraph }, '@', action, 'end'
   ;
replicated\ process \rightarrow
   ';', replication declarations, '@', process
   I'[]', replication declarations, '@', process
```



```
| '|~|', replication declarations, '@', process
| '|||', replication declarations, '@', process
| '||', chanset expression, '||', replication declarations, '@', process
| '||', replication declarations, '@', '[', [chanset expression], ']', process
;

action paragraph →
   type declarations
| value declarations
| function declarations
| operation declarations
| action declarations
| nameset declarations
| state declarations
;

renaming expression →
   '[[', renaming pair, { ', ', renaming pair }, ']]'
| '[[', renaming pair, '|' bind list, [ '@', expression], ']]'
;
```

Note that the current parser only supports a single expression after an identifier in a *renaming pair*; this will be corrected in a future release.

```
renaming pair →
    identifier, { '.', expression }, '<-', identifier, { '.', expression }
    ;

replication declarations →
    replication declaration, { ', ', replication declaration }
    ;

replication declaration →
    identifier, { ', ', identifier }, ':', type
    | identifier, { ', ', identifier }, 'in' 'set', expression
    ;
```

COMPASS

6 Actions

```
action \rightarrow
   identifier
   I'Skip'
  I'Stop'
   I'Diverge'
   I'Wait' expression
   I communication, '->', action
   l'[', expression, ']', '&', action
   laction, ';', action
   laction, '[]', action
   | action, '|~|', action
   l action, '/_\', action
   l action, '/_', expression, '_\', action
   | action, '[_>', action
   | action, '[_', expression, '_>', action
   | action, '\\', chanset expression
   laction, 'startsby', expression
   laction, 'endsby', expression
   Laction, renaming expression
   | 'mu', identifier, { ', 'identifier }, '@', '(', action, { ', 'action }, ') '
   I parallel action
   I parametrised action
   I'(', action, ')'
   I instantiated action
   I replicated action
   | statement
communication \rightarrow
   identifier, { communication parameter }
   ;
communication\ parameter 
ightarrow
   '?', bindable pattern, [ ':', '(', expression, ')']
   l'!', parameter
   l'.', parameter
   ;
parameter \rightarrow
   identifier
```



```
(' expression ') '
   I symbolic literal
   I tuple expression
   | record expression
parallel\ action \rightarrow
   action, '|||', action
   | action, '[||', nameset expression, '|', nameset expression, '||]', action
   Laction, '[', chanset expression, '||', chanset expression, ']', action
   laction, '[', nameset expression, '|', chanset expression, '||', chanset expression,
   '|', nameset expression, ']', action
   l action, '[|', chanset expression, '|]', action
   l action, '[|', nameset expression, '|', chanset expression, '|', nameset
   expression, '|]', action
parametrised\ action 
ightarrow
   '('parametrisation, {',', parametrisation}, '@', action, ')'
instantiated\ action \rightarrow
   parametrised action, '(', expression, { ', ', expression }, ')'
   ;
replicated action \rightarrow
   ';', replication declarations, '@', action
   l'[]', replication declarations, '@', action
   I'|~|', replication declarations, '@', action
   I'|||', replication declarations, '@', '[', [nameset expression], ']', action
   l'[|', chanset expression'|]', replication declarations, '@', '[', [ nameset
   expression ], ']', action
   I'||', replication declarations, '@', '[', [ nameset expression ], '|', [ chanset
   expression], ']', action
   ;
```

7 Statements

```
statement →
'let', local definition, { ',', local definition }, 'in', action
```



```
I'(', ['dcl', assignment definition, {',', assignment definition}, '@'], action,
   | cases statement
   I if statement
   | 'if' non-deterministic alt, { '|', non-deterministic alt }, 'end'
   I 'do' non-deterministic alt, { '|', non-deterministic alt }, 'end'
   I'while', expression, 'do', action
   l'for', bindable pattern, [':', type]'in', expression, 'do', action
   I'for', 'all', bindable pattern, 'in set', expression, 'do', action
   I 'for', identifier, '=', expression, 'to', expression, ['by', expression], 'do',
   l'[',[frame],['pre', expression], 'post', expression, ']'
   l'return', [expression]
   l assign statement
   I multiple assign statement
   I call statement
   I new statement
   ;
local\ definition \rightarrow
   value definition
   I function definition
   ;
non-deterministic alt \rightarrow
   expression, '->', action
   ;
if statement \rightarrow
   'if', expression, 'then', action, { elseif statement }, [ 'else', action ]
elseif statement \rightarrow
   'elseif', expression, 'then', action
   ;
\textit{cases statement} \rightarrow
   'cases', expression, ':', cases statement alt, { ', ', cases statement alt }, [ ', ',
   others statement ], 'end'
```



```
cases statement alt \rightarrow
   pattern list, '->', action
others statement \rightarrow
   'others', '->', action
   ;
\textit{assign statement} \rightarrow
   assignable expression, ':=', expression
multiple \ assign \ statement 
ightarrow
   'atomic', '(', assign statement, ';', assign statement, { ';', assign statement },
   ') '
   ;
call\ statement 
ightarrow
   name, '(', [expression, {',', expression }], ')'
   | assignable expression, ':=', name, '(', [expression, { ', ', expression } ], ')'
new\ statement \rightarrow
   assignable expression, ':=', 'new', name, '(', [ expression, { ', ', expression } ],
   ') '
   ;
      Types
8
type\ declarations 
ightarrow
   'types', [ type definition, { '; ', type definition } ]
   ;
type\ definition \rightarrow
   [ qualifier ], identifier, '=', type, [ type invariant ]
   [ qualifier ], identifier, '::', { field }, [ type invariant ] }
```

```
type \rightarrow
   '(', type, ')'
   I basic type
   I quote literal
   I 'compose', identifier, 'of', { field }, 'end'
   | type, '|', type, { '|', type }
   | type, '*', type, { '*', type }
   | '[', type, ']'
   I'set' 'of', type
   I'seq' 'of', type
   I'seq1' 'of', type
   I'map', type, 'to', type
   l'inmap', type, 'to', type
   I function type
   1 name
   ;
basic\ type \rightarrow
   'bool'| 'nat'| 'nat1'| 'int'| 'rat'| 'real'| 'char'| 'token'
   ;
\mathit{field} \rightarrow
   type
   | identifier, ':', type
   | identifier, ':-', type
   ;
function type \rightarrow
   discretionary type, '+>', type
   | discretionary type, '->', type
discretionary\ type 
ightarrow
   type | '()'
   ;
type\ invariant \rightarrow
   'inv', pattern, '==', expression
   ;
```



9 Operations

Operations do not include reactive constructs; while the parser will accept any action in an operation body, the typechecker will only allow statements, the ';' sequential composition operator, and the constant action 'Skip'. In essence, operation bodies in CML allow only what is allowed in VDM operation bodies.

```
operation\ declarations \rightarrow
   'operations', { operation definition }
operation \ definition \rightarrow
   explicit operation definition
   I implicit operation definition
   ;
explicit operation definition \rightarrow
   [ qualifier ], identifier, ':', operation type, identifier, parameters, '==', operation
   body, ['pre', expression], ['post', expression]
operation \ type \rightarrow
   discretionary type, '==>', discretionary type
operation body \rightarrow
   action
   I'is subclass responsibility'
   I'is not yet specified'
implicit operation definition \rightarrow
   [ qualifier ], identifier, parameter types, [ identifier type pair list ], [ frame ], [
   'pre', expression], 'post', expression
   ;
frame \rightarrow
   'frame', var information, { var information }
```



10 Functions

```
function declarations \rightarrow
    'functions', { function definition }
   ;
function definition \rightarrow
   explicit function definition
   I implicit function definition
explicit function definition \rightarrow
   [ qualifier ], identifier, ':', function type, identifier, parameters list, '==', function
   body, ['pre', expression], ['post', expression], ['measure', name]
   ;
parameters\ list 
ightarrow
   parameters, { parameters }
parameters \rightarrow
    '(', [ pattern list ], ')'
implicit function definition \rightarrow
   [ qualifier ], identifier, parameter types, identifier type pair list, [ 'pre', expression
   ], 'post', expression
   ;
parameter\ types \rightarrow
   '(', [ pattern list, ':', type, { ', ', pattern list, ':', type } ], ') ' }
```

```
identifier type pair list \rightarrow
  identifier, ':', type, { ', ', identifier, ':', type }
;

function body \rightarrow
  expression
  | 'is not yet specified'
  | 'is subclass responsibility'
  ;
```

11 Expressions

```
expression \rightarrow
   'self'
  l name
  I old name
  I symbolic literal
  1 '(', expression, ')'
  I unary operator, expression
  Lexpression, binary operator, expression
  l'let', local definition, { ', ', local definition }, 'in', expression
  I'forall', bind list, '@', expression
  I'exists', bind list, '@', expression
  l'exists1', bind, '@', expression
  l'iota', bind, '@', expression
  I'lambda', type bind list, '@', expression
  | 'is_', '(', expression, ', ', type, ')'
  'is_', basic type, '(', expression, ')'
  'is_', name, '(', expression, ')'
  'pre_', '(', expression, { ', ', expression }, ')'
  l'isofclass', '(', name, expression, ')'
  I tuple expression
  | record expression
  | set expression
  I sequence expression
  | subsequence
  I map expression
  I if expression
  | cases expression
  I apply
  I field select
  I tuple select
```

```
;
name \rightarrow
   identifier, [ '.', identifier ]
  ;
old name \rightarrow
  identifier, '~'
  ;
\textit{unary operator} \rightarrow
   '+'|'-'|'abs'|'floor'|'not'|'card'|'power'|'dunion'|'dinter'|
  'hd'|'tl'|'elems'|'inds'|'reverse'|'conc'|'dom'|'rng'|
   'merge'l'inverse'
binary\ operator 
ightarrow
   '+'|'-'|'*'|'/'|'div'|'rem'|'mod'|'<'|'<='|'>'|'>='|'='|'<>'|'or'|
   'and'|'=>'|'<=>'|'in''set'|'not''in''set'|'subset'|
   'union'|'\'|'inter'|'^'|'++'|'munion'|'<:'|'<-:'|':>'|
   'comp'|'**'
  ;
tuple\ expression 
ightarrow
   'mk_', '(', expression, ', ', expression, { ', ', expression }, ') '
record\ expression \rightarrow
   'mk_', 'token', '(', expression, ')'
  | 'mk_', name, '(', [expression, { ', ', expression } ], ') '
  ;
set\ expression \rightarrow
   `{', [ expression, { ', ', expression } ], '}'
  [ '{', expression, '|', bind list, ['@', expression], '}'
  | '{', expression, ', ', '...', ', expression, '}'
  ;
sequence\ expression 
ightarrow
```

```
'[', [ expression, { ', ', expression } ], ']'
   | '[', expression, '|', set bind, ['@', expression], ']'
subsequence \rightarrow
   expression, '(', expression, ', ', ', ...', ', expression, ')'
   ;
map\ expression 
ightarrow
   `{', `|->', `}'
   | '{', maplet, { ', ', maplet }, '}'
   | '{', maplet, '|', bind list, ['@', expression], '}'
   ;
maplet \rightarrow
   expression, '|->', expression
apply \rightarrow
   expression, '(', [ expression, { ', ', expression } ], ')'
   ;
field select \rightarrow
   expression, '.', identifier
   ;
tuple\ select \rightarrow
   expression, '. #', numeral
if\ expression 
ightarrow
   'if', expression, 'then', expression, { elseif expression }, 'else', expression
elseif expression \rightarrow
   'elseif', expression, 'then', expression
```



12 Patterns

```
pattern →
  bindable pattern
| match value
;

bindable pattern →
  '_'
| identifier
| 'mk_', '(', pattern, ', ', pattern list, ')'
| 'mk_', name, '(', [ pattern list ], ')'
;

match value →
  '(', expression, ')'
| symbolic literal
.
```



```
pattern\ list \rightarrow
   pattern, { ', ', pattern }
bind \rightarrow
   set bind
   I type bind
set bind \rightarrow
   pattern, 'in' 'set', expression
type bind \rightarrow
   pattern, ':', type
bind list \rightarrow
   multiple bind, { ', ', multiple bind }
   ;
multiple\ bind \rightarrow
   pattern list, 'in' 'set', expression
   | pattern list, ':', type
type bind list \rightarrow
    type bind, { ', ', type bind }
```

13 Lexical Specification

[Please note: the parser's implementation of this is still incomplete. For now it's probably best to stick within the ASCII character set.]

Unlike the rest of this specification, the rules in this section are sensitive to whitespace; as such, whitespace may not implicitly separate any pair of components in a rule here.

Note that the unicode character categories can be found online at http://www.fileformat.info/info/unicode/category/index.htm. The present release of the tool only supports characters below U+0100; support for characters outside of the extended ASCII subset of unicode is planned for a future release.

```
initial letter \rightarrow if 'codepoint < U+0100' then Any character in categories Ll, Lm, Lo, Lt, Lu, or the character 'U+0024' ('\$') else Any character, excluding categories Cc, Zl, Zp, Zs, Cs, Cn, Nd, Pc.;
```

```
following letter → if 'codepoint < U+0100' then Any character in categories Ll, 
Lm, Lo, Lt, Lu, Nd, or the characters 'U+0024' ('\$'), 'U+0027' ('''), and 
'U+005F' ('_') else Any character, excluding categories Cc, Zl, Zp, Zs, Cs, Cn.
```

```
ascii letter \rightarrow Any character in the ranges ['U+0041', 'U+005A'] and ['U+0061', 'U+007A'] --- A-Z and a-z, respectively.;
```

 $character \rightarrow$ Is left underdefined, except to note that it may be any unicode character except those that conflict with the lexical rule that uses the character class. For example, character does not include '\' in the *character literal* rule.;

```
identifier \rightarrow
   initial letter, { following letter }
   ;
digit \rightarrow
   '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'
hex\ digit \rightarrow
   digit | 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'A' | 'B' | 'C' | 'D' | 'E' | 'F'
   ;
numeral \rightarrow
   digit, { digit }
symbolic\ literal \rightarrow
   numeric literal
   I boolean literal
   I nil literal
   I character literal
   | text literal
   I quote literal
```



```
;
numeric\ literal \rightarrow
   decimal literal
   l hex literal
exponent \rightarrow
   ('E' | 'e'), [ '+' | '-'], numeral
decimal\ literal \rightarrow
   numeral, [ '.', digit, { digit } ], [ exponent ]
   ;
hex\ literal \rightarrow
   ('0x' | '0X'), hex digit, { hex digit }
boolean\ literal \rightarrow
   'true' | 'false'
nil\ literal \rightarrow
   'nil'
   ;
\textit{character literal} \rightarrow
   ", character, ",
   I'', escape sequence, "'
escape\ sequence 
ightarrow
   '\\'I'\r'\\n'\\t'\\f'\\e'\\\a'\\\"'\\x', hex digit, hex digit
   I'\u', hex digit, hex digit, hex digit
   I '\c', ascii letter
   ;
```





References

- [BCW14] Jeremy Bryans, Samuel Canham, and Jim Woodcock. CML definition 4. Technical report, COMPASS Deliverable, D23.5, March 2014. Available at http://www.compass-research.eu/.
- [Col13] Joey W. Coleman. Third release of the COMPASS tool CML grammar reference. Technical report, COMPASS Deliverable, D31.3c, November 2013.

Index of Rule Definitions

action, 9, 10, 12–15, 17	chanset declarations, 7
definition, 12	definition, 7
action declarations, 11	chanset definition, 8
definition, 9	definition, 8
action definition, 9	chanset expression, 8–13
definition, 9	definition, 9
action paragraph, 10	character, 25, 26
definition, 11	character literal, 24
action process, 10	definition, 25
definition, 10	class declaration, 7
apply, 19	definition, 9
definition, 21	class paragraph, 9
ascii letter, 25	definition, 10
assign statement, 14, 15	communication, 12
definition, 15	definition, 12
· ·	•
assignable expression, 15	communication parameter, 12
definition, 22	definition, 12
assignment definition, 8, 14	decimal literal, 25
definition, 8	definition, 25
basic type, 16, 19	digit, 24, 25
definition, 16	definition, 24
binary operator, 19	
• •	discretionary type, 16, 17
definition, 20	definition, 16
bind, 19	elseif expression, 21
definition, 23	definition, 21
bind list, 9, 11, 19–21	elseif statement, 14
definition, 23	definition, 14
bindable pattern, 7, 12, 14, 22	escape sequence, 25, 26
definition, 22	= =
boolean literal, 24	definition, 25
definition, 25	explicit function definition, 18
11	definition, 18
call statement, 14	explicit operation definition, 17
definition, 15	definition, 17
cases expression, 19	exponent, 25
definition, 22	definition, 25
cases expression alternatives, 22	expression, 7–23
definition, 22	definition, 19
cases statement, 14	6.11.15.16
definition, 14	field, 15, 16
cases statement alt, 14	definition, 16
definition, 15	field select, 19
channel declarations, 7	definition, 21
definition, 7	following letter, 24
channel name declarations, 7	frame, 14, 17
definition, 7	definition, 17

function body, 18	definition, 23
definition, 19	15 16 18 20 22
function declarations, 7, 10, 11	name, 15, 16, 18–20, 22
definition, 18	definition, 20
function definition, 14, 18	nameset declarations, 11
definition, 18	definition, 8
function type, 16, 18	nameset definition, 8
definition, 16	definition, 8
	nameset expression, 8, 13
hex digit, 25	definition, 9
definition, 24	new statement, 14
hex literal, 25	definition, 15
definition, 25	nil literal, 24
<i>y</i>	definition, 25
identifier, 7–12, 14–22, 26	non-deterministic alt, 14
definition, 24	definition, 14
identifier type pair list, 17, 18	numeral, 21, 22, 25
definition, 18	definition, 24
if expression, 19	
definition, 21	numeric literal, 24
if statement, 14	definition, 25
	-14 10
definition, 14	old name, 19
implicit function definition, 18	definition, 20
definition, 18	operation body, 17
implicit operation definition, 17	definition, 17
definition, 17	operation declarations, 10, 11
initial letter, 24	definition, 17
instance variable definition, 8	operation definition, 10, 17
definition, 8	definition, 17
instantiated action, 12	operation type, 17
definition, 13	definition, 17
invariant definition, 8	others statement, 14
definition, 8	definition, 15
acjumien, e	dejaman, 10
local definition, 13, 19	parallel action, 12
definition, 14	definition, 13
	parameter, 12
map expression, 19	definition, 12
definition, 21	parameter types, 17, 18
maplet, 21	definition, 18
definition, 21	•
match value, 22	parameters, 17, 18
definition, 22	definition, 18
model	parameters list, 18
	definition, 18
definition, 7	parametrisation, 8–10, 13
model paragraph, 7	definition, 9
definition, 7	parametrisation qualifier, 9
multiple assign statement, 14	definition, 9
definition, 15	parametrised action, 12, 13
multiple bind, 23	definition, 13

pattern, 16, 22, 23	tuple select, 19
definition, 22	definition, 21
pattern list, 15, 18, 22, 23	type, 7–9, 11, 14–16, 18, 19, 23
definition, 23	definition, 15
process, 8, 10, 11	type bind, 23
definition, 10	definition, 23
process declaration, 7	type bind list, 19
definition, 8	definition, 23
	type declarations, 7, 10, 11
qualifier, 7, 8, 15, 17, 18	definition, 15
definition, 7	type definition, 15
quote literal, 16, 24	definition, 15
definition, 26	type invariant, 15
	definition, 16
record expression, 13, 19	,
definition, 20	unary operator, 19
renaming expression, 10, 12	definition, 20
definition, 11	•
renaming pair, 11	value declarations, 7, 10, 11
definition, 11	definition, 7
replicated action, 12	value definition, 7, 14
definition, 13	definition, 7
replicated process, 10	var information, 17
definition, 10	definition, 18
replication declaration, 11	
definition, 11	
replication declarations, 10, 11, 13	
definition, 11	
selector, 22	
definition, 22	
sequence expression, 19	
definition, 20	
set bind, 21, 23	
definition, 23	
set expression, 19	
definition, 20	
state declarations, 10, 11	
definition, 8	
statement, 12	
definition, 13	
subsequence, 19	
definition, 21	
symbolic literal, 13, 19, 22	
definition, 24	
· y ·······	
text literal, 24	
definition, 26	
tuple expression, 13, 19	
definition, 20	